

CONFIDENTIAL

LA-UR- 93-341

Title: Supercomputer Debugging Workshop '92

RECEIVED

FEB 11 1993

O

LA-UR--93-341

DE93 007322

Author(s): Jeffrey S. Brown

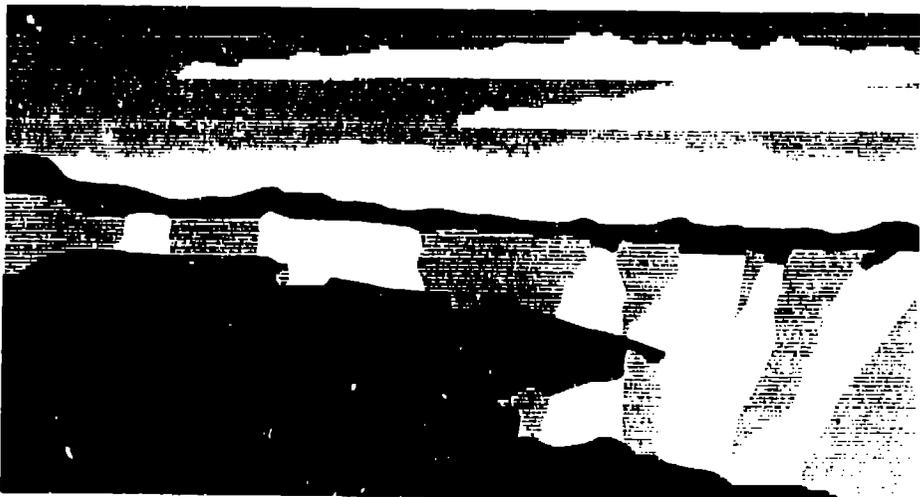
Submitted to: Supercomputing Debugging Workshop '92
Dallas, Texas
October 7-9, 1992

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

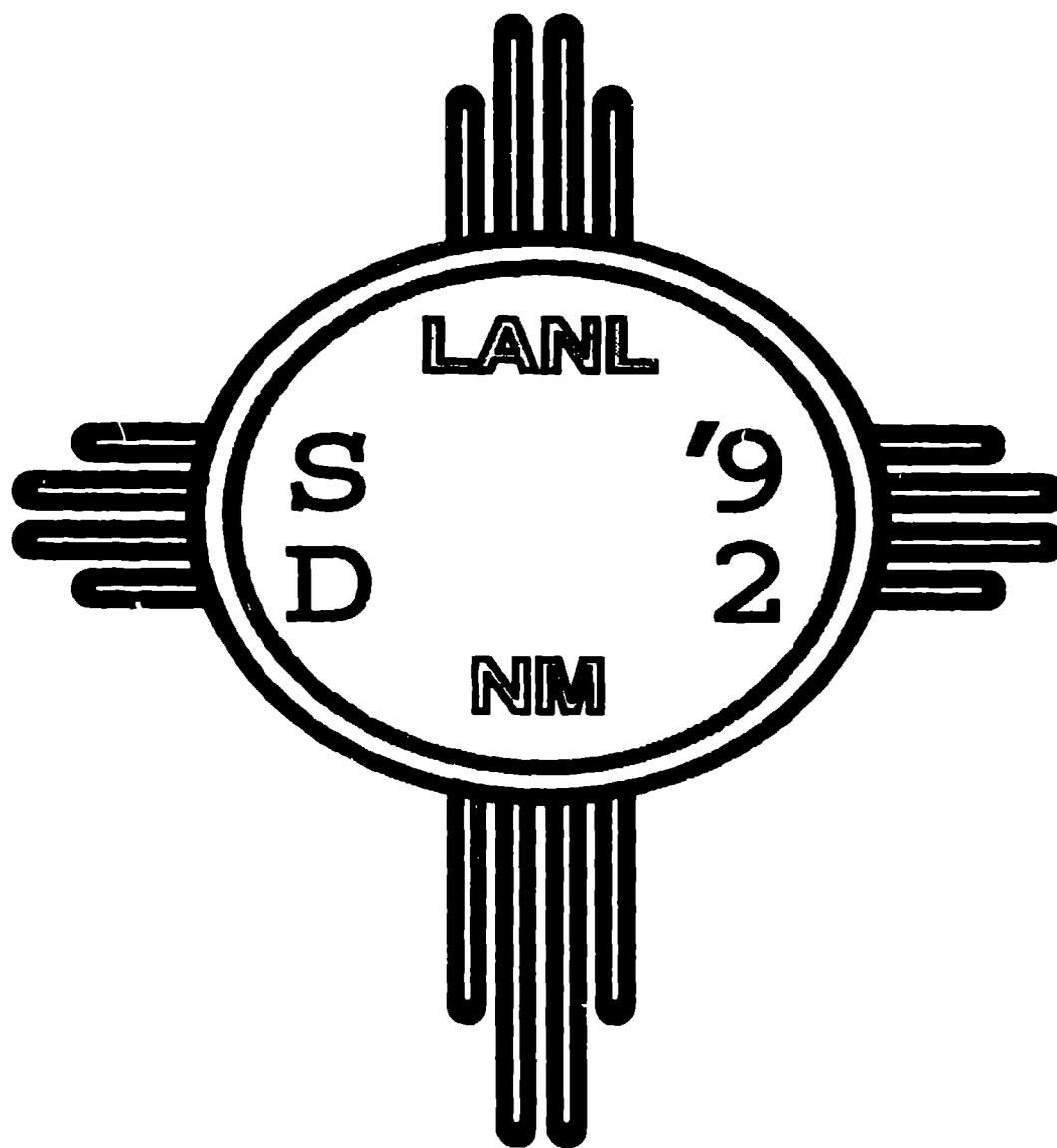
DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Los Alamos
NATIONAL LABORATORY



Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36. By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. The Los Alamos National Laboratory requests that the publisher identify the article as work of the U.S. Government.

Supercomputer Debugging Workshop '92



Dallas, Texas

October 7-9, 1992

MASTER

Sponsored by the Los Alamos National Laboratory

FORWARD

The Supercomputer Debugging Workshop '92 (SD '92) was held October 7-9 in Dallas, Texas. The workshop was sponsored by the Los Alamos National Laboratory and hosted by Convex Computer Corporation.

SD '92 focused on topics related to debugger construction and use in a high-performance computing environment. The workshop brought together debugger developers and users to discuss topics and experiences of mutual interest, and establish a basis for future collaborations.

The objective of the workshop was to promote a free and open exchange of information between an interdisciplinary group of debugger developers and users from the academic and commercial communities, thereby facilitating technology transfer and advancing the state-of-the-art of applied debugging technology.

Program Chair: *Larry Streepy, Convex Computer Corporation*

Program Committee: *Jeff Brown, Los Alamos National Laboratory*
Bart Miller, University of Wisconsin
Cherri Pancake, Oregon State University
Dennis Parker, Cray Research Incorporated
Rich Title, Thinking Machines Corporation
Ben Young, Cray Computer Corporation

Administrative Chair: *Denise Dalmas, Los Alamos National Laboratory*

Keynote Speaker: *Mark Linton, Silicon Graphics*

Table of Contents

	<u>Page</u>
Addresses of Participants	1
<u>Keynote Address</u>	
The ABCs of Debugging in the 1990s <i>Mark Linton, Silicon Graphics</i>	5
<u>User Perspectives</u>	
Los Alamos National Laboratory <i>Mike Clover and Johnny Collins</i>	25
National Center for Atmospheric Research <i>Jeff Kuehn</i>	35
<u>Vendor Updates</u>	
Cray Computer Corporation <i>Ben Young</i>	51
Thinking Machines Corporation <i>Rich Title</i>	63
Cray Research, Incorporated <i>Dennis Parker and Pete Johnson</i>	83
Sun Microsystems, Inc. <i>Ivan Soliemanipour</i>	89
Kendall Square Research <i>Steve Zimmerman</i>	95

<u>Papers</u>	<u>Page</u>
The Effects of Register Allocation and Instruction Scheduling on Symbolic Debugging <i>Ali-Reza Adi-Tabatabai and Thomas Gross,</i> Carnegie-Mellon University	115
Debugging Optimized Code: Currency Determination with Data Flow <i>Max Copperman,</i> University of California at Santa Cruz	127
A Debugging Tool for Parallel and Distributed Programs <i>Andreas Weininger,</i> Technische Universitat Munchen	147
Analyzing Traces of Parallel Programs Containing Semaphore Synchronization <i>D.P. Helmbold, C.E. McDowell, T. Haining,</i> UCSC	157
Compile-time Support for Efficient Data Race Detection in Shared-Memory Parallel Programs <i>John Mellor-Crummey,</i> Rice University	169
Direct Manipulation Techniques for Parallel Debuggers <i>Cherri M. Pancake,</i> Oregon State University	179
Transparent Observation of XENOOPS Objects <i>S. Bijnsens, W. Joosen, P. Verbaeten,</i> Department of Computer Science K.U. Leuven	209
A Parallel Software Monitor for Debugging and Performance Tools on Distributed Memory Multicomputers <i>Don Breazeal, Ray Anderson, Wayne D. Smith,</i> <i>Will Auld, Karla Callaghan</i> Intel Corporation	221
Profiling Performance of Inter-Processor Communications in an iWarp Torus <i>Thomas Gross and Susan Hinrichs,</i> Carnegie Mellon University	239
The Application of Code Instrumentation Technology in the Los Alamos Debugger <i>Jeff Brown and Richard Klamann,</i> Los Alamos National Laboratory	277
CXdb: The Road to Remote Debugging <i>Larry Streepy, Rob Gordon, Dave Lingle</i> Convex Computer Corporation	305
User Needs Discussion Summary	353

Addresses of Participants

Ali-Reza Adl-Tabatabai
Carnegie-Mellon University
ali@cs.cmu.edu

S. Bijnens
Department of Computer Science K.U. Leuven
stijn@cs.kuleuven.ac.be

John Blaylock
Los Alamos National Laboratory
jwb@lanl.gov

Brian Bliss
University of Illinois
bliss@csrd.uiuc.edu

Don Breazeal
Intel Supercomputer Systems Division
donb@ssd.intel.com

Gary Brooks
Convex Computer Corporation
gbrooks@pixel.convex.com

Jeff Brown
Los Alamos National Laboratory
jxyb@lanl.gov

Charlie Burns
Mercury Computer Systems
burns@mc.com

Mike Clover
Los Alamos National Laboratory
mrc@lanl.gov

Johnny Collins
Los Alamos National Laboratory
juan@lanl.gov

Max Copperman
University of California at Santa Cruz
max@cse.ucsc.edu

Charles Fineman
NASA Ames
fineman@ptolemy.arc.nasa.gov

Thomas Gross
Carnegie-Mellon University
Thomas.Gross@cs.cmu.edu

Theodore Haining
University of California at Santa Cruz
haining@cse.ucsc.edu

Carol Hakansson
Verdix Corporation
carolyn@verdix.com

Anthony Hefner
SAS Institute Incorporated
sasahr@unx.sas.com

Susan Hinrichs
Carnegie-Mellon University
shinrich@cs.cmu.edu

Peter Johnsen
Cray Research, Inc.
pjj@cray.com

Janis Johnson
ACSET
johnson@acset.be

Jeff Kuehn
National Center for Atmospheric Research
kuehn@ncar.ucar.EDU

Mark Linton
Silicon Graphics
linton@marktwain.rad.sgi.com

Glenn Luecke
Iowa State University
gm.grl@isumvs.iastate.edu

Jack McDonald
Microtec Research
jackmac@mri.com

Charlie McDowell
University of California at Santa Cruz
charlie@cse.ucsc.edu

Mike Meier
IBM PSC
meier@paloalto.vnet.ibm.com

John Mellor-Crummey
Rice University
johnmc@cs.rice.edu

Cherri Pancake
Oregon State University
pancake@cs.orst.edu

Dennis Parker
Cray Research, Inc.
dep@cray.com

John Roth
SAS Institute, Inc.
sasjor@unx.sas.com

Terry L. Sigle
Electronic Data Systems, Research and Development
tls@edsr.eds.com

Ivan Soliemanipour
Sun Microsystems
ivan@soliemanipour@Eng.Sun.Com

Eric Stotzer
Texas Instruments
eric@tools.micro.ti.com

Larry Streepy
Convex Computer Corporation
streepy@convex.com

Richard Title
Thinking Machines Corporation
title@think.com

Wheels Vanderweele
Verdix Corporation
wheels@verdix.com

Andreas Weininger
Technische Universitat Munchen
weininge@informatik.tu-muenchen.de

Ben Young
Cray Computer Corporation
bby@craycos.com

Steve Zimmerman
Kendall Square Research
z@ksr.com

The ABCs of Debugging in the 1990s

Mark Linton
Silicon Graphics Computer Systems
linton@sgi.com

Dbx evolution

Using adb/sdb

6,000-line MS project (pdx)

Port to VAX, 680x0, SPARC,
MIPS, RS/6000, ...

Support for C, F77, Modula-2, ...

25,000-line blob

Graphical front-ends (dbxtool)

Semi-retired

Dbx contributions

Organization – abstract data types

Portability – one week/machine

Multilingual support – depends on compiler

Ease of use – naive programmers

Surprise: Longevity of stabstrings

Simple design and implementation
(1 week on compiler, debugger)

Goal was to limit modifications
as, ld, nm, ar, ...

Other designs offer better clarity and
performance for specific environment

**BUT stabstrings were best choice
over Dbx's active lifetime**

Sadly, DWARF isn't a significant advance

Notable mistakes

Slow breakpoints

Slow to adopt regression tests

Replicating full environment

Symbol table management (compiler)

Expression evaluation (compiler)

Execution (shell)

Source viewing (editor)

Made the implementation easier?

The Next Generation

Old problems

Compiler optimizations

Integration with environment

User interface

New(er) problems

Higher-level languages

Wider variety machine architectures

Blending with performance analysis

As easy as ABC ...

A is for Abstraction

Machine's model of computation

Program's model of execution

User's model of (mis)understanding

Model of computation

**Derived from language, compiler,
operating system, machine**

Examples when debugging:

**Dynamic loading is part of model
Instruction pipelining is not**

Reflected in object->source mapping

Understood by user!

Debuggers allow access to ALL computational state

Non-local variables

Execution status (e.g., goal stack in Prolog)

Automatically-generated processes

Transparent access to network

Key difference from interpreters

Model of execution

Programs may contain more abstractions

Hidden control flow (e.g., method dispatch)

Hidden data (e.g., access functions)

Application-defined presentation

Examples from C++ and InterViews

stop in g->draw

stop in Button::draw

show dag(glyph)

For now, add DebugGlyph objects

Model of misunderstanding

Debugging to see what is going on

Task-oriented user interface

Tracing/watchpoints

Application-oriented presentation

Fast turnaround

B is for Big

Layer on top of lots of library code

Call up and down layers

Run for a long time

Allocate lots of memory (free some)

Process lots of data

Checkpoint execution state

Selective presentation

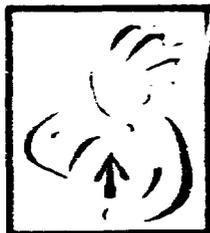
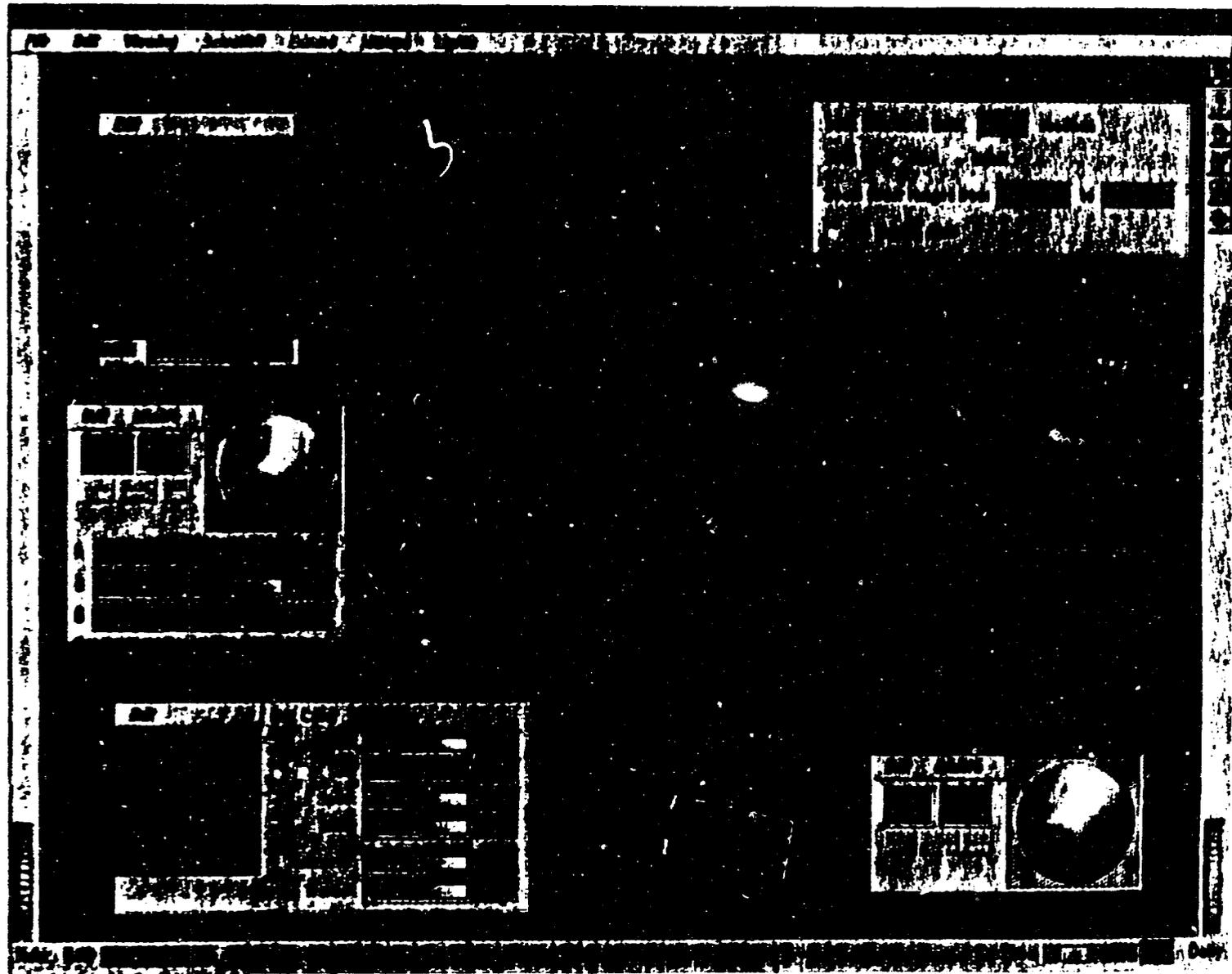
Show important information

Navigate to explore

Domain-specific visualization

Sharing views means sharing data model

Analogous to database problem



INVENTOR

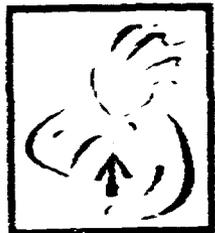


SiliconGraphics
Computer Systems

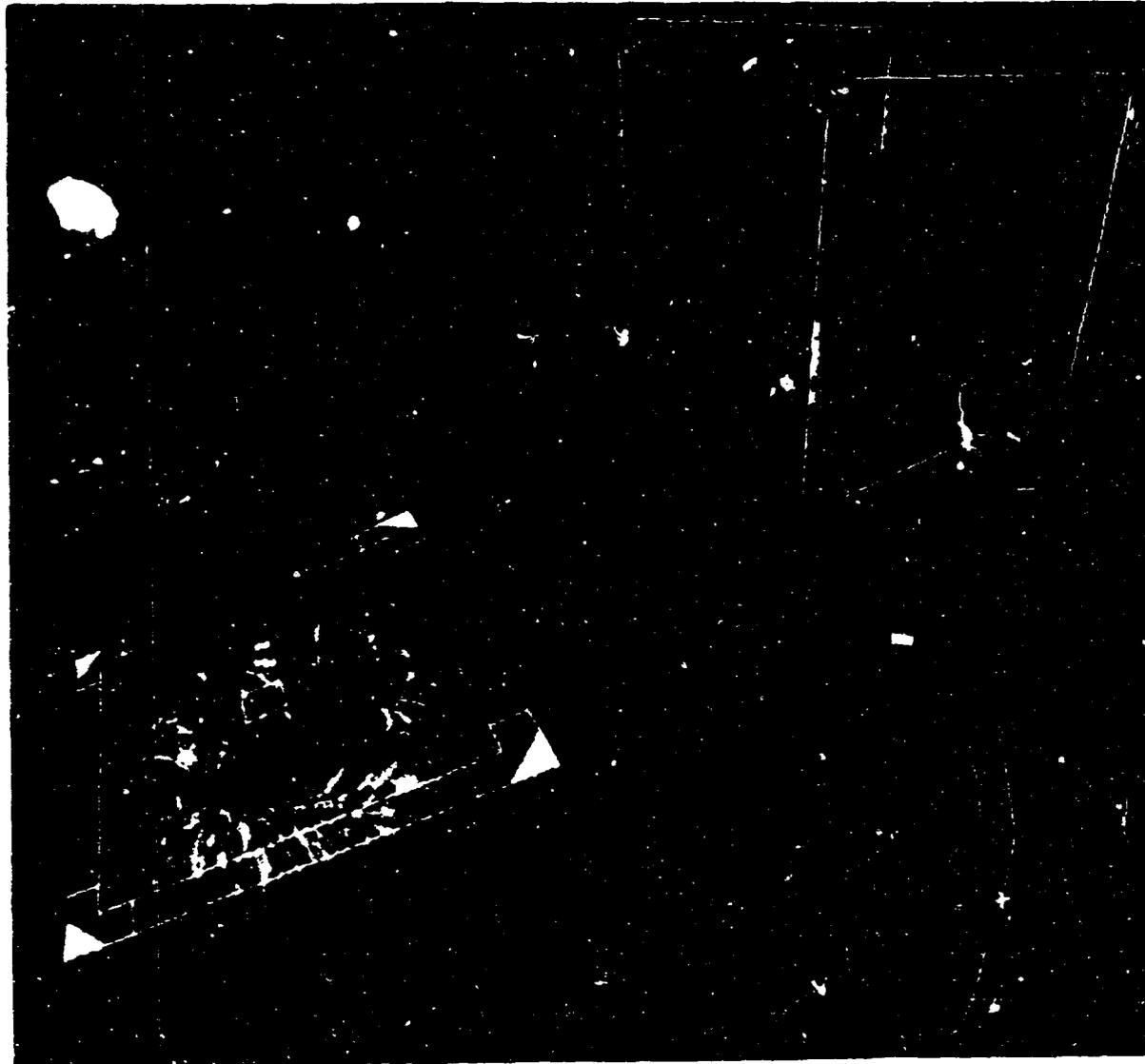
Interaction: Examples of Manipulators

Scale 1D

**Handle
Box**



INVENTOR



**Transform
Box**

Trackball

Document editor

Glyphs: Flyweight Objects for User Interfaces

6 Introduction

Most user interface toolkits use object-oriented design: program objects are a natural way to represent the objects that a user manipulates. However, current toolkits have tried very hard to wholeheartedly convert to the object-oriented model. These toolkits provide objects such as buttons and menus that let programmers build interfaces to applications, but they do not provide objects for building windows to applications. Without this support, programmers must often define many new components from scratch; the savings afforded by the toolkit is offset by the cost of handcrafting the new components.

6.1 Overview

To offer the full benefits of an object-oriented model, a toolkit must encourage programmers to use objects for even the smallest components in the interface. Current toolkits let programmers build interfaces that consist

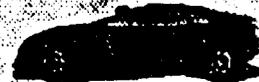
hundreds of objects. But to reflect the fine-grained structure in application data, programmers may need hundreds of thousands of objects. For example, text is logically composed of characters, so a text view should be built from objects that represent individual characters. A standard-sized document, such as a technical paper or a chapter in a book, will need at least 30,000 character objects.

6.2 Other toolkits

Most user interface toolkits use object-oriented design: program objects are a natural way to represent the objects that a user manipulates. However, current toolkits have tried very hard to wholeheartedly convert to the object-oriented model. These toolkits provide objects such as



TextView object structure



hundreds of objects. But to reflect the fine-grained structure in application data, programmers may need hundreds of thousands of objects. For example, text is logically composed of characters, so a text view should be built from objects that represent individual characters. A standard-sized document, such as a technical paper or a chapter in a book, will need at least 30,000 character objects.

This approach can dramatically simplify an implementation, but it is practical only if objects are simple and cheap. The combination of simple toolkits and more efficient object-oriented languages has now made it possible to use objects at this fine level of granularity without incurring excessive response.

We have designed a set of "Overlight" components, called Glyphs, that are simple and efficient. The Glyphs base class defines a protocol for drawing; subclasses define specific appearance such as graphic primitives (lines and

```

TextView::TextView(FILE* file) {
    TBox* page = new TBox();
    LBox* line = new LBox();
    int c;
    while ((c = getc(file)) != EOF) {
        if (c == '\n') {
            page->append(line);
            line = new LBox();
        } else {
            line->append(new Character(c));
        }
    }
    body(page);
}
    
```

A TextView built from glyphs

Request	Preference	IO
Allocation	run	10
Draw	update	30

Glyphs.html

C is for Concurrency

Performance-oriented

Simplicity-oriented

“The kernel never blocks.”

Users want it (e.g., printing)

Mainstream in ~3 years

Debugger challenges

Consistency checking (fast)

Symbolic execution/interpretation

Presenting large-scale concurrency

Concurrent debugging/analysis tools

A few wild ideas

Objects R Us

A debugging framework

Runtime compilation and optimization

Why objects?

Users like objects

Programmers like objects (sometimes)

Applications -> objects

Computational modules -> objects

Distributed objects

anywhere, somewhere, everywhere

A few wild ideas

Objects R Us

A debugging framework

Runtime compilation and optimization

Why objects?

Users like objects

Programmers like objects (sometimes)

Applications -> objects

Computational modules -> objects

Distributed objects

anywhere, somewhere, everywhere

A debugging framework

Simplify

Use existing compiler and editor

Expose internal components

Make available over network

Allow concurrent “users”

Runtime compilation and optimization

Make compiler invisible

Dynamically choose performance level

Simulate

Interpret

Compile

Optimize

Inline/optimize across linkage boundaries

Give optimizer direct access to profile

Source compatibility

Promising results from Self compiler

User Perspective

1 Large Code

46,055,728 bytes

2 The Code changes

We add about 10000 lines of mods every 4 months

3 Portions of the Code are Multi Tasked

4 We sometimes use LDB/DDT on a "DROP" file

Parallel Debugging

- 1 **An error in Our Multi Tasked package may not be reproducible**
(there is no replay scheduler capability)
- 2 **We use a DDT to debug Multi tasked code**
- 3 **The debugger will stop the code when any processor reaches a**
set breakpoint and reports the processor
(a conditional breakpoint on a particular processor would be nice).

Nice Things to have

- 1 **Parallel debugging of 2 versions with comparison**
- 2 **List all opened files and their attributes**
- 3 **Save and Restore**
- 4 **A FORTRAN representation of what the optimizer has done**
- 5 **Source listing provided in debuggers should designate new coding**
- 6 **The ability to spell**
- 7 **The option to deoptimize portions of the code at debug time**
- 8 **The ability to list the library a module comes from**
- 9 **Conditional breakpoints (that are fast)**

ldb prof.x

ldb version 1.3
built: 09/21/92 at 12:29:51

attached to absolute file: /usr/tmp/ld32231.copy (copy of prof.x)
entering debug mode ...

AUXX> list source

```
program auxx
dimension x(11),a0(10),a1(10),a2(10),delta(10),chkreg(22)
dimension vl(22),ebths(22)
pointer (mme, mne(1))
pointer (mmeq21, cg21(1))
pointer (mmaux0, aux0(1))
pointer (mmaux1, aux1(1))
AUXX (1) mme=loc(x(1))
mmeq21=loc(x(1))
mmaux0= loc(a0(1))
mmaux1= loc(a1(1))
mmaux2= loc(a2(1))
lkman=10
S10A do 10 i=1,10
aux0(i)=1
aux1(i)=1
mne(i)=1
chkreg(i)=1.
j=2*i
chkreg(j)=1.
vl(i)=1.
vl(j)=1.
```

AUXX> roll source 50

```
S10B 10 continue
chkreg(22)=1.
mne(11)=11.
id1=2
j=2
cdirs nextscalar
S180A do 180 lk=2,lkman
j=j+id1
aux0(lk)=chkreg(j)*aux0(lk)*mne(lk)
aux1(lk)=chkreg(j+id1)*aux1(lk)*mne(lk+1)
delta(lk)=aux0(lk)-aux1(lk)
ebths(j)=mne(lk)*vl(j)
cg21(lk)=0.
S180
S180B 180 continue
write(" ",(aux1(i),i=1,10))
stop
end
```

AUXX> run to S180a

user process stopped at program counter: 371pa = S180A @ AUXX()

AUXX> id1, j, chkreg(4), aux0(4), mne(4)

00000243752b: id1 = 2

00000243753b: j = 2

00000243701b: chkreg(4) = 1.000000e+00

00000243632b: aux0(4) = 4.000000e-00

00000243615b: mne(4) = 4.000000e+00

AUXX> print chkreg(4)*aux0(4)*mne(4)

1.600000e+01

AUXX> bkp S180b

AUXX> run

user process stopped at program counter: 417pd = S180B @ AUXX()

AUXX> aux0(4)

00000243632b: aux0(4) = 1.600000e-01

AUXX> end

killing user process ...

ok

ldb version 1.3
built: 09/21/92 at 12:29:51

attached to absolute file: /usr/tmp/ld32000.copy (copy of prof.x)
entering debug mode ...
AUXX> run to \$180a
user process stopped at program counter: 451pa = \$180A @ AUXX()

```
AUXX> list source
$10B      10      .
           continue
           chkreg(22)=1.
           xne(11)=11.
           id1=2
           j=2
-> $180A    do 180 lk=2,lkmax
           j=j+id1
           aux0(lk)=chkreg(j)*aux0(lk)*xne(lk)
           aux1(lk)=chkreg(j+id1)*aux1(lk)*xne(lk+1)
           delta(lk)=aux0(lk)-aux1(lk)
           ebths(j)=xne(lk)*v1(j)
           cg21(lk)=0.
$180B    180 continue
           write(*,*) (aux1(1),i=1,10)
           stop
           end
```

```
AUXX> j, id1, chkreg(4), aux0(4), xne(4)
00000244014b: j = 2
00000244013b: id1 = 2
00000243742b: chkreg(4) = 1.000000e+00
00000243721b: aux0(4) = .000000e+00
```

unable to complete di command
address exceeds length of data area

```
AUXX> mmaux0
00000243702b: mmaux0 = 0600000000000000000243670
AUXX> 0000000000000000000243670\4
00000243670b: 040001400000000000000000    0400n24000000000000000    0400024000000000000000
00000243673b: 040003400000000000000000
```

```
AUXX> dec
AUXX> 0000000000000000000243670\4
00000243670b: 1.000000e+00    2.000000e+00    3.000000e+00    4.000000e+00
```

```
AUXX> print chkreg(4)*4.*4.
1.600000e+01
```

```
AUXX> mmxne
00000243666b: mmxne = 83883
AUXX> 83883\4
00000243653b: 1.000000e+00    2.000000e+00    3.000000e+00    4.000000e+00
```

```
AUXX> bkp $180b
```

```
AUXX> run
user process stopped at program counter: 467pa = $180B @ AUXX()
```

```
AUXX> mmaux0
00000243702b: mmaux0 = 83896
AUXX> 83896\4
00000243670b: 1.000000e+00    0    0    0
```

```
AUXX> end
killing user process ...
```

e4

USER PERSPECTIVE

LOS ALAMOS NAT'L LAB

HARDWARE ENVIRONMENT

- CRAY YMP's
 - CTSS
 - UNICOS

SOFTWARE ENVIRONMENT

- PRODUCTION CODES (>100 kLOC)
 - optimized
 - vectorized
- FORTRAN
 - /common/ blocks
 - pointers
- MEMORY MANAGEMENT
 - run time array allocations

DEBUGGING ENVIRONMENT

- OPTIMIZED CODE
- COMMAND LINE INTERFACE
 - SAVE/RESTORE
 - CONDITIONAL BREAKPOINTS

16069	89411	3279.		if (and(jhqiter,8) .eq.0) then	mc7hyd2a	238
16070	89412	3280.		do 470 j=kmaxp2,kalm,incz	mc7hyd2a	239
16071	89413	3281.		jl = j - kmax	mc7hyd2a	240
16072	89414	3282.		dediabt(j)--(avfrq1(j)*ut(jl-1)+avfzq1(j)*vt(jl-1)	mc7hyd2a	241
16073	89415	3283.	6	+avfrq2(j)*ut(jl)+avfzq2(j)*vt(jl)	mc7hyd2a	242
16074	89416	3284.	6	+avfrq3(j)*ut(j)+avfzq3(j)*vt(j)	mc7hyd2a	243
16075	89417	3285.	6	+avfrq4(j)*ut(j-1)+avfzq4(j)*vt(j-1))	mc7hyd2a	244
16076	89418	3286.		dediabt(j)=dediabt(j)*dthyd*0.5	mc7hyd2a	245
16077	89419	3287.	470	continue	mc7hyd2a	246
16078	89420	3288.		else	mc7hyd2a	247
				:	mc7hyd2a	248
				:		
				end's		
				end f		

management and
blame allocation.

An atypical loop in Lasnex - it
would fit in a window.

MEMORANDUM FOR FILE # 100-12550-100

.di memo \$470a for 20

0A 01172525pb -	024741	A7	B41	
01172525pc -	002007	VL	A7	
01172525pd -	024662	A6	B62	
01172526pa -	024542	A5	B42	
01172526pb -	031656	A6	A5-A6	
01172526pc -	024444	A4	B44	
01172526pd -	030046	A0	A4+A6	
01172527pa -	024261	A2	B61	
01172527pb -	176702	V7	, A0, A2	
01172527pc -	024356	A3	B56	
01172527pd -	030035	A0	A3+A5	
01172530pa -	024147	A1	B47	
01172530pb -	176602	V6	, A0, A2	
01172530pc -	030016	A0	A1+A6	
01172530pd -	030115	A1	A1+A5	
01172531pa -	030445	A4	A4+A5	
01172531pb -	165567	V5	V6*RV7	
01172531pc -	025763	B63	A7	
01172531pd -	025741	B41	A7	
01172532pa -	024757	A7	B57	
01172532pb -	030775	A7	A7+A5	
01172532pc -	176402	V4	, A0, A2	
01172532pd -	024350	A3	B50	
01172533pa -	025464	B64	A4	
01172533pb -	030436	A4	A3+A6	
01172533pc -	030007	A0	0+A7	
01172533pd -	176302	V3	, A0, A2	
01172534pa -	024755	A7	B55	
01172534pb -	030775	A7	A7+A5	
01172534pc -	030335	A3	A3+A5	
01172534pd -	165734	V7	V3*RV4	
01172535pa -	171657	V6	V5+RV7	
01172535pb -	025165	B65	A1	
01172535pc -	024146	A1	B46	
01172535pd -	025277	B77	A2	
01172536pa -	012 101202173	A2	00240436pd	IPKARD: \$280+330pc
01172536pc -	1123 00000000	00000000, A2	A3	:
01172537pa -	024277	A2	B77	
01172537pb -	030316	A3	A1+A6	
01172537pc -	030115	A1	A1+A5	
01172537pd -	030004	A0	0+A4	
01172540pa -	176202	V2	, A0, A2	
01172540pb -	024454	A4	B54	
01172540pc -	030445	A4	A4+A5	
01172540pd -	025577	B77	A5	
01172541pa -	015 101202173	A5	00240436pd	IPKARD: \$280+330pc
01172541pc -	1156 00000001	00000001, A5	A6	: +1pa
01172542pa -	024577	A5	B77	
01172542pb -	030007	A0	0+A7	
01172542pc -	176102	V1	, A0, A2	
01172542pd -	024753	A7	B53	
01172543pa -	030775	A7	A7+A5	
01172543pb -	024652	A6	B52	
01172543pc -	030665	A6	A6+A5	
01172543pd -	025377	B77	A3	
01172544pa -	013 101202173	A3	00240436pd	IPKARD: \$280+330pc
01172544pc -	1134 00000002	00000002, A3	A4	AUTOTCOM: AUTOTTEK
01172545pa -	024377	A3	B77	
01172545pb -	165412	V4	V1*RV2	
01172545pc -	171764	V7	V6+RV4	
01172545pd -	010003	A0	0+A3	
01172546pa -	176502	V5	, A0, A2	

01172572pb -	011 101202173	S0 +A1	
01172572pd -	1016 00000001	A1 00240436pd	IPKARD:\$280+330pc
01172573pb -	030776	A6 00000001,A1	:+1pa
01172573pc -	071717	A7 A7+A6	
01172573pd -	1357 00000763	S7 +A7	
01172574pb -	1350 00000651	00000763,A5 S7	LASNEX:\$100+14pd } one of these
01172574pd -	006 004753627	00000651,A5 S0	LASNEX:\$90+2pb } sets the
		J 01172745pd	NHYDRO:\$472B

470B
ind of loop")

contents of "j"
to kalm+incz.

By saving and restoring the settings and registers at \$470A, we can construct specialized test cases, within the context of the larger code. If we might have misread the algorithm, and coded "ut(j+1)" instead of "ut(j-1)" and vice-versa. On longer loops, the section of loop that crashes is on one page, and it is not easy to see that the line defining "j+1+max" was forgotten on the previous page. This is where it would be very nice if the register table told me what register the induction variable is in case if the optimizer knows that it never needs to be stored in memory.

```
.rel $470a
.save tf01
.run to $470b
breakpoint reached at 01172574pd = NHYDRO:$470B task roottask
```

```
.dthyd
DTHYD = 1.000000e-05
```

```
.kalm
KALM = 202
```

```
.dediabtc,9
DEDIABTC(1) = 0 0 0 3.441064e-15 0 0 0 0 0 original result
```

```
.restore tf01
.set dthyd=1.e-6
```

```
.run
\005i
```

```
.run
\005pc2
```

```
[2.000]
\005?
```

```
8 INFO: mx=8 on pm from 1900 to 2300 10/06/92
8 INFO: pm machine 8 tue 10/06/92 1900-2300...
8 10/06/92 09:48:08 tr= 29.196 RUN +jfnk
```

```
*
jfnk suspended after cycle 118, t= 4.4507e-03
```

```
#\005i
.restore tf01
```

```
.rtr
01172525pb = NHYDRO:$470A
returns to 00566700pb = EHYDRO:$20B+6pc
returns to 00210362pd = EPHYSC:EPHYSC+51pd
returns to 00026021pc = ECYCLE:$60-1pa
returns to 00000633pd = LASNEX:$80-1pd
end of trace
```

```
.sub=nhydro;bkp $470b;dthyd
DTHYD = 1.000000e-05
```

```
.set dthyd=1.e-6
.run
```

```
breakpoint reached at 01172574pd = NHYDRO:$470B task roottask
```

```
.dthyd
DTHYD = 1.000000e-06
```

```
.dediabtc,9
DEDIABTC(1) = 0 0 0 3.441064e-15 0 0 0 0 0
```

```
.loc $470a
01172525pb = NHYDRO:$470A
```

```
.di mne 1172520b for 5
```

01172520pa =	030053	A0 A5+A3
01172520pb =	023330	A3 S3
01172520pc =	025055	B55 A0
01172520pd =	030056	A0 A5+A6
01172521pa =	030630	A6 A3+1
01172521pb =	025056	B56 A0
01172521pc =	025641	B41 A6
01172521pd =	023050	A0 S5
01172522pa =	030652	A6 A5+A2
01172522pb =	025042	B42 A0
01172522pc =	025657	B57 A6
01172522pd =	1070 00002747	A0 00002747, A7
01172523pb =	010754	A7 A5+A4
01172523pc =	023670	A6 S7
01172523pd =	025760	B60 A7
01172524pa =	025661	B61 A6
01172524pb =	025062	B62 A0
01172524pc =	041700	S7 000
01172524pd =	074600	S6 T00

this is not the original result

SAMPLE: 9999145pc = 1002

```
.set s6
s6 = 5.000000e-06
.setr s6 = 5.e-7
.run to $470b
breakpoint reached at 01172574pd = NHYTC:S470B task reottask
.dediabtc,kalm
DEDIABTC(1) = 0 0 0 3.441064e-16 0 0 0 0 0 0 0 0
DEDIABTC(13) to DEDIABTC(201) = 0
DEDIABTC(202) = 9.492560e-32
```

no. 3
1 2 3 4 5 6 7 8 9 10

PROBLEMS ENCOUNTERED

- CORRUPT PHYSICS

- many CPU hours into the run

(need "save"/"restore" (the bulk of our debugging does not occur with file i/o in between, ("print x" is NOT a solution.)

- debug two versions of code to track down corruption

(2 debug sessions \Rightarrow minimal ^{# of} windows)

- CORRUPT DATA

- we use binary chop to localize. E.g.

```
.bkr $47DA if j.gt.40  
.run
```

```
... things are ok  
.save t1 (save environment in file "t1")  
.bkr $47DA if j.gt.100  
.run
```

```
... things crash  
.restore t1 (restore environment t1, not t0 or t2)  
.bkr $47DA if j.gt.60  
...
```

- need to display various formats (ascii / hex / dec / mnemonic)

- need to display large arrays (many elements \Rightarrow not one window / element)

CAVEATS

• Conditional breakpoints should be "fast", via coding inserted into the executable, not via interrupts that cause swapping

• Some uses of the debugger require a minimal ~~two~~ number (i.e. one) of windows

• Some buttons should be "user definable"

• to the macro invocation's to them

- aliases ("dh" = "display hex" should be one allowable subset of macros)

A User Perspective of Debugging on Supercomputers

Jeffery A. Kuehn

National Center for Atmospheric Research
Scientific Computing Division

ABSTRACT

Typically, when a user brings a problem to be debugged to the NCAR/SCD consulting office, the code is peppered with print statements. The print statements do not provide output data from the simulation, but rather, they record intermediate values of solution variables and information about entry and exit of subroutines. In other words, print statements are chosen as the preferred debugging method instead of debuggers which would allow the user to stop and start the code, print variables and investigate the stack as necessary. When asked, the users are quite clear about their reasons for not using the debuggers available to them. Their reasons involve the complexity of the debugger's interface and a mismatch between the user's runtime environment and the debugger's support environment. This paper summarizes these issues and suggests several possible avenues for addressing them.

1. INTRODUCTION

The user community at NCAR has a great deal of experience in supercomputing, on machines from many different vendors at university, corporate and research sites all over the world. Many of these users have participated in evaluations of many machines often very early in their production cycles. Several hundred problems pass through the NCAR/SCD consulting office every month, but only rarely does someone try to use a debugger—estimates based on accounting statistics suggest that less than one percent of our programmers use the debugger on our Cray Y-MPs. The typical approach is to insert print statements throughout the code to record the entry and exit of subroutines and print values of critical variables at strategic locations. Since both tracing subroutine entry/exit and examination of critical variables can be accomplished with a debugger, the author began interviewing users and other consultants. The

interviews revealed several reasons why users prefer debugging with print statements over the use of a debugger. In many cases, these "user complaints" also suggest areas for improvement of current debuggers.

2. DEBUGGER AVOIDANCE TACTICS

2.1 Print Statements

Users' preference for the print statement technique of debugging stems from five basic issues. (1) The mechanics behind the approach are basically intuitive; any effort expended is spent towards interpreting the data and finding the bug. (2) The approach has no learning curve associated with it since it can be done in the language of the original code with which the user is (presumably) familiar. (3) The approach works on all machine architectures and operating systems since user I/O depends only on programming language standards. (4) It works

interactively, in batch queues, in any job class, and under any time and memory constraints under which the original code ran. (5) It even produces results in cases where memory overwrites of static data or stack space cause the program to crash far from the point at which the error occurred; in short, the method is fairly robust.

2.2 Other Tools

Print statements are not the only debugging method users apply to their codes to avoid the debugger. FORTRAN-lint (flint) marketed by IPT allows a user to analyze a code in great detail, performing syntax checks, conformance to standards, argument consistency, etc. This, however, is not a substitute for a debugger, since flint usually produces a huge volume of informational messages and warnings, most (if not all) of which have nothing to do with the problem.

3. MAKING DEBUGGERS MORE USEFUL

User complaints about current debuggers fall into two categories. First, there are complaints about the user interface being difficult, inconsistent, primitive, or incomplete in one way or another. The second category deals with how debugger fits (or rather doesn't fit) into the user's computing environment or paradigm. Most of these complaints are relatively straightforward to address.

3.1 User Interface Issues

3.1.1 Intuitive Interface: The strongest comment made in the user interviews—and in fact the only comment unanimously echoed by all of the users—involved the long learning curve for a user approaching the debugger for the first time. The user interfaces *must* be designed to be intuitive even to the novice user of a debugger. It is obvious that no user will ever use a debugger unless they have encountered a problem in their code for which the solution is not apparent. Often by the time someone resorts to using a debugger, they are already frustrated and probably cursing every piece of silicon in sight. It is reasonable to assume that this state of mind is not conducive to learning a new software package, especially one as

complex as current debuggers. Also, it should be noted that since users apply the debugger so infrequently, every time they go to use one, they essentially must relearn it from scratch. One user even went so far as to say that his concept of a good debugger would be one for which he didn't need to read a man page. It is incorrect to assume that users will have access to printed documentation—vendors supply a limited number of copies of free documentation which ends up in offices and libraries far from the users' desks. Suggestions on how to address these issues include:

- Provide a user interface that the user will find intuitive enough so a manual or help function will not be needed.
- Take full advantage of the graphics capabilities of a windowing system (X-Windows with Motif or OpenLook at a minimum) to improve the appearance and intuitiveness of the interface.
- Provide a detailed online help facility for additional user support even though the user interface is so highly intuitive that use and function are obvious to the user.
- Add a "printpoint" capability to the debugger that functions like a breakpoint except that when a printpoint is reached, a list of user-specified variables is printed. Allow users to specify the format of this output.
- Along these same lines, add a "watchpoint" capability to the debugger which simply notes when execution passes through a particular line of code. The watchpoint facility should also allow the user to track entry and exit for subroutines—a list of the routines in the current sequence of calls displayed on the screen would be useful.
- Add conditional clauses to breakpoint and printpoint statements that allow the user to stop the code or print the data only when specified conditions are true. The syntax of any common programming language (such as C or FORTRAN) should be acceptable.
- Make breakpoints, printpoints, and watchpoints installable by pointing and clicking at

a menu and a source code window.

- Make debugger and system error messages printed by the debugger clear and simple. "Operand Range Error" means nothing to most users. Perhaps allow a user to point-and-click an error message, then click an "explain" button.

3.1.2 Graphics as an Aid: Current debuggers presume that the user is familiar with the code they are trying to debug. This assumption is often incorrect. More frequently, users either inherit or borrow a program from someone else, then they try to modify it for their purpose. A graphic display could be of great assistance here:

- Represent the calling tree with a flowchart graphic.
- Display the stack traceback path by highlighting the appropriate nodes in the flowchart graphic of the calling tree. Group system library routines for convenience.
- Allow point-and-click access to subroutine source code via the calling tree graphic.
- Represent the data structures and memory map graphically.
- Allow the user to highlight routines in the calling tree, and have the debugger highlight the corresponding static data areas in the memory map. This allows users to see what is adjacent to overwritten structures.
- Allow direct point-and-click access to global data structures, and allow point-and-click access to local data via highlighting the calling tree and pressing a button to see local data structures expanded.
- All of this should be implemented under X-Windows.

3.1.3 Data Flow Analysis: It is very difficult to examine large arrays by printing individual elements or looking at long lists of numbers. Printing individual array elements worked fine on computers of 10 and 20 years ago, but the forte of supercomputers is to manipulate massive quantities of data. Could it be that the *emphasis* of debugging needs to be shifted from logic to data? The advent of parallel

processing—which introduces a non-deterministic ordering into the program logic—may further underscore this shift as we become more experienced. Combine these with computer graphics that allow the representation of huge quantities of data in a format which can be examined very quickly:

- Provide the ability to graphically display large arrays as line graphs, contour plots, scatter plots, and histograms.

3.1.4 Languages and Symbols: Computer simulations are becoming more sophisticated in taking advantage of the differences and strengths of various languages, including FORTRAN, C, and Assembler. The user should be allowed to debug their code in the language(s) in which it was written. The references to symbols in debuggers need to be more consistent: one debugger prints symbols with one syntax on output but requires a different syntax for symbol input. Finally, all of the language's constructs should be available for referencing variables. Therefore:

- Provide the ability to print entire data structures as well as sections of a data structure in high-level language syntax.
- Provide support for all languages and inter-language calls.
- Keep symbol format consistent throughout. When a symbol is written to the screen, it should be exactly the symbol a user would type to view information on the symbol.

3.1.5 Complete Environment: A debugging session frequently runs as cycles of editing, compiling, running code, then running debugger; often the last two are combined. Situations arise where more information is needed about what is happening within a particular subroutine, so users would like to "turn-off optimization"—on that subroutine—this requires recompiling and linking of the routine in question. Within an iterative process such as this, entering and exiting a series of utilities feels like extra work and seems awkward. Therefore:

- Add compiler, loader, and editor interfaces within the debugger when changes are made

to the source code, then when an "update" button is pressed, compile and re-link the code. Preferably, recompile only those pieces of the code that require updating.

- Allow users to recompile and reload pieces of a code with optimization turned off. This should be implemented as an "unoptimize" button which, in reality, recompiles without optimization and re-links the code. Preferably, recompile only those pieces of the code that require updating.
- It may be useful to incorporate syntax-checkers and code analyzers such as lint, IPT's FORTRAN-lint product or similar tools. However, it should be noted that while these tools are useful, their application is more limited than that of a compiler, loader, or editor.

3.2 User Environment Issues

Software designers and computer support staff usually do not have constraints placed on their use of a computer; *this is not the case for users*. User codes typically run in a batch environment with CPU time limits and memory limits. Because a batch queuing system allows more control over the job mix, the time and memory limits for batch processing are typically more relaxed than those for interactive work. Additionally, users are typically charged for their use of the machine, based on algorithms containing CPU time and memory residency factors among others. Because some models and simulations have large memory requirements, users must often multitask a large code. Finally, because the users write their jobs to run in batch, the tools they use are not the same as those often used for interactive work.

3.2.1 Time: Frequently, a code will run for several hours in a batch system before crashing; in the cases of codes that do crash early in the job, there can still be a significant block of time for startup. Because of situations like this, interactive debugging is not always practical, especially if the code must be run without optimization. One user explained that he could turn around five runs with FORTRAN WRITE statements inserted into strategic places in the time it took an unoptimized version of his code

to crash once. Thus it is critical to:

- Provide source level debugging with fully optimized code.
- Report as much information as possible from a core dump without the need for the user to repeat the compile/run with debugging turned on.

3.2.2 Memory: For machines that swap memory instead of paging it, large memory processes can typically be run only through a batch system. Thus if the code is too big, the user must debug with print statements unless the code is to be debugged on a near-dedicated machine. Perhaps:

- Offer a memory segmenting/paging mechanism to keep large data arrays on secondary storage. Admittedly, this is the most difficult suggestion to implement.

3.2.3 Multitasking: Debugging multitasked jobs has been a sore spot among users for some time now. Debuggers seem to rarely support multitasked code, and when they do, it is often not possible to run the debugger with live multitasked codes. This is not the case for all vendors, and those who do support multitasking should be applauded for their efforts. Those who are not currently providing multitasking support must:

- Provide support for debugging live multitasked jobs.
- Attempt to provide a representation of a multitasked job which sequential-thinking users will find intuitive.
- Provide visualization tools within the debugger to assist the user in monitoring multitasked jobs.

3.2.4 Tools: One vendor's debugger requires a code to be split up, one subroutine or function per file, before source-level debugging is possible. At this point, a tool such as the UNIXTM make utility is needed to keep track of all the pieces and put them back together in an efficient manner. Worse yet, for the slightly more sophisticated user who had created meaningful groupings of subroutines and functions within files and placed these groupings under

control of make, the makefile had to be rewritten from scratch. In another case, a vendor's debugger failed to work with one of the vendor's compiler extensions that allowed file inclusion, making debugging a total nightmare.

- Ensure that the debugger supports *all* compiler features.
- Keep the debugger's functionality independent of other software development tools: i.e. the debugger should not *require* the user to use anything except the compiler and the loader, all other tools should be optional. The debugger should be able to work with a single source file or a code that has been split up for use with utilities such as make/nmake, SCCS or RCS, etc.

4. SUMMARY

The title of this paper could very well have been "*Interface! Interface! Interface!*" since that is the key to the complaints users are making about debuggers. The current interfaces are fine for someone who is familiar with the details of computer architecture, compiler internals, and maybe an assembler language, but they miss the mark for users who know only the high-level language in which they program. When users need to use a debugger, they already have one problem; this does not need to be compounded by an intractable debugging tool. Moreover, since the user is not likely to be the author of the code (more likely the user is a consultant or someone who borrowed or inherited the code) the debugger should not require the user to be familiar with the code, but rather the debugger should lead them through the code. Debugger use can be greatly simplified by offering:

- Graphics as an aid to visualizing code structure.
- Graphics as an aid to visualizing data structures.
- Graphics as an aid to visualizing flow of execution (stack tracebacks).
- Graphics as an aid to visualizing relations between static data regions and subroutine code.

- Point-and-click installation of tracing features such as breakpoints, printpoints, and watchpoints.
- Point-and-click access to the source code.
- Point-and-click access to the local and global data structures.
- Consistent use of symbols.
- Native language syntax.
- Editor, compiler, and loader interfaces within the debugger.

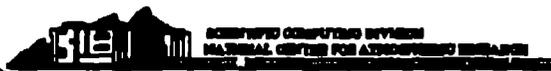
Lastly, the typical user environment of time limits, memory limits, multitasking requirements, and software development utilities must be kept in mind when designing the debugger

In closing, there is one final user comment to be heard: "*The battle for better debuggers will not be over until you see users defending their favorite debugger with the same fervor as they defend their favorite machine, their favorite editor, or their favorite GUI.*"

A USER PERSPECTIVE OF DEBUGGING ON SUPERCOMPUTERS

Jeffery A. Kuehn

**National Center for Atmospheric Research
Scientific Computing Division**

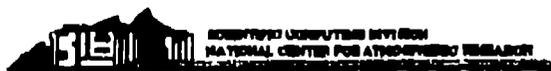


USER DEBUGGING TECHNIQUES

WRITE (*, *) 'ENTERING SUB1'

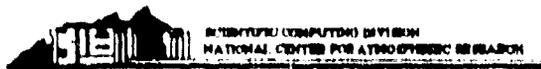
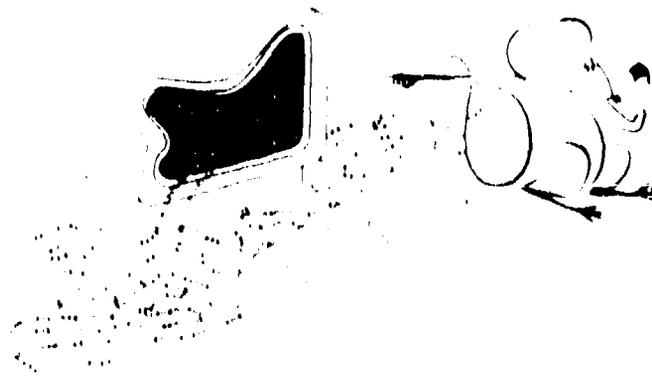
WRITE (*, *) A, B, (X (I), I = 1, N)

WRITE (*, *) 'EXITING SUB1'



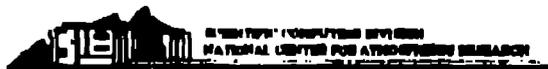
OTHER TOOLS USED

- lint
- flint (IPT)
- compiler
- profiler



USER COMPLAINTS

- **Difficult**
- **Inconsistent**
- **Primitive**
- **Incomplete**
- **Doesn't Fit**



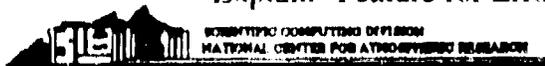
INTERFACE PROBLEMS

- **Learning Curve**
- **User Frustrated**
- **Long Intervals Between Uses**
- **Complexity of Debugger**
- **Unfamiliar With Code**



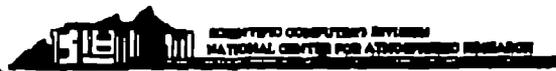
INTERFACE SUGGESTIONS

- More Intuitive
- X Window Graphics Interface
- Detailed Online Help
- "Print Point"
- "Watch Point"
- Conditionals on
 - Break Point
 - Print Point
 - Watch Point
- Install With Click on Menu & Source Window
- "Explain" Feature for Errors



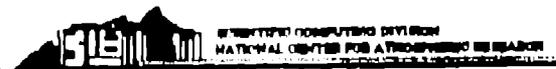
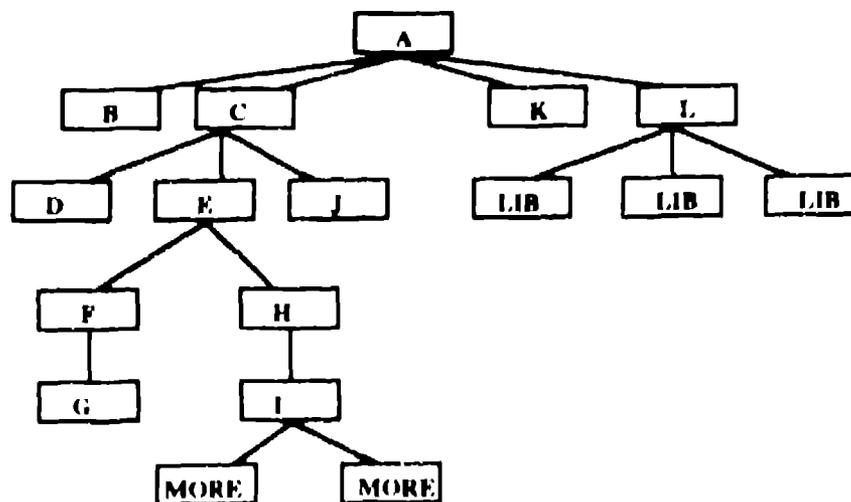
GRAPHICS/APPEARANCE PROBLEMS

- User Often Not Familiar With Code
- Many Debuggers Use DBX Interface With Buttons For Typing Short Cuts



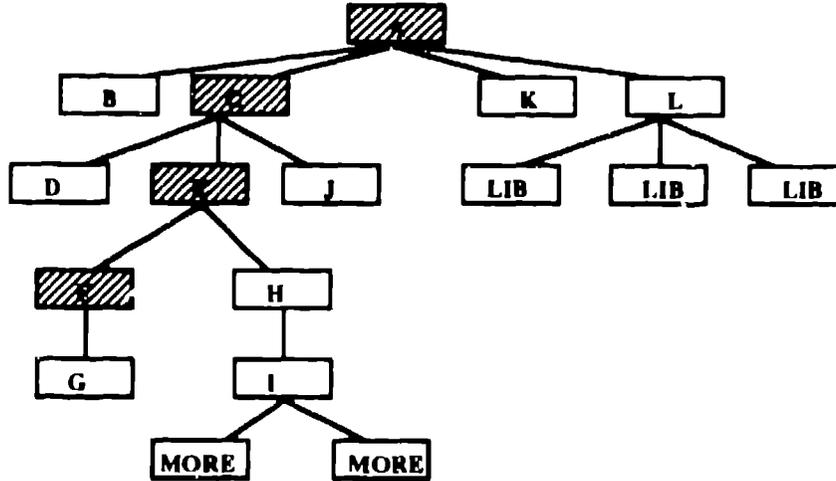
GRAPHICS SUGGESTIONS

- Represent Calling Tree With A Flow Chart Graphic



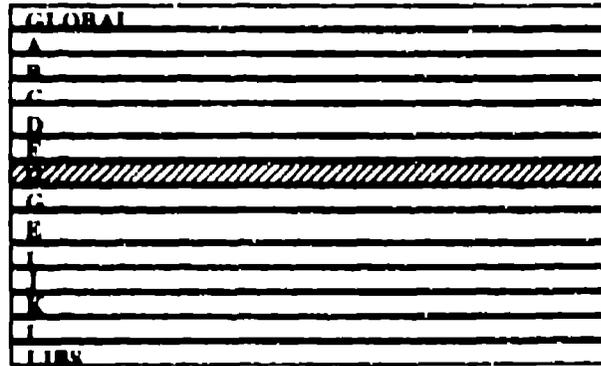
GRAPHICS SUGGESTIONS

- Show Traceback On Calling Tree



GRAPHICS SUGGESTIONS

- Memory Map Graphic



GRAPHICS SUGGESTIONS

- **Allow Source Access Via Point Click on Call Tree**
- **Allow Point Click Access To Data Structures**



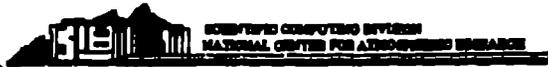
VISUALIZING DATA

- **Line Graphs**
- **Multi-Line Graphs**
- **Contour Plots**
- **Scatter Plots**
- **Histograms**



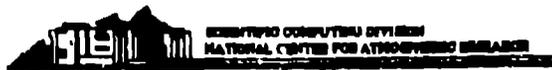
LANGUAGES AND SYMBOLS

- **Full Support For All Languages/Compilers**
- **Ability To Examine Data Structure With High Level Language Syntax**
- **Consistency of Symbols**



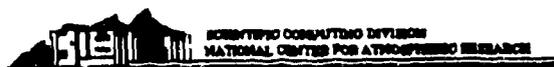
COMPLETE ENVIRONMENTS

- **Source Editor (vi, emacs, other)**
- **Compiler & Loader**
- **"Unoptimize" Button**
- **Syntax & Type Checkers**



USER ENVIRONMENT

- **Time Limits**
 - **No Optimization Runs Too Slow**
 - **Codes Run Long**
- **Suggestions**
 - **Source Level Debugging Must Be Provided For Fully Optimized Codes**
 - **As Much Information As Possible Must Be Extracted From A Core Dump Without Recompiling and Re-running**



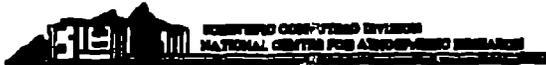
USER ENVIRONMENT

- **Memory Limits**
 - **Jobs With Large Memory Requirements Cannot Be Run Interactively On Most Machines**
- **Suggestion**
 - **Develop A Segmenting/Paging Mechanism To Reduce The Physical Memory Requirement**



User Environment

- **Multitasking**
 - **Often Required**
 - **Time Restrictions**
 - **Memory Restrictions**
- **Suggestions**
 - **Provide Support For Debugging Live Multitasked Jobs**
 - **Attempt to Represent A Multitasked Job In A Manner That Sequential - Thinking Users Will Find Intuitive**
 - **Visualization Tools Within Debugger**
 - **Mark Active Routines**
 - **Display Tasks Attaching To Processors**



TOOLS

- **One Debugger Required Use of "Make" With A Specific Structure**
- **One Debugger Didn't Support The Compiler's File Include Feature**
- **Suggestions**
 - **Debugger Should Work With Tools, but Not Require Their Use**
 - **Debugger Should Support All Compiler Features**



SUMMARY

- **Interface! Interface! Interface!**
- **"The Battle For Better Debuggers Will Not Be Over Until You See Users Defending Their Favorite Debugger With The Same Fervor As They Defend Their Favorite Machine, Their Favorite Editor, or Their Favorite GUL."**



bdb: Vendor Update for 1992

Benjamin Young, Cray Computer Corporation

Abstract

bdb is a new source level debugger being developed by Cray Computer Corporation. Work has been underway since May, 1990 and it was first released to a customer in October of 1991.

To accomplish our design goals and to simplify implementation, we chose a library approach to the debugger design. We split debugger functionality into several different areas (many of which were common areas for other tools). For each area we designated a new library to be written or used existing libraries from other sources where possible.

The end result of this design technique is a very modular debugger which has been or can be extended to multi-tasking debugging, distributed debugging, process monitoring, symbol table debugging, dump debugging, and many other useful tools.

This update to information presented at SD'91 will first review standard capabilities of **bdb** and will then highlight new functionality that has been or is currently being added to **bdb**.

Standard bdb capabilities

bdb is a source level C and Fortran debugger designed and developed by Cray Computer Corporation. As with any debugger, **bdb** comes with its own set of standard features. These include:

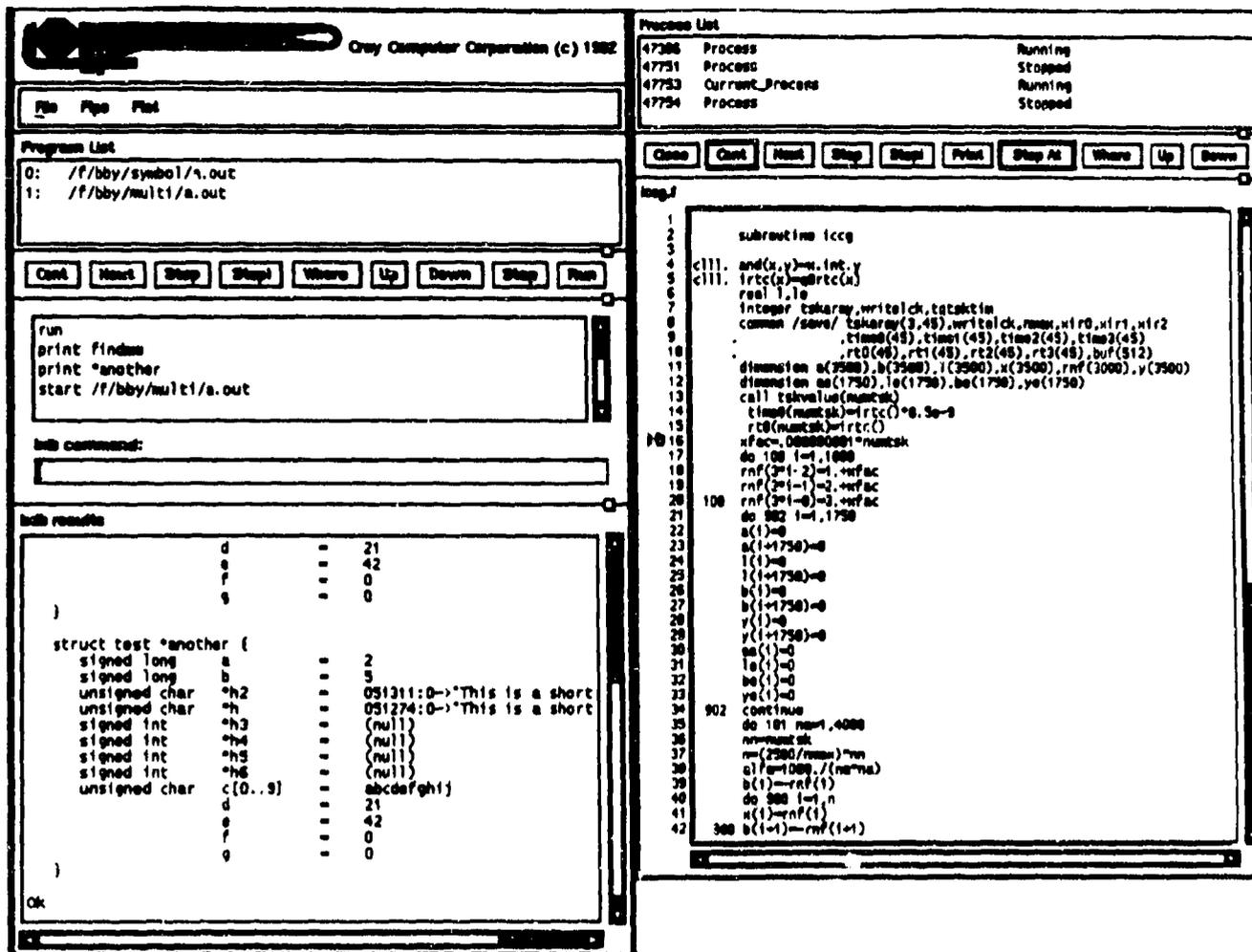
- Ability to debug multiple independent processes.
- Ability to debug live multi-threaded (multi/macro tasked) processes.
- Ability to attach to and debug pre-existing processes.
- Full symbolic capability for Fortran and C.
- Conditional breakpoint capability.
- Standard high level debugging capabilities (e.g.: source line stepping, symbolic access and display, call command).
- Good low level debugging capabilities (e.g.: extensive dump support, full register access, single machine instruction stepping).
- Multiple user interfaces available including line mode, Athena Widgets window mode and OSF/Motif window mode.
- Ability to debug core files.
- Full on-line help facilities.

Much of the emphasis to date in **bdb** has been in the area of process control and symbolics. It was decided very early on in the **bdb** design process that flexible process control was key to multi-threaded and distributed debugging. Good symbolic handling was also recognized as tremendously important to any debugger design. We felt that most user frustration with supercomputer debuggers could be traced to either the inability to debug certain types of processes or the inability to fully support the symbolics of the programming language being debugged.

This emphasis is reflected in the work performed during this past year on **bdb**. New features currently being added include:

- Ability to debug piped processes (a first step into distributed debugging).
- Ability to debug simulated processes (including the operating system).
- Initial implementation of C native language expression evaluation.
- Full signal control including the ability to have **bdb** "register" signals to be ignored or to "register" a signal handler for the process being debugged.
- Integration of **stb** (symbol table browser) into the windowed versions of **bdb**.
- Integration of **datamash** (a generic "data to symbolic" overlay utility) into all versions of **bdb**.

Figure 1 bdb OSF/MOTIF Windows - Debugging a multi-tasked Fortran Program



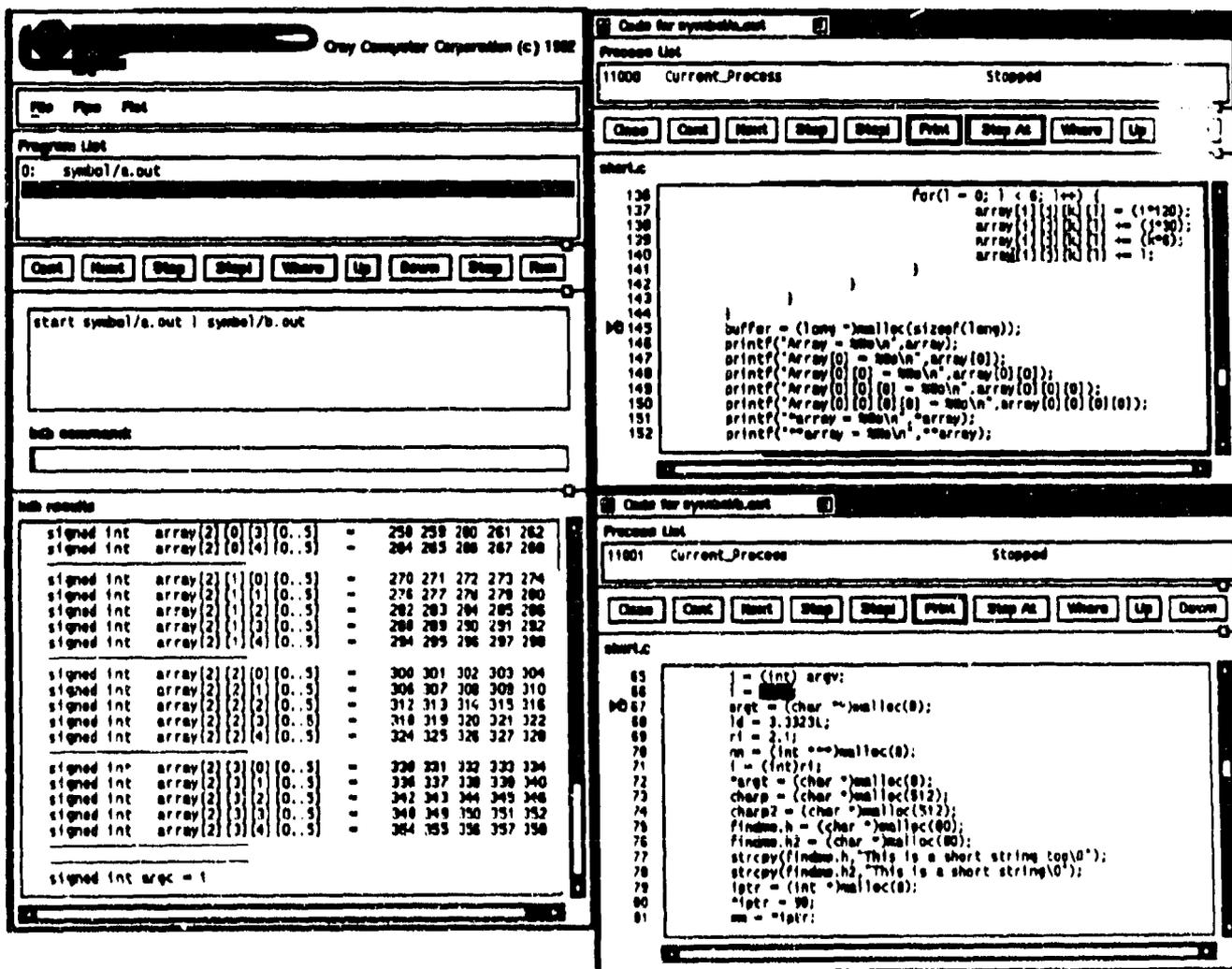
All of the aforementioned new features exist in the current version of bdb with the exceptions of stb and datamash. Both stb and datamash have been created as stand alone products (as proof of concept) and work is currently in progress to integrate them into the debugger.

The rest of this paper describes each of the new features in greater detail as well as outlining future directions for bdb in the coming year.

Debugging Piped Processes

We decided that the first step to take in distributed debugging was to be able to debug distributed processes that exist on the same machine. This led us to the notion that the debugger should be able to recognize a command line that included piping from one process to another.

Figure 2 Debugging piped processes in window mode



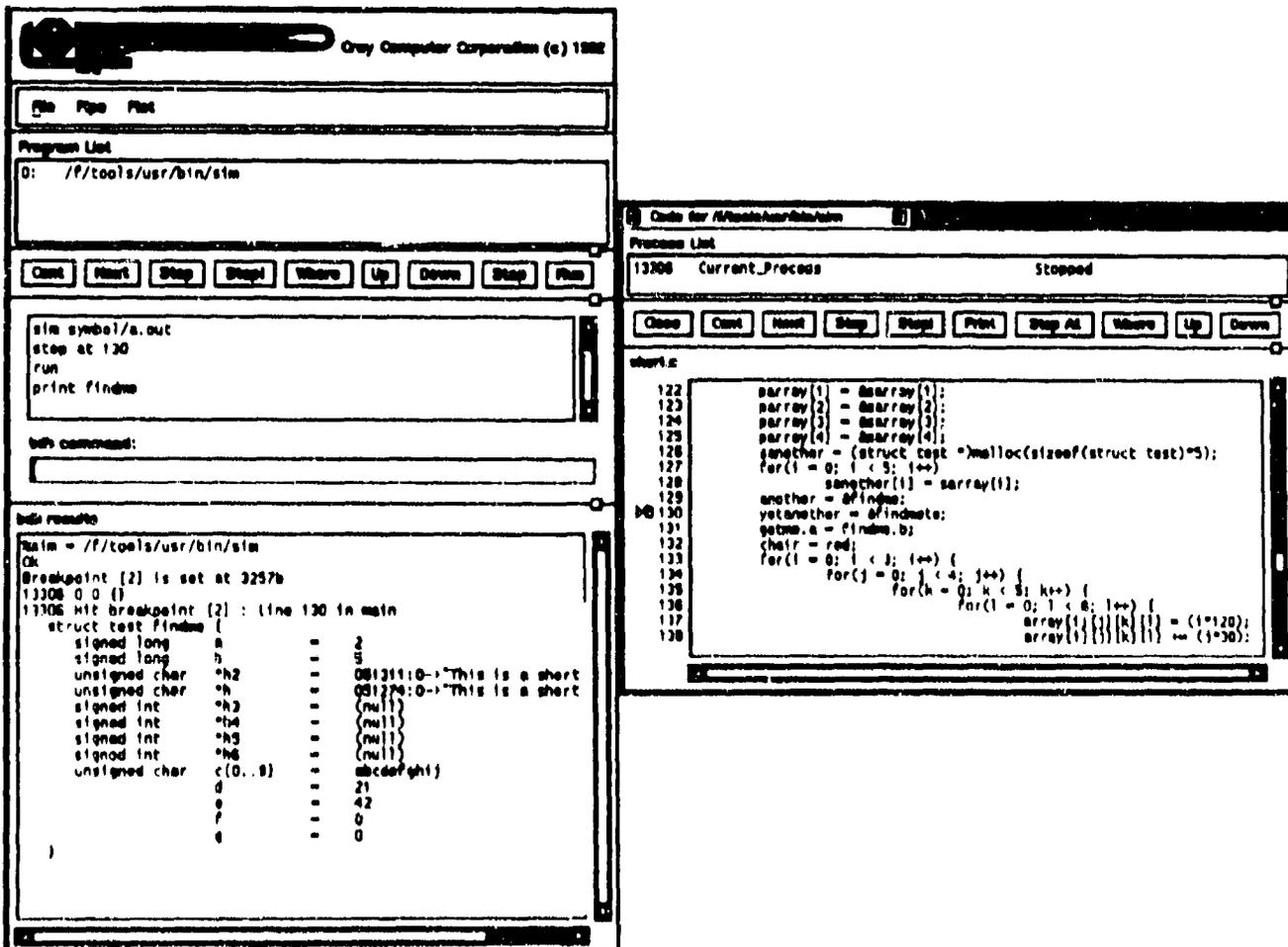
When a user starts a set of piped processes, bdb creates all pipes and processes needed and then gives the user control. In window mode, a separate code window is provided for each process.

Debugging simulated processes

Our instruction level simulators (sim and sim3) are our main debugging tools for the operating system. sim and sim3 have the ability to simulate the CRAY-2 and CRAY-3 hardware, respectively, but lack the symbolic support found in bdb. bdb has the symbolic support but lacks the ability to simulate hardware. With a few minor changes to our process control library and to our simulators, bdb is now able to debug simulated processes at a source level.

The advantages of being able to debug simulated processes include being able to debug the operating system at source level and the ability to debug CRAY-3 processes on a CRAY-2.

Figure 3 Debugging simulated processes in window mode



To the user, debugging a simulated process looks just like debugging a real process except the process runs a bit slower. All debugger commands (with a few notable exceptions) that can be used with a real process are available for use with simulated processes. In fact, in most cases, the debugger doesn't know the difference between a simulated or real process.

The modular design of **bdb** made adding this feature rather simple. Since all process references made by the debugger (including all memory and register references) are made through a library, the key changes needed were to the library itself, not to **bdb**. By adding the ability to reference (read and write) a simulated process memory and register set to the library, 95% of **bdb**'s capabilities were then available for use in debugging simulated processes. The main change to **bdb** itself was the addition of a single new command (called **sim**) that caused the debugger to start a user specified simulator instead of the binary noted on the **bdb sim** command line.

About the only commands that a user can't use with a simulated process are the new **bdb** signal control commands. This is due to a deficiency in the simulators used (which at this point in time do not support simulating signals for user process simulation).

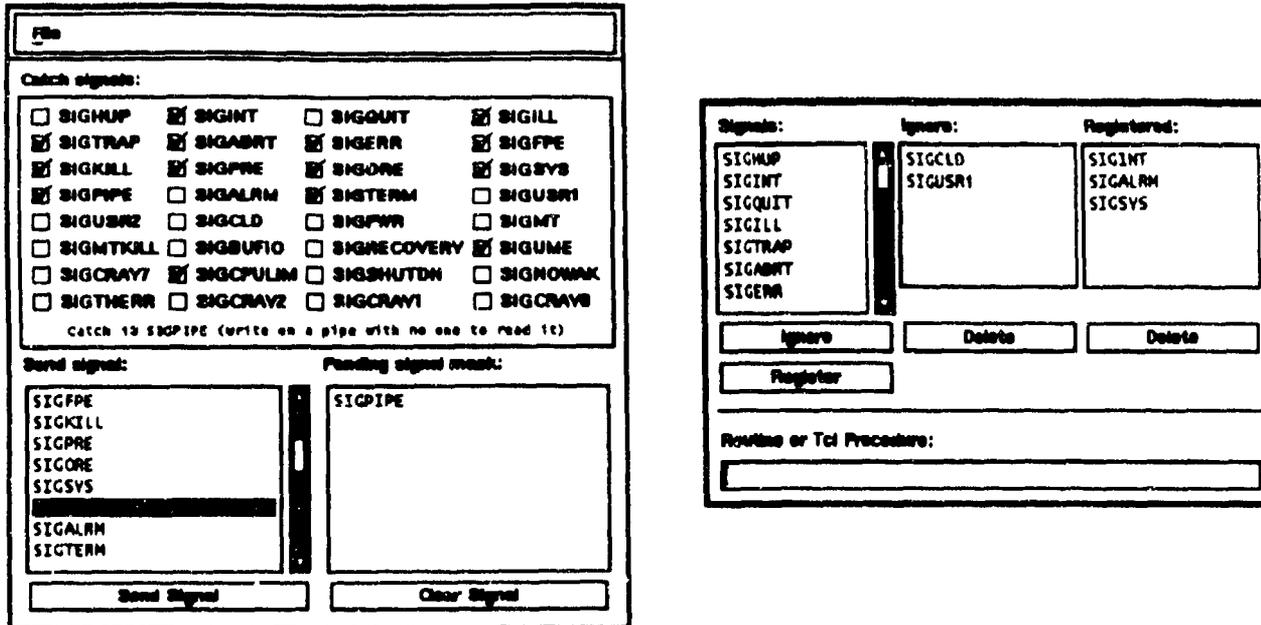
C native language expression evaluation

Our first attempt at native language expression evaluation was added to **bdb**. Currently we limited it to the C language and will add Fortran in the near future. Native language expression evaluation allows users to print the value of arbitrary expressions in the programming language being debugged. This is especially useful in window mode where users can mouse off a line of C code and hit the print button to have the value of a line displayed.

Signal support

Until just recently, a deficiency in **bdb** support was in the area of signals. With the new signal support in **bdb** users are able to selectively catch or uncatch signals, send signals, clear pending signals, and register signal handlers for the debugged process.

Figure 4 Signal control windows



Two new windows were created to make signal control easier for the end user. The first window (on the left in Figure 4) controls which signals will be caught by the debugger and also provides the user an easy way to select and send a signal to a process. If any signals are pending for the process, they appear in the pending list.

To catch a signal, the user clicks on the box next to the desired signal in the Catch signals window and a check mark appears in the box. To "uncatch" a signal the user again clicks on the box and the check mark is cleared to indicate the signal is no longer being caught. To send a signal to a process, the user selects the signal to send from the Send signal list and then hits the Send Signal button. To clear a pending signal, the user selects a signal from the pending list and then hits the Clear Signal button.

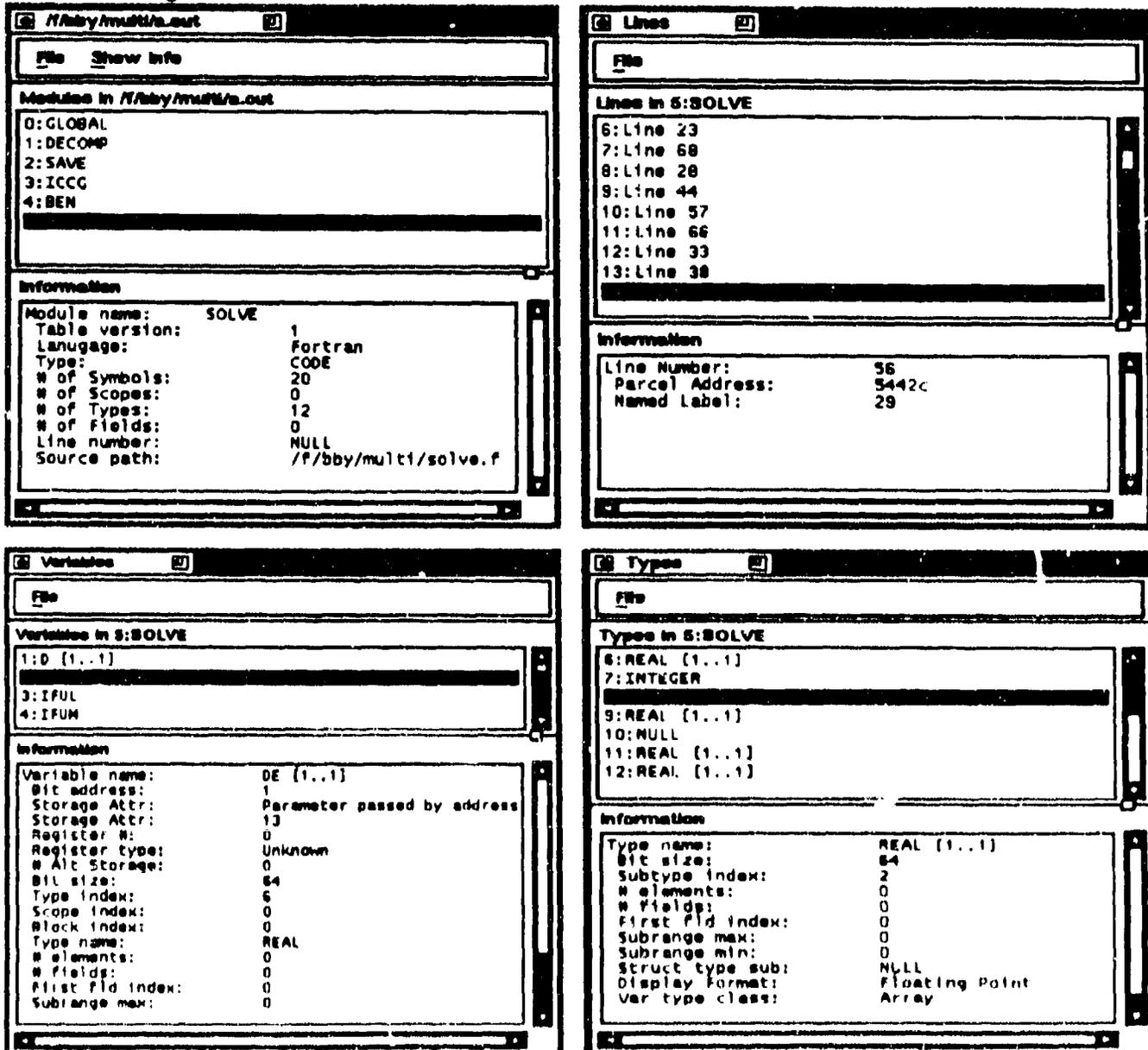
When the user selects any signal for either the Catch signals window, the Send signals list, or the Pending signal mask list, a help message appears in the window which gives a brief explanation of the signal selected.

The second window (on the right in Figure 4) is the signal register window through which users can register signals to be ignored by their process or can register a signal handler for their process.

Symbol Table Browser

The symbol table browser (stb) is a stand alone utility that is currently being integrated into bdb. The purpose of stb is to allow users to easily browse through symbol tables getting information about program modules, program variables, type definitions, source code line positions, and scope definitions.

Figure 5 stb windows



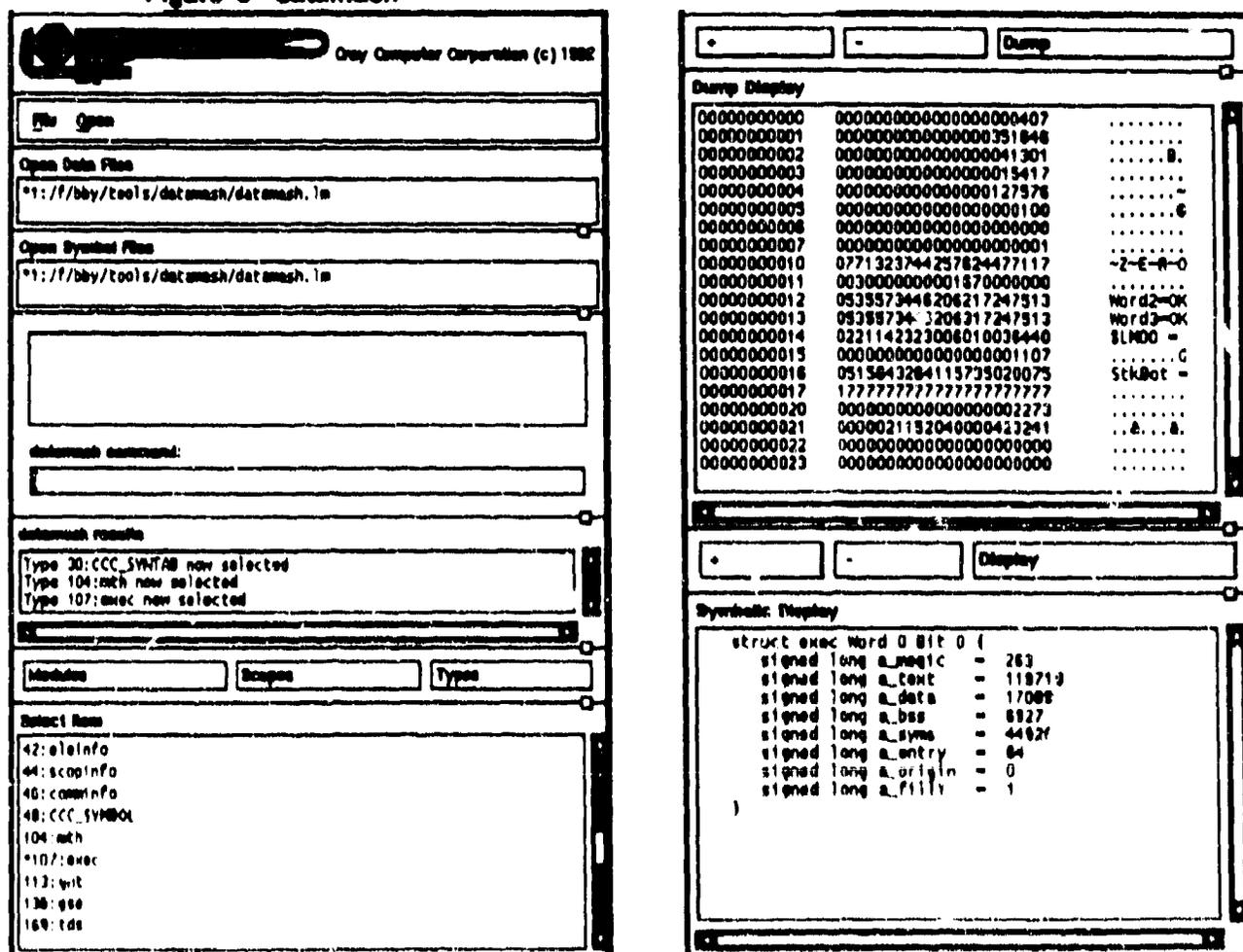
By integrating stb into bdb, the user not only gets the symbol table browsing capabilities but also gets a more intuitive way of selecting program variables or

type definitions for use in other areas of **bdb** (such as variable display or type overlays of data).

datamash

datamash (like **stb**) is a stand alone utility that is currently being integrated along with **stb** into **bdb**. **datamash** provides the user with the ability to easily overlay any type definition (from any symbol table file) over any set of data. Currently, data is found in files, although, once **datamash** is fully integrated into **bdb**, data will come from both files and processes.

Figure 6 datamash



datamash was initially created to help in system dump debugging. Its usefulness has since been discovered in all kinds of file and data debugging.

The integration of **datamash** will be the first step in breaking the common link in debuggers between a process and its associated symbol tables. With **datamash**, any symbol table can be overlaid on any process or file data. We believe the benefits of this split will become more apparent in the area of distributed debugging where users may need to use the symbolics of one process in a distributed group to help view the data of another process in the same group.

The merger of **bdb** and **datamash** will provide more benefits than just a superset of **bdb** and **datamash** commands. As mentioned earlier, **datamash** will now have access to process data as well as file data. Also, by taking advantage of **bdb**'s callback loop, **datamash** will be able to provide a real time data and symbolic display of a process while it runs.

Integration of **datamash**, **stb**, and **bdb** provides a single main challenge, that being in the user interface area of **bdb**. One of the things we try to avoid in **bdb** (and in all of our tools) is an explosion of windows. At the present time we are re-examining the current **bdb** interface to see how we can better present to the user all the different capabilities **bdb** will have to offer without overwhelming the user with a large set of default windows that the user must display.

Future Work

Work for the short term in **bdb** will continue in the areas of user interface, graphical data visualization, process simulation, watchpoints, and native language expression evaluation.

The user interface may go through a major overhaul with the merging of **stb** and **datamash** into **bdb**. This should be a relatively painless exercise given the split between **bdb** and its user interface. The **bdb** user interface is written entirely in Tcl (Tool command language developed by John Ousterhout from the University of California at Berkeley) making the interface changes simply a matter of rewriting the Tcl code. None of the core code of **bdb** will be affected by this change.

Also being added in the near future to the list of **bdb** interface options is an OPEN LOOK window mode.

Finishing touches are currently being put onto a set of routines that will provide **bdb** with its first dive into the world of graphical data visualization.

In the area of process simulation, we have had user requests for the ability to switch a process from real to simulated and back again. We are currently

investigating this possible functionality. We are also looking into the ability to debug a **vm** (virtual machine) process. **vm** processes are very similar in nature to simulated processes but run at machine speed. The main disadvantage of a **vm** process over a simulated process is the lack of any debugging capability in **vm**. By adding the ability in **bdb** to debug **vm** processes, one would gain this debugging capability.

Watchpoints are another deficiency in **bdb**. Currently **bdb** does not support the idea of a watchpoint. Our main concern with watchpoints is the performance overhead one pays to have them. We are currently looking into the fast watchpoint scheme used in **ldb** from Los Alamos.

Native language expression evaluation is another area in **bdb** that needs more work. Fortran expression evaluation will be added and the C expression evaluation will be extended to cover the ability to use functions or subroutines in expressions.

Acknowledgments

The author wishes to acknowledge the help of a number of individuals at Cray Computer Corporation without whom **bdb** would still be an idea on my white board. These include Scott Bolte, Randy Murrish, and Tom Engel.

A special thanks also goes to John Ousterhout from the University of California at Berkeley, the developer of Tcl, upon which the low level **bdb** interface is based.

References

Benjamin Young, "bdb: A library approach to writing a new debugger," Proceedings of the Supercomputer Debugging Workshop '91, Albuquerque, New Mexico, November 14-16, 1991.

John Ousterhout, "Tcl: An Embeddable Command Language," Proceedings of the Winter 1990 USENIX Conference, Washington, D.C., January 22-26, 1990. (1990A)

Information about Tool Command Language, along with the latest source code, may be obtained from John Ousterhout, University of California at Berkeley. A mailing list exists which is devoted to Tcl questions. To join, mail a request to tcl-request@sprite.berkeley.edu and ask to be included on the distribution.

Copyrights

stb and **bdb** are trademarks of Cray Computer Corporation.

OSF/Motif is a trademark of the Open Software Foundation, Inc.

OPEN LOOK is a registered trademark of USL in the United States and other countries.

Tool Command Language (Tcl) was developed by Prof. John Ousterhout of the University of California at Berkeley.

Author Information

The author can be contacted by mail at Cray Computer Corporation, 1110 Bayfield Drive, Colorado Springs, CO, 80906 or by e-mail at bby@craycos.com.

Thinking Machines Vendor Update

Rich Title

October 1992

.

Recap from last year's talk

- Prism 1.0 was about to be released
- Debugging and performance analysis for our data-parallel languages (FORTRAN 90, C*)
- *dbx*-like capabilities with a graphical interface
- OSF/Motif based (slide)
- FORTRAN 90 expression interpretation
- Data visualization capabilities for arrays
- Performance histograms at the source-line level

Performance Menu

Choose: To: Performance Histograms Display p

Cancel

File Options

Program: arrays.x

Source File: intarray.fcm

```

1  Miscellaneous operations on integer arrays
2
3  program intarray
4
5
6
7  integer j
8  integer, array(32,32):: a
9  integer, array(32,32):: b
10 integer a
11
12
13 call _start
14 a = 0
15 do 10 j = 2,32
16   a(j,:) = a(j-1,:)+32
17 continue
18

```

Resource

FE cpu (user) 39.7 X

FE cpu (system) 7.0 X

FE wait 37.4 X

OH cpu (user) 36.7 X

OH cpu (system)

Comm (Send/Get)

Comm (MEMS) 38.2 X

Comm (Reductions) 0.0 X

Cancel Cancel All Help

Resource OH cpu (user)

complexarray 36.0 X

floatarray 0.5 X

MAIN 0.2 X

Cancel Up Show Source Help

Resource OH cpu (user), Procedure complexarray

```

a(i,:) = a(i,:)+[1:32] | 0.4 X
do 10 j = 2,32
  a(j,:) = a(j-1,:)+(0,32,0) | 17.2 X
10 continue
b(i,:) = (1,0,0,0) | 0.3 X
b(i,:) = b(i,:)+[1:32] | 0.4 X
do 20 j = 2,32
  b(j,:) = b(j-1,:)+(32,0,0,0) | 17.1 X
20 continue
a = a * b | 0.2 X

```

Cancel Help

Welcome to Prim, Version 1.0-beta1.1.
load /u/sistat/testa/cm2/arrays.x
(1) stop at "intarray.fcm":20
Running: /u/sistat/testa/cm2/arrays.x
stopped in procedure "MAIN" at line 17
collection status will not be saved
/u/sistat/testa/cm2/arrays.x

1	2	3	4
33	34	35	36
65	66	67	68
97	98	99	100
129	130	131	132
161	162	163	164
193	194	195	196
225	226	227	228
257	258	259	260
289	290	291	292
321	322	323	324
353	354	355	356
385	386	387	388

Activities in the past year

Releases:

- Prism 1.1 was released in summer 1992
- Prism 1.2 is about to be released

What's in these releases:

- Bug-fixes
- Rounding out functionality
- Fine-tuning of user interface
- Some new features (next few slides)

Extensions to array visualization

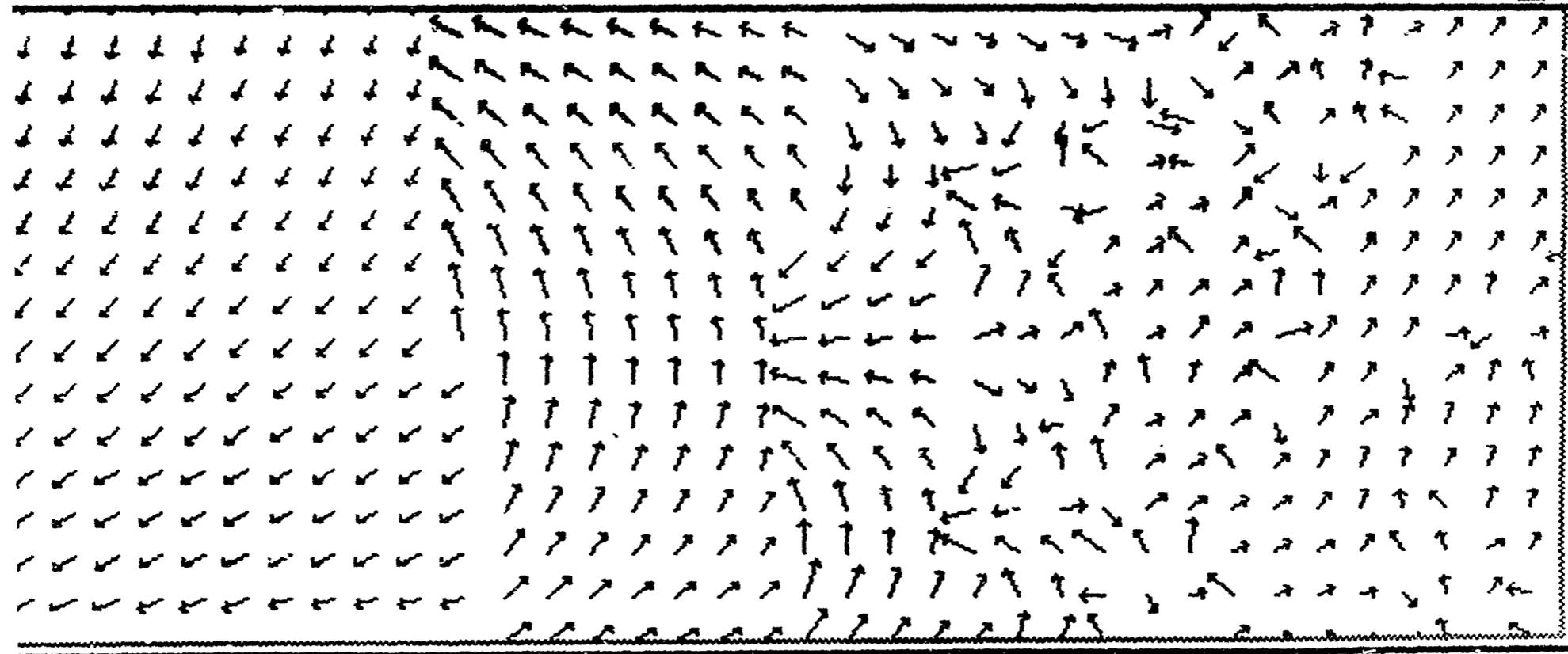
- Vector representations of complex numbers
(slide)
- Graph representation for 1-dimensional arrays
(slide)
- Surface representation for 2-dimensional arrays
(slide)

cmplx(zr,zi)

File Options

0

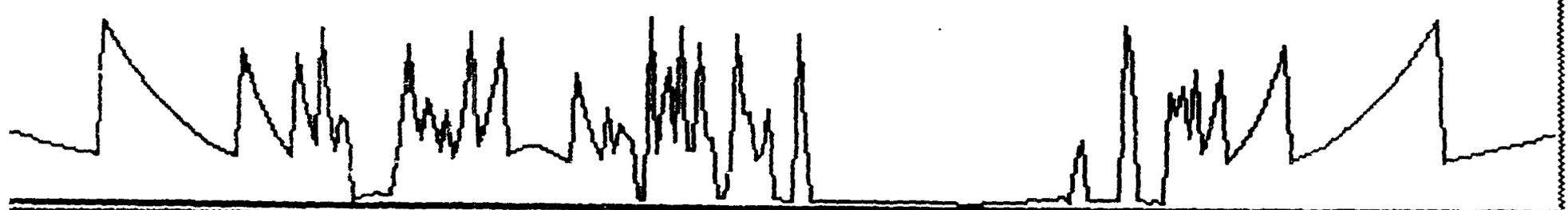
1

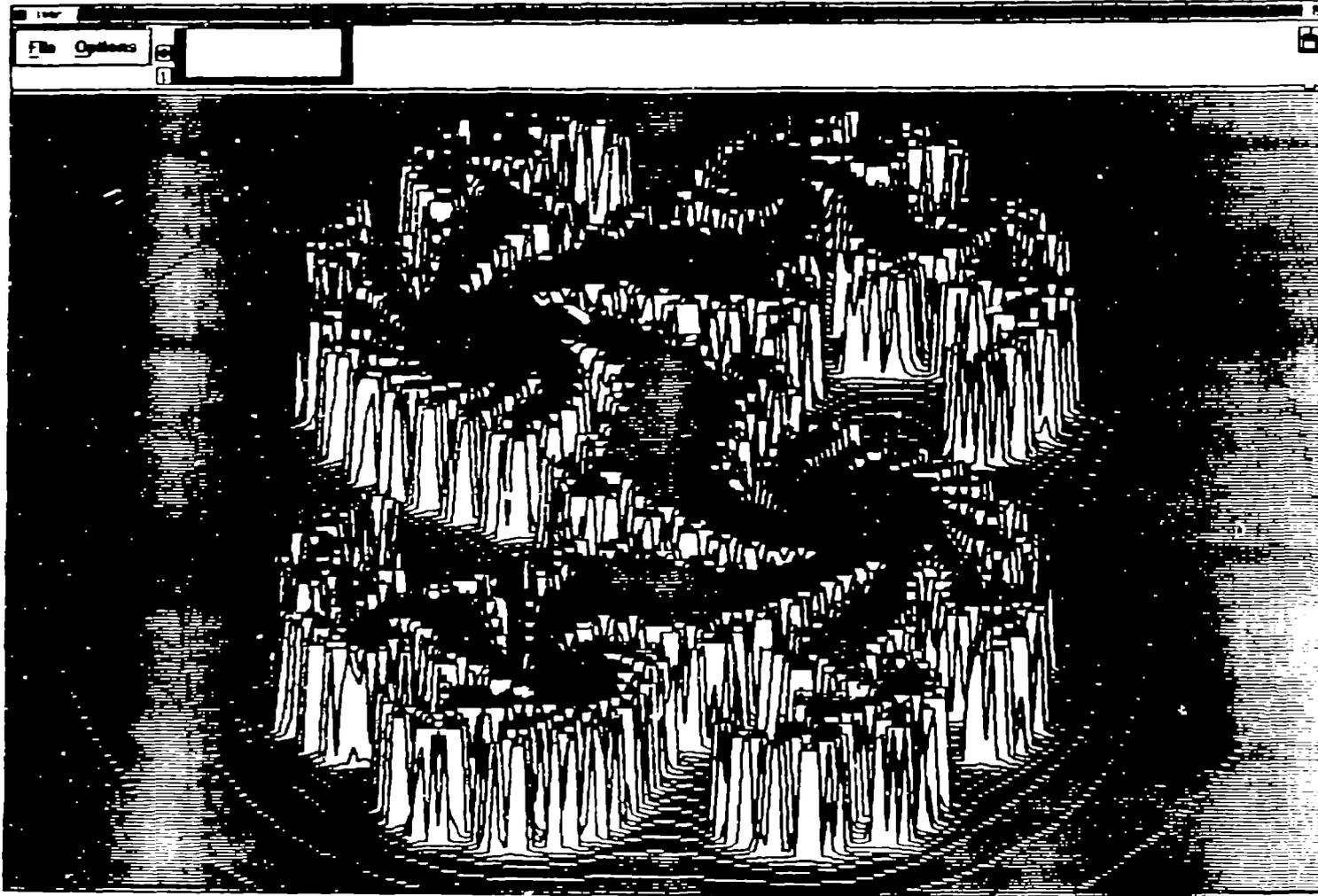


cmplx(zr,zi)

File Options

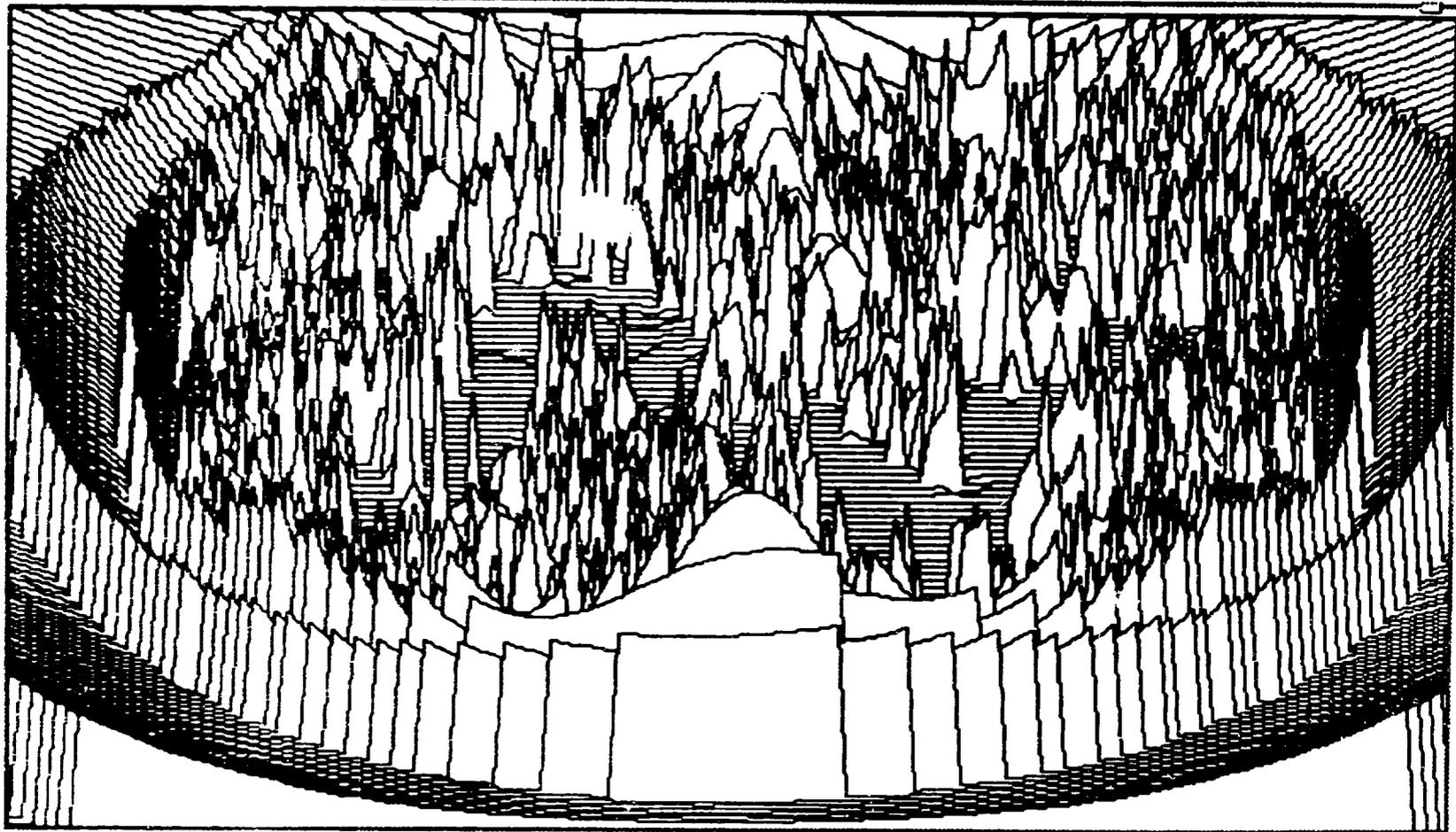
0	██████████
1	██████████





Thinking Machines Corporation

File Options



Structure Visualization

- Follows pointers to show arbitrary graphical data structures
- Automatic layout
- Zoom in/out for different levels of detail
(slides)

File Options



Zoom



```
topwidget = 0x4b02f8
draw = 0x490128
left_x = 0
top_y = 0
total_h = 364
total_w = 1072
win_h = 400
win_w = 1078
zoom = 4
win = 6291872
gc = 0x41f648
font = 0x438ee0
root = 0x4a6230
current = 0x4a6230
nodes = 0x4a6250
```

File Options

Zoom

```
topwidget = 0x4b02f8  
draw = 0x490128  
left_x = 0  
top_y = 0  
total_h = 364  
total_w = 1072  
win_h = 400  
win_w = 1078  
zoom = 4  
win = 6291872  
gc = 0x41f648  
font = 0x438ee0  
root = 0x4a6230  
current = 0x4a6230  
nodes = 0x4a6250
```

File Options



Zoom



```
topwidget = 0x4b02f8   
draw = 0x490128   
left_x = 0  
top_y = 0  
total_h = 364  
total_w = 1072  
win_h = 400  
win_w = 1078  
zoom = 4  
win = 6291872  
gc = 0x41f648   
font = 0x438ee0   
root = 0x4a6230   
current = 0x4a6230   
nodes = 0x4a6250 
```

```
ext_data = 0x0   
fid = 6291680  
direction = 0  
min_char_or_byte2 = 0  
max_char_or_byte2 = 127  
min_byte1 = 0  
max_byte1 = 0  
all_chars_exist = 1  
default_char = 0  
n_properties = 21  
properties = 0x438f38   
min_bounds = {  
  lbearing = 0  
  rbearing = 0  
  width = 8  
  ascent = -1  
  descent = -6  
  attributes = 0  
}  
max_bounds = {  
  lbearing = 3  
  rbearing = 8  
  width = 8  
  ascent = 10  
  descent = 3
```

File Options



Zoom



00
8

80
→

```
frame = 0x29f9f8  
processors = 0x4a1260  
w = 361  
h = 87  
x = 587  
y = 153  
parent = 0x4a6230  
children = 0x49e550
```

```
name = 0x15f008 "data_loop"  
id = 2  
line = 100  
parent = 0x29f9d8
```

```
length = 1  
badbits = 28  
bits = {  
  14  
}
```

```
node = 0x4a8630  
line = 100  
next = 0x49e4d8
```

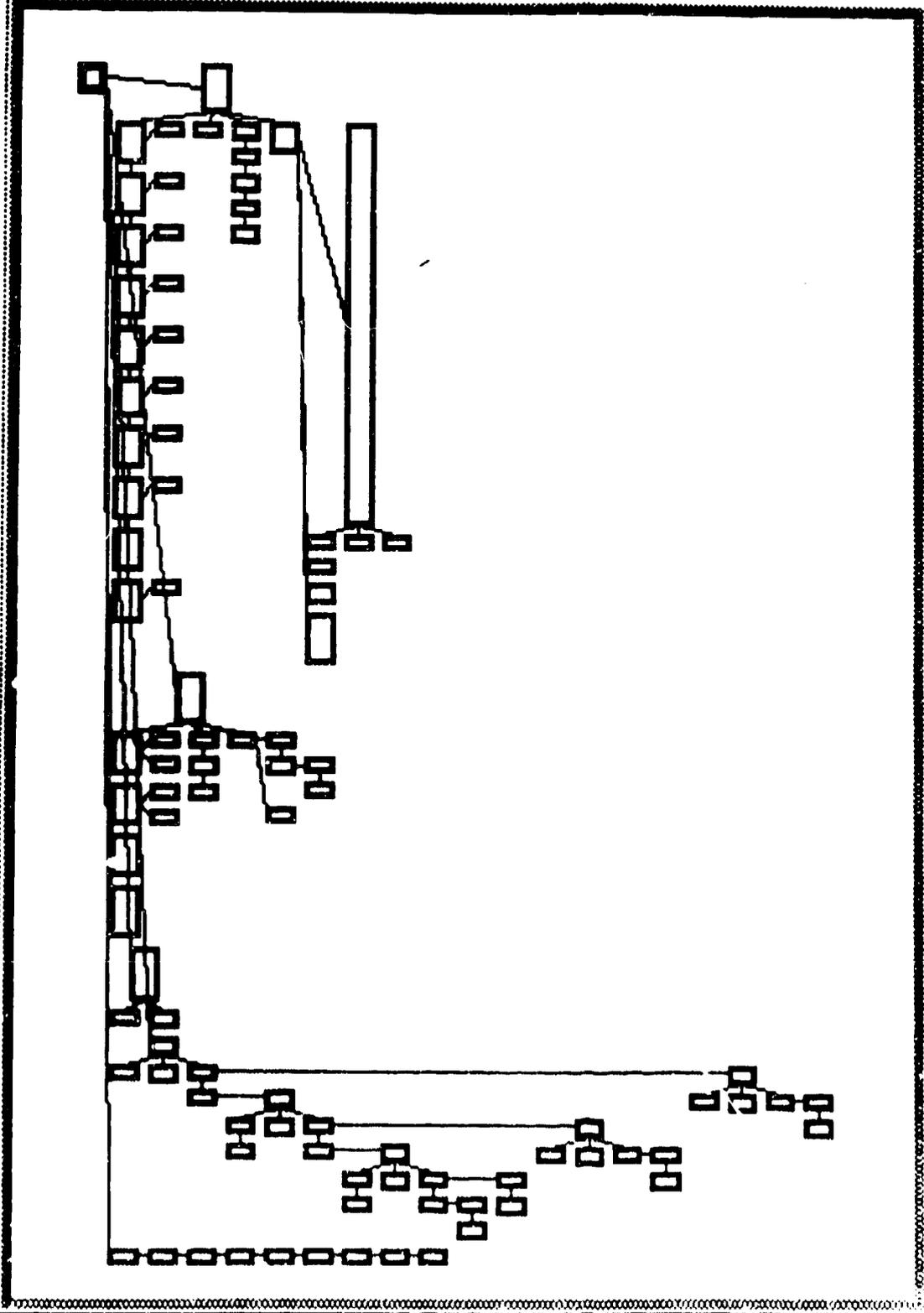
```
name = 0x15eff8 "main"  
id = 0  
line = 13  
parent = 0x0
```

```
node = 0x49e120  
line = 120  
next = 0x0
```

File Options



Zoom



Assembly-language support

- **Split window source/assembler (slide)**
- **Shows correspondence**
- **Assembler window acts like a source window**
(can set breakpoints, etc)

.

File Execute Debug Performance Events Utilities Doc Help

Load... Run Continue Interrupt Step Next Print... Display... Collection Step

Program: optest2.x Status: stopped

Line	Source File. optest2.fcm
6	complex z1,z2,z3
7	double complex dz1,dz2,dz3
8	character*10 c
9B	i1 = .false.
10	i2 = .true.
11	i3 = i1 .and. i2
12	i1 = 0
13B*	f1 = 0.0
14B	d1 = 0.0
15	z1 = (0.0, 0.0)
16	dz1 = (0.0, 0.0)
17	c = 'abcdefghihj'
18	do 20 j = 1,10
19	i2 = i1 + 1
20	f2 = f1 + 1.0

10	22cc	st	x10, [x17 + -3792]
	22d0	mov	-1, x10
	22d4	st	x10, [x17 + -3800]
	22d8	mov	0, x10
11	22dc	st	x10, [x17 + -3808]
12	22e0	mov	0, x10
	22e4	st	x10, [x17 + -3816]
13B*	22e8	ld	[x17 + -3888], xF0
	22ec	st	xF0, [x17 + -3840]
14	22f0	ldd	[x17 + -3944], xF0
B	22f4	std	xF0, [x17 + -3896]
15	22f8	ldd	[x17 + -3952], xF0
	22fc	sethi	xhi(0x44000), x10
	2300	std	xF0, [x10 + 184]
16	2304	ldd	[x17 + -4080], xF0


```

s
stopped in procedure "MAIN" at line 11 in file "optest2.fcm"
s
stopped in procedure "MAIN" at line 12 in file "optest2.fcm"
s
stopped in procedure "MAIN" at line 13 in file "optest2.fcm"
print i1
i1 = 0
where
MAIN(), line 13 in "optest2.fcm"
main() at 0x1ad34
CMTS_ScalarMain() at 0x1a8f8
(2) stop at "optest2.fcm":13
(3) stop at 0x22f4

```

Support for preprocessed source

- **Support automatic F77->CMF translation**
- **Split window F77/CMF**
- **Shows correspondence**
- **Either window can act as source window**
(set breakpoints, print variables by pointing, etc)

Prism @ vanilla.think.com

File Excute Debug Performance Events Utilities Doc Help

Load... Run Continue Interrupt Step Next Print... Display... Collection Step1

Program: flo67 Status: stopped

Source File: /users/cmsg7/ttitle/flo67/flo67.fcm

```

371
372 C
373 C*****
374 C Turn vectorization back on for the rest.
375 C
376
377 c INITIALIZE CM ARRAYS
378B*   pp0 = 0.
379     zw0 = 0.
380     zw10 = 0.
381     zwr0 = 0.
382
383     pi = 0.
384     zw1 = 0.
385     zw11 = 0.

```

```

374 C Turn vectorization back on for the rest.
375 C
376
377 c INITIALIZE CM ARRAYS
378B-   do 1000, k=0,ke1
378 -   do 1000, j=0,jem
378 -   do 1000, i=0,iem
378 *   pp0(i,j,k)=0.
379     do 1000, h=1,5
379     zw0(h,i,j,k)=0.
380     zw10(h,i,j,k)=0.
381     zwr0(h,i,j,k)=0.
382
383     1000 continue
384
385     do 1001, k=0,ke1

```

```

print pp0
pp0 =
(0:,0,0) 0.000000 0.000000 0.000000 0.000000
(4:,0,0) 0.000000 0.000000 0.000000 0.000000
(8:,0,0) 0.000000 0.000000 0.000000 0.000000
(12:,0,0) 0.000000 0.000000 0.000000 0.000000
(16:,0,0) 0.000000 0.000000 0.000000 0.000000
(20:,0,0) 0.000000 0.000000 0.000000 0.000000
(24:,0,0) 0.000000 0.000000 0.000000 0.000000
(0:,1,0) 0.000000 0.000000 0.000000 0.000000
(4:,1,0) 0.000000 0.000000 0.000000 0.000000
(8:,1,0) 0.000000 0.000000 0.000000 0.000000
(12:,1,0) 0.000000 0.000000 0.000000 0.000000
(16:,1,0) 0.000000 0.000000 0.000000 0.000000
(20:,1,0) 0.000000 0.000000 0.000000 0.000000

```

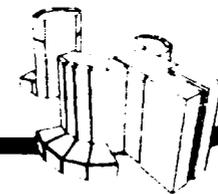
Future directions

- Remote debugging.
- Support for MIMD (message-passing) programming models.
- Software watchpoint technology
(more efficient watchpoints and conditional breaks)
- A more distributed internal implementation of Prism.

cdpx Debugger

Release 6.1

- ❑ Support for multitasked codes.
- ❑ SCC 3.0 and CFT77 5.0 Compiler release support
- ❑ *invoke* command - execute user program to process data.
- ❑ *reinit* command - reinitialize symbol tables
- ❑ *switch* command - changes debugging images

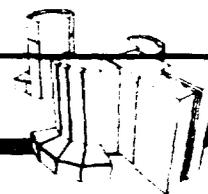


cdbx Debugger

Multitasked Codes

- ❑ Supports both autotasked and macrotasked codes
- ❑ Allows switching between tasks and logical CPUs
- ❑ *TASK* command:

```
[3] stopped in mtask$c.subtask2 at line 43
 43==>                                     geirb = (2 * i) -j;
Current task has id 3 running on logical cpu 95531.
(cdbx) task
  Internal
  task id   User defined task value   Task status
-----
->         3   00000000000000000000000002   running on logical cpu 95531
                                     (stopped in subtask2 at line 43 in file mtask.c)
         2   00000000000000000000000001   running on logical cpu 95530
                                     (stopped in subtask1 at 0p11446d)
         1   00000000000000000000000000   waiting for task 2
```

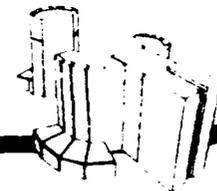


cdbx Debugger

Multitasked Codes

□ CPU command:

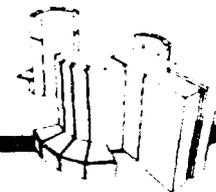
```
(cdbx) cpu
  Cpu id  Active/  If running,  User defined
         Inactive  Internal    task value
         -----
          90474  Inactive
->      95531  Active      3            000000000000000000000002
         (stopped in subtask2 at line 43 in file mtask.c)
          95530  Active      2            000000000000000000000001
         (stopped in subtask1 at 0p11446d)
(cdbx)
```



cdbx Debugger

Release 7.0/7.C

- ❑ Type casting when printing variables**
- ❑ Hardware watchpoint feature and other C90 support**
- ❑ A 'fuzzy match' symbol lookup option**
- ❑ Xwindows drag-and-drop support with other vTools**
- ❑ 'printf' Xwindows menu button**
- ❑ Improved internal interprocess communications (stdout properly output)**

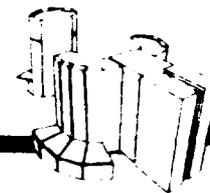


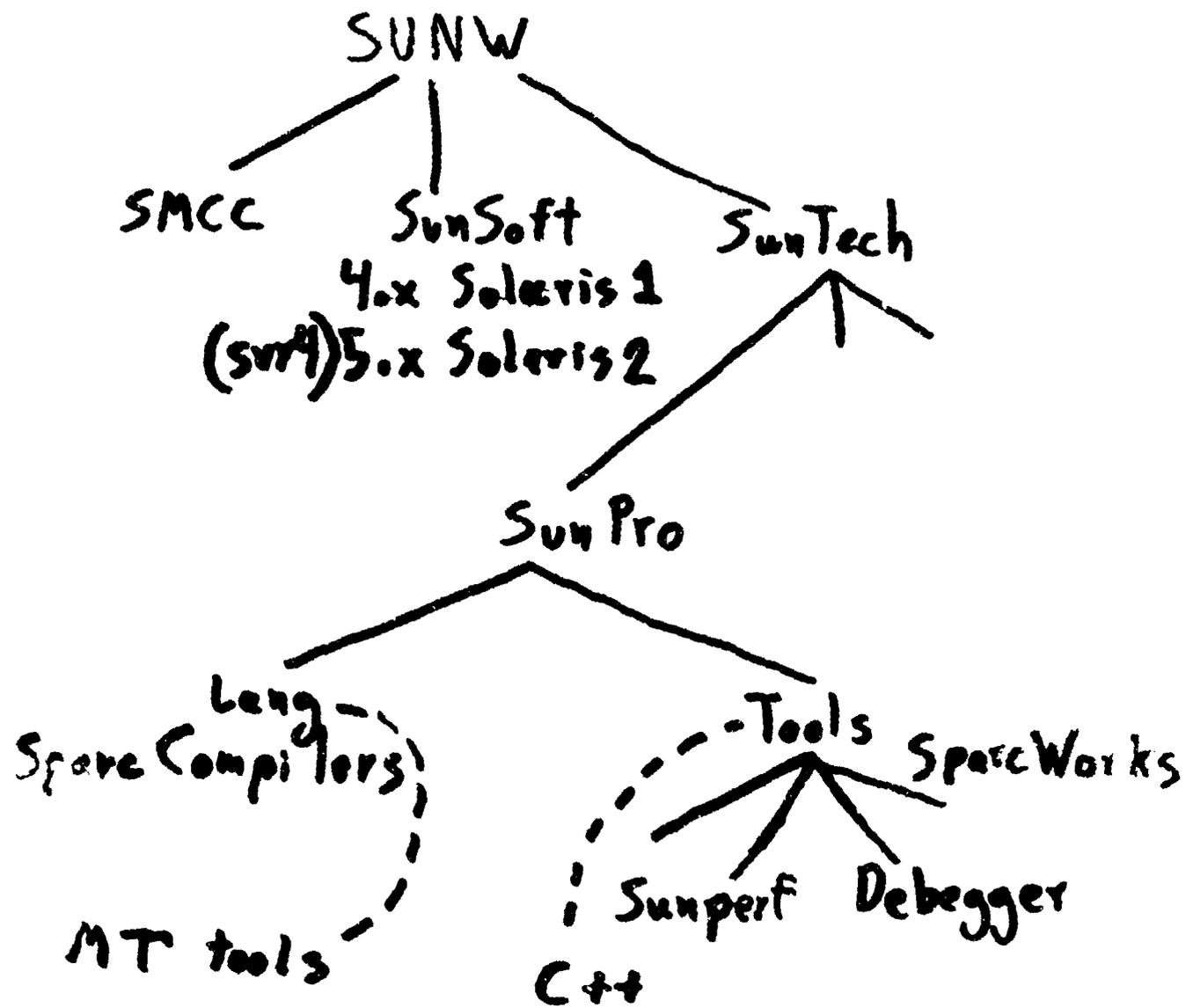
cdbx Debugger

Release 8.0

- ❑ **Unbundled / binary release with other vTools**
- ❑ **Support for CF77 6.0, SCC 4.0 compiler releases**
- ❑ **Support for new C++ compiler including name demangling**
- ❑ **Support for new FORTRAN 90 compiler**
- ❑ **Faster conditional breakpoints**
- ❑ **MPP Emulator support**
- ❑ **CDBX quick reference card**
- ❑ **Many other fixes/improvements**

Cray Research, Inc.





SW 2.0.1 Features

dbx: Bug Fixes

Significant C++ Support

Lazy Stabs

Improved language specific semantics

Debugging of Shared Libraries

History + simple SAVE/RESTORE/"UNDO"

-Og

testsuite 1003

sample: sampling

lots of GUI

lbr-int: sympathetic stop "synchronous" debugging

93

92

← Now ⁶⁺² 2.0.1 FCS

2.0.1 a 4.x & 5.x

91

4+1

dbx restarted

90

89

SC1.0

88

dbx

C++ hacks

3 Architectures

dbx tool

OW ??

- ksh based command language

More C++ support: virtual function calls
exceptions
templates
compiler

Regularized event mgmt:

```
WHEN 0 { do-x;  
do-y; }
```

Tool talk interface

Page Protection based watchpoints

Patching technology: memory lock
fast breakpoints?
ld/st based watchpoints?

GUI data inspector

Separate User I/O window

fork/exec following

more

F90

["asynchronous" dbx-nt
multi process / remote debugging
DOE support

UDB: A Parallel Debugger for the KSR1

Steven A. Zimmerman
Kendall Square Research
Waltham, Massachusetts

September 28, 1992

Abstract

UDB is a parallel debugger developed at Kendall Square Research to run on the KSR1. Its command set is in general a superset of those of GDB and dbx, reflecting the philosophy that the user is well served by having available the full variety of features present in traditional debuggers. In addition, UDB has integrated into this command set many features specifically designed for debugging parallel programs. Since the KSR1 runs the OSF operating system, parallel programs running on the KSR1 are typically written using pthreads. UDB's parallel debugging features are designed to be used with this type of multithreaded program, although they can also be used with programs that use just the basic kernel threads. Breakpoints and traces may be set in one thread, all threads, or any combination of threads. The user may let all threads execute, or may restrict execution to one or a set of threads. Data expressions may also be evaluated in the context of any given thread, or sequentially in all threads or a group of threads. In general, any valid UDB command can be made to execute in the context of a given thread, or in the context of all threads where this makes sense.

UDB also has a complete windowing facility. The user can create windows reflecting all of his or her threads, or windows reflecting just a single thread. The difference between the two types is that windows reflecting all threads contain breakpoint and PC markers for all threads, while windows reflecting a single thread contain breakpoint and PC markers for that thread only. The user can create source or instruction windows of arbitrary size; a single program I/O window is also available. Since the typical program has many more threads than can be displayed in windows, UDB has a mechanism for automatically displaying interesting threads in available windows.

1 UDB Basics

Kendall Square Research's KSR1 is a massively parallel supercomputer that presents a shared memory model to the programmer. This model allows programs to be written for the KSR1 in much the same way as for single processor machines, and so most standard debugging techniques are useful with programs that run on the KSR1. For this

reason, the UDB debugger was developed with the goal of incorporating the best features of current state-of-the-art serial debuggers, as well as a number of new features specifically useful for debugging parallel programs. The GDB debugger was selected as having a wide variety of debugging features as well as a clean and easy-to-use command set, and so UDB's user interface is based largely on GDB's. As the command set of dbx has much in common with that of GDB, and as dbx has several important features missing in GDB, most notably trace and assertion facilities, the dbx command set has also been incorporated into UDB. By supporting both command sets, UDB presents a user interface that is already familiar to a very large proportion of the Unix user community, as well as one that offers a full range of debugging facilities.

In addition to standard GDB and dbx facilities, UDB offers a number of new features that are useful for general purpose debugging. For example, UDB supports the debugging of both C and Fortran programs. Since C is case sensitive while Fortran is not, UDB allows case sensitivity to be turned on or off depending on the language of the object module. UDB's signal handling mechanism has been extended so that signals can have command lists associated with them that are executed when the signal is received, analogous to the way that breakpoint command lists are executed when breakpoints are hit. User-defined commands have been extended from the GDB model so that they are allowed to take arguments, just as in the dbx-style alias feature. UDB has "if" and "for" commands, whose function is similar to the corresponding keywords in the C language. These commands are especially useful in writing command lists, command files, and user-defined commands. UDB also allows recording of either the commands being issued in the current session, or a complete record of the entire session output (including commands), or both. The first facility is very useful for interactive development of command scripts. UDB also has a small command file debugger which allows the user to step through command files a line at a time. Finally, UDB has extended the GDB command line editing facility to include a fairly complete vi mode, although some would not consider this to be a feature.

UDB has an optional windowing system that is available for debugging sessions. A source window may be created containing the current source file. Whenever program execution stops, the source window is automatically updated to reflect the new location of the PC. Markers indicate the location of the PC and all enabled breakpoints. Various UDB commands that list one or more lines of source code automatically update this window as well. Within the window, a basic set of either emacs or vi commands may be used to navigate around the window. Single-letter versions of many of the most frequently used UDB commands are also available within the window and operate typically on the current cursor position. These commands allow the user to set and remove breakpoints; run, step, and continue the program, print out the value of variables; etc.

Figure 1 Source Window

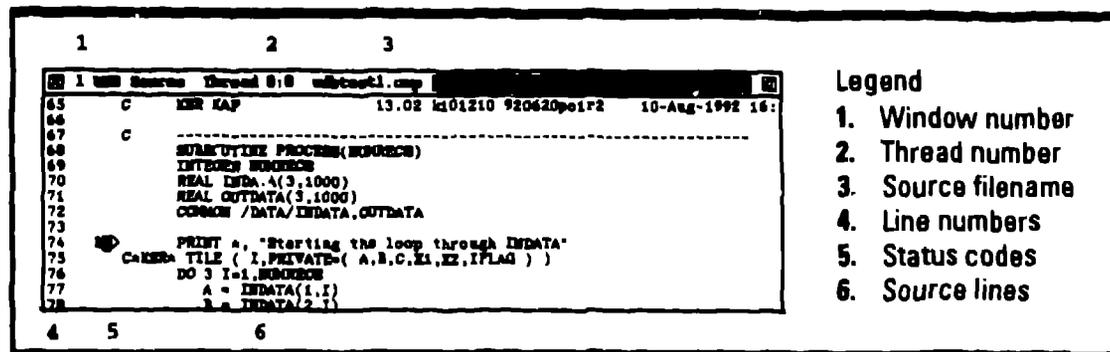


Table 1 Status Codes

Window	Code	Meaning
Source	>	PC of the window's thread, if stopped
	R	PC of the window's thread, if in run state
	B	Breakpoint in which the thread participates
	W	Synchronous breakpoint in which the thread participates

An instruction window may be created that contains the disassembled program instructions around the current PC. As in the case of the source window, the current location of the instruction window may be manipulated either by specific commands within UDB or directly by commands within the window. Certain commands within the instruction window have a slightly different meaning than when used in the source window; specifically, the stepping commands when used in the instruction window will step by machine instruction rather than by source line. Also, there are up to three different PC markers, representing the addresses of the three instructions that are currently in the execution pipeline.

Figure 1-2 Instruction Window

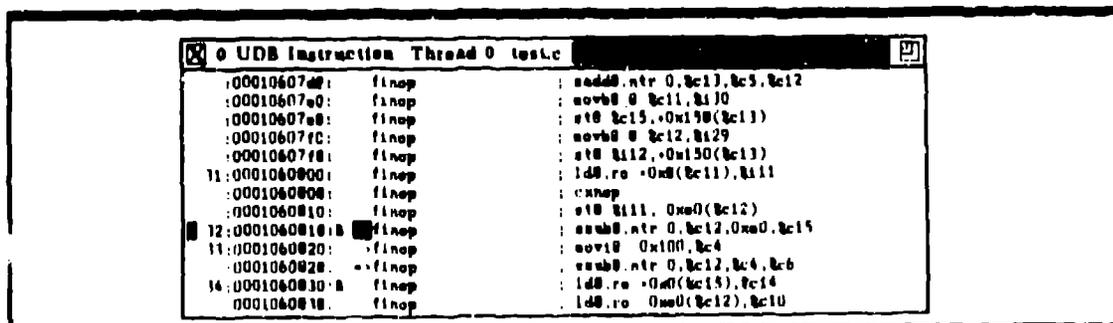


Table 1-1 Instruction Window Status Codes

Code	Meaning
>	PC
->	PC for the next pipelined instruction
=>	PC for the second pipelined instruction
B	Breakpoint in which the thread participates
W	Synchronous breakpoint in which the thread participates

A program I/O window is also available; all the program's input and output may be directed to this window instead of to the same place where UDB is running. There are also several other types of windows available under UDB, but as these are all used specifically with parallel debugging, they are described later.

2 Parallel Debugging

2.1 Managing Multiple Threads

Since the KSR1 runs the OSF operating system, parallel programs running on the KSR1 are typically written using pthreads. UDB's parallel debugging features are designed to be used with this type of multithreaded program, although they can also be used with programs that use just the basic kernel threads. UDB identifies individual threads by both a barrier ID and a thread ID within the barrier. Threads that are operating within the serial portion of the program and are therefore not associated with a barrier are assigned a barrier ID of zero, since barrier numbering normally starts at one. To remind the user of which thread is current, the barrier ID and thread ID are displayed in the user prompt as follows:

```
uDB (4:2) :
```

In this case, the current thread is thread 2 of barrier 4. Since a thread may be participating in multiple barriers at once, the `info numbers` command was created to list all the other names of the thread. In UDB output, the thread is always listed with the name found in its innermost barrier, although UDB will recognize any valid name on input. An `info threads` command also exists; this command lists either all threads or a specified subset of threads, and gives information about each thread including the thread ID, the state of the thread, and its current PC.

```
(udb) info threads
# Thread ID H State Num Address Procedure File:Line
> 4:2* 17682 N step 5 0x1040938 main myprog f.23
...
```

By default, most commands are executed in the context of the current thread only. The main exceptions to this rule are the breakpoint and trace commands; these apply to all threads by default because the user typically does not know ahead of time which thread or threads will be of interest. These commands can be restricted to apply to the current thread only by prefacing them with the `thread` keyword. Any command, including the breakpoint and trace commands, can be made to apply to a specific thread or to a group of threads by prefacing the command with the `thread` keyword followed by a list of one or more threads. Finally, those commands that normally apply to just the current thread can be made to apply to all threads by prefacing them with the keyword `all`. In these last two cases, commands that normally produce output will produce output for each thread specified. Typing the `thread` keyword followed by a single thread number but without a following command causes the current thread to be switched to the specified thread.

2.2 Running Multiple Threads

Special issues arise when running multiple threads with breakpoints. One of the most crucial has to do with how to handle other threads when one thread hits a breakpoint. UDB monitors the child task's exception port continuously, and suspends the child task as soon as a breakpoint exception is encountered. At this point, the `info threads` command may show a number of threads in the `run` state, but since the task is suspended, they are not really running, and the UDB documentation describes them as `frozen`. Once UDB has stopped due to a breakpoint, it is possible to switch between threads that are stopped or frozen, examine and change their data, do stack backtraces on them, etc. By default, the `continue` command will continue the current thread plus all threads in the `run` state. The `continue` command may be given an optional thread specification to continue threads other than or in addition to the current thread; typing `all continue` will cause all threads to continue regardless of their run state. The exception to this is that individual threads may be prevented from executing by specifying their names in a `hold` command; held threads never execute until after they are freed by means of a `free` command. Both stopped and frozen threads may be held.

The main reason that the decision was made in UDB to suspend the entire task when any thread hits a breakpoint is that normal program execution patterns are maintained by doing so. Parallel programs typically have locks and synchronization code, and these can cease to function properly if some threads are stopped at breakpoints while other threads continue running. Additionally, there is the problem of what to do when many threads hit a breakpoint in close sequence. Either the screen may be flooded by similar breakpoint messages one right after the other, or if the decision is made to suppress such messages, threads may silently and unpredictably change state

from running to stopped. Suspending the entire task avoids these problems, and allow breakpoints to be handled sequentially at the user's option. When some threads have hit a breakpoint and it is no longer interesting to watch other threads hit the same breakpoint, the user can disable the breakpoint in the other threads or in all threads, and continue execution.

Single stepping in a multithreaded program works somewhat differently in UDB than just continuing threads. When a thread is being single stepped, the user typically wants to study the execution of just that thread. If frozen threads are also allowed to run, as with the `continue` command, then they may make asynchronous changes to data of interest, or worse, one of them may hit a breakpoint during the stepping operation. To avoid these problems, only the stepped thread or threads are allowed to execute during the stepping operation.

2.3 Synchronous Breakpoints

Normally, a breakpoint suspends the entire task as soon as any thread hits it. When dealing with parallel programs, often the opposite behavior is desired, that is to say, the user would like the program to stop only after all threads belonging to a given barrier hit a specific breakpoint. To handle this case, synchronous breakpoints have been introduced. Whenever a thread hits a synchronous breakpoint, it stops, but if other threads belonging to the barrier are still running, execution of the program continues. Only when all threads in the barrier have stopped at the breakpoint does UDB suspend the task and return control to the user. An exception to this case is if some of the threads stop at this breakpoint and then another thread hits a regular breakpoint; in this case, the program will stop immediately. If the thread containing the regular breakpoint is resumed, and no other regular breakpoints are encountered, the program will then continue until the synchronous breakpoint is hit in all threads.

If a synchronous breakpoint contains a command list, the command list will be executed by each thread at the time that it first hits the breakpoint.

2.4 Windows and Multiple Threads

In multithreaded programs, several other window types are available in addition to those described above. The first of these is known as a shared source window. This window is similar to a regular source window, except that it contains breakpoint and PC markers for all of the program's threads. Since a program may contain hundreds of threads, and since breakpoints may exist in either some threads or in all threads, the information for these markers must be encoded efficiently. UDB uses the following scheme:

Figure 2-3 Shared Source Window with Status Codes

```

63 C KSR MAP 13.02 ki01210 920620polr2 10-Aug-1992 16:
64
65 C -----
66 SUBROUTINE PROCFIN(NUMRECS)
67 INTEGER NUMRECS
68 REAL INDATA(3,1000)
69 REAL OUTDATA(3,1000)
70 COMMON /DATA/INDATA,OUTDATA
71
72
73
74 B PRINT = "Starting the loop through INDATA"
75 CAESAR TILE ( I,PRIVATE=( A,B,C,X1,X2,IFLAG ) )
76 DO 3 I=1,NUMRECS
77 nR A = INDATA(1,I)
78 B B = INDATA(2,I)
79 w C = INDATA(3,I)

```

Status codes

Table 2-1 Status Codes

Window	Code	Meaning
Shared Source	<i>nR</i>	PC of a frozen thread. <i>n</i> is the thread's ID in a barrier.
	<i>n*R</i>	PC of several frozen threads. <i>n</i> is the number of threads running at that point.
	<i>n></i>	PC of a stopped thread. <i>n</i> is the thread's ID in a barrier.
	<i>n*></i>	PC of several stopped threads. <i>n</i> is the number of threads stopped at that point.
	B	Breakpoint that applies to all program threads
	b	Breakpoint that applies to only a subset of threads
	w	Synchronous breakpoint that applies to all threads in a barrier
	w	Synchronous breakpoint that applies to only a subset of threads in a barrier

Note that this method allows the user to easily distinguish between regular breakpoints and synchronous breakpoints, which behave rather differently.

Shared instruction windows are also available in UDB. These work similarly to shared source windows, with some minor limitations.

An important issue that needs to be addressed when using windows with multiple threads is how many windows to use, and which windows get displayed in which threads. When UDB is started up with the default windows options, it creates one regular source window, one shared source window, and one program I/O window. When the program starts executing, the source window displays the program's initial thread. Whenever any thread hits a breakpoint that causes program execution to stop, that thread is displayed in the source window. Also, if the user manually changes the current thread by means of the `thread` command, the new thread is automatically displayed in the source window.

If more than one regular source window is desired, the user can either create multiple source windows at startup or create them individually at any time during UDB execu-

tion. When multiple source windows exist, the events described above that cause a thread to be displayed in a source window use the available source windows in round robin fashion. If the user wants one or more source windows to always display a certain thread, he can tell UDB to remove them from the list of windows available for new thread display.

UDB also allows the user to create prompt windows for use with multithreaded programs. These windows work exactly like the main UDB command window, allowing the user to execute commands and see their output. They are useful because they are created on a per thread basis, and thereby allow the user to segregate UDB command streams by thread number.

SQUARE

UDB

A PARALLEL DEBUGGER FOR THE KSR1

Steven A. Zimmerman
z@ksr.com

SUPERSET OF GDB AND DBX

Traditional serial debuggers work well with KSR's programming model

GDB selected as powerful and well known serial debugger

DBX selected for compatibility reasons and trace and assertion facilities

ADDITIONAL SERIAL DEBUGGER EXTENSIONS

Full support for debugging Fortran programs

Command lists for signals

Arguments for user-defined commands

New control commands "if" and "for"

Command and session recording

Command file debugging

Vi mode for line editing

Source, instruction, and program I/O windows available

Markers indicate the location of the PC and all enabled breakpoints

Windows automatically updated to current location whenever program stops

UDB commands that list source files update source windows as well

Motion within a window is accomplished by either emacs or vi commands

Frequently used UDB commands have single letter window versions

```

1  cTSTc Basic test for parallel stuff - a'la' linpack
2  cc-----
3      parameter (n=16)
4      common / aa/ a(n,n)
5
6  c    --- Initialization
7  B  do 1 i = 1,n
8  B    do 1 j = 1,n
9      a(i,j) = 1.0
10     1  continue
11
12  c    --- the a'la' linpack loop - in parallel
13      nml = n - 1
14      do 60 k = 1, nml
15
16          kp1 = k + 1
17          t = 1.0
18
19      C*KSR* TILE (j, tilesize=(j:i) )
20          do 30 j = kp1, n
21              do 31 i =1, n-k
22                  a(k+i,j) = a(k+i,j) + t*a(k+i,k)
23              31  continue
24              30  continue
25      C*KSR* END TILE
26
27      60  continue
28
29      end
30  c-----

```

```

:0001090918: finop ; ssub8.ntr 0,%c13,%c4,%c13
:0001090920: finop ; st8 %c11,+0x170(%c13)
:0001090928: finop ; st8 %c12,+0x178(%c13)
:0001090930: finop ; st8 %i29,+0x140(%c13)
:0001090938: finop ; st8 %i30,+0x138(%c13)
:0001090940: finop ; st8 %c14,+0x168(%c13)
:0001090948: finop ; st8 %i12,+0x158(%c13)
:0001090950: finop ; movi8 0x180,%c5
:0001090958: finop ; mov8_8 %c10,%c11
:0001090960: finop ; sadd8.ntr 0,%c13,%c5,%c12
:0001090968: finop ; movb8_8 %c11,%i30
:0001090970: finop ; st8 %i13,+0x150(%c13)
:0001090978: finop ; movb8_8 %c12,%i29
:0001090980: finop ; st8 %i14,+0x148(%c13)
:0001090988: finop ; ld8.ro +0x30(%c11),%c10
:0001090990: finop ; cxnop
:0001090998: finop ; cxnop
:00010909a0: finop ; sadd8.ntr 0,%c31,%c10,%c10
7:00010909a8:B movi8 0x11,%i6 ; st8 %i6,-0xe0(%c12)
:00010909b0: ->movi8 0x1,%i7 ; st8 %i7,-0xe0(%c12)
:00010909b8: =>movi8 0x10,%i2 ; cxnop
8:00010909c0:B movi8 0x11,%i8 ; st8 %i8,-0xe8(%c12)
:00010909c8: movi8 0x1,%i6 ; st8 %i6,-0xe8(%c12)
:00010909d0: movi8 0x10,%i3 ; cxnop
9:00010909d8: fmovi8 1,%f0 ; ld8.ro -0xe8(%c12),%c6
:00010909e0: finop ; ld8.ro -0xe0(%c12),%c7
:00010909e8: finop ; ld8.ro +0x28(%c11),%c8
:00010909f0: finop ; movb8_8 %c6,%i7
:00010909f8: finop ; cxnop
:00010909a00: finop ; cxnop
:00010909a08: lsh8 0x7,%i7,%i7 ; movb8_8 %i7,%c6
:00010909a10: finop ; cxnop
:00010909a18: finop ; cxnop
:00010909a20: finop ; sadd8.ntr 0,%c8,%c6,%c6
:00010909a28: finop ; sadd8.ntr 3,%c7,%c6,%c7
:00010909a30: finop ; st8 %f0,-0x88(%c7)

```

BREAKPOINTS IN MULTITHREADED CODE

Entire task is suspended

Threads in the "run" state are actually frozen

All threads may be examined and manipulated

CONTINUING AND STEPPING

The "continue" command works on the current thread and all frozen threads

Thread execution is affected by "hold" and "free" commands

Stepping commands execute only specified thread(s)

X zappa: /langwork7/users/z

Bpt 2 in thread 1:0, MAIN() at koo.f:23

23 B 31 continue

(udb[1:0]) info threads

#	Mach ID	H	State	Num	Address	Procedure	File:Line
> 1:0	16	N	break	2	0x1090dd8	MAIN	koo.f:23
1:1	18	N	run		0x1090dd8	MAIN	koo.f:23
1:2	19	N	run		0x1090dd8	MAIN	koo.f:23
1:3	20	N	run		0x1090dd8	MAIN	koo.f:23
1:4	21	N	run		0x1090dd8	MAIN	koo.f:23
1:5	22	Y	break	1	0x1090cc0	MAIN	koo.f:22
1:6	23	N	run		0x1090dd8	MAIN	koo.f:23

(udb[1:0]) quit

The program is running. Quit anyway? (y or n)

113 zappa

Breakpoint is set by default in all threads of the current barrier

Task is stopped only after all threads hit the breakpoint

Command lists are executed when the thread actually hits the breakpoint

Program stops in the thread from which "continue" was issued

THREAD BASICS

Parallel programs on the KSR1 typically use **pthread**s

UDB can **debug** programs that use either **pthread**s or **kernel threads**

Thread numbers consist of a **barrier ID** and a **thread ID**

Current thread number is always displayed in the prompt, e.g., "(udb[4:2])"

Other thread names can be listed with "info threads" command

SCOPE OF COMMANDS

Most commands execute in the context of the **current thread**

Notable exceptions are **breakpoint** and **trace** commands

"thread <command>" restricts command to **current thread**

"thread n <command>" restricts command to **thread n**

"all <command>" executes command over **all threads**

Current thread may be changed by typing "thread n"

```

1 UDB Source Thread 1:0 hoo.f
2 c73c Basic test for parallel stuff - a'la' liqash
3 -----
4 parameter (n=16)
5 rmaxm / m/ a(n,n)
6
7 c --- Initialization
8 do 1 i = 1, n
9   do 1 j = 1, n
10    a(i,j) = 1.0
11  continue
12
13 c --- the a'la' liqash loop - in parallel
14 m1 = n - 1
15 do 60 h = 1, m1
16
17   hpl = h + 1
18   t = 1.0
19
20   C=KMP TILE (j, (hloop-(j+1))
21   do 20 j = hpl, n
22     do 31 i = 1, n-h
23      a(h+1,j) = a(h+1,j) + t*a(h+1,h)
24    continue
25  continue
26 C=KMP END TILE
27
28 60 continue
29
30 end
31 -----

```

```

2 UDB Shared Source All Threads hoo.f
3 c73c Basic test for parallel stuff - a'la' liqash
4 -----
5 parameter (n=16)
6 rmaxm / m/ a(n,n)
7
8 c --- Initialization
9 do 1 i = 1, n
10  do 1 j = 1, n
11   a(i,j) = 1.0
12 continue
13
14 c --- the a'la' liqash loop - in parallel
15 m1 = n - 1
16 do 60 h = 1, m1
17
18   hpl = h + 1
19   t = 1.0
20
21   C=KMP TILE (j, (hloop-(j+1))
22   do 20 j = hpl, n
23     do 31 i = 1, n-h
24      a(h+1,j) = a(h+1,j) + t*a(h+1,h)
25    continue
26  continue
27 C=KMP END TILE
28
29 60 continue
30
31 end
32 -----

```

```

32 c73c in thread 1:0 (MAIN) at hoo.f:23
33   B 31 continue
34 wh(1:0) continue
35
36 c73c in thread 1:0 (MAIN) at hoo.f:23
37   B 20 continue
38 wh(1:0) continue
39
40 c73c in thread 1:0 (MAIN) at hoo.f:23
41 wh(1:0) solo threads
42
43   PID PPID P State Run Address Procedure File:Line
44   ---
45   16 0 0 0 break 2 0x1070400 MAIN hoo.f:23
46   17 0 0 0 break 2 0x1070400 MAIN hoo.f:23
47   18 0 0 0 break 1 0x1070400 MAIN hoo.f:23
48   19 0 0 0 run 0x1070400 MAIN hoo.f:23
49   20 0 0 0 run 0x1070400 MAIN hoo.f:23
50   21 0 0 0 run 0x1070400 MAIN hoo.f:23
51   22 0 0 0 break 1 0x1070400 MAIN hoo.f:23
52   23 0 0 0 run 0x1070400 MAIN hoo.f:23
53
54 wh(1:0) free 0
55 wh(1:0) continue
56
57 c73c in thread 1:0 (MAIN) at hoo.f:23
58   B 31 continue
59 wh(1:0) continue
60
61 c73c in thread 1:0 (MAIN) at hoo.f:23
62   B 31 continue
63 wh(1:0) continue
64
65 c73c in thread 1:0 (MAIN) at hoo.f:23
66   B 31 continue
67 wh(1:0) continue
68
69 c73c in thread 1:0 (MAIN) at hoo.f:23
70   B 31 continue
71 wh(1:0) continue
72
73 c73c in thread 1:0 (MAIN) at hoo.f:23
74   B 31 continue
75 wh(1:0) continue
76
77 c73c in thread 1:0 (MAIN) at hoo.f:23
78   B 31 continue
79 wh(1:0) continue
80
81 c73c in thread 1:0 (MAIN) at hoo.f:23
82   B 31 continue
83 wh(1:0) continue
84
85 c73c in thread 1:0 (MAIN) at hoo.f:23
86   B 31 continue
87 wh(1:0) continue
88
89 c73c in thread 1:0 (MAIN) at hoo.f:23
90   B 31 continue
91 wh(1:0) continue
92
93 c73c in thread 1:0 (MAIN) at hoo.f:23
94   B 31 continue
95 wh(1:0) continue
96
97 c73c in thread 1:0 (MAIN) at hoo.f:23
98   B 31 continue
99 wh(1:0) continue
100

```



SHARED SOURCE AND INSTRUCTION WINDOWS

Windows contain PC and breakpoint markers for all threads

PC markers indicate number of threads at given location

If only one thread at a location, PC marker identifies it

Breakpoint markers distinguish between regular and synchronous breakpoints

Breakpoint markers indicate thread scope of breakpoints

SELECTING SINGLE SOURCE WINDOWS

Stopping due to breakpoints, etc. causes thread to be displayed

Selecting new thread causes thread to be displayed

User can create additional windows

Windows may be either changeable or fixed

The Effects of Register Allocation and Instruction Scheduling on Symbolic Debugging

Ali-Reza Adl-Tabatabai and Thomas Gross
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

1 Introduction

A number of modern high-performance processors expose instruction-level parallelism as well as large register files to the compiler. Since the parallelism and the storage hierarchy are exposed, the compiler has the opportunity to exploit the parallelism in the program and to reduce the memory traffic by keeping the most frequently accessed variables in registers. Instruction scheduling and register allocation/assignment are two optimizations that are commonly included in compilers for modern processors. These optimizations, however, affect setting breakpoints and inspecting variables by a symbolic debugger, which attempts to present to the user a source-level view of program execution. The instructions for multiple source language statements are intermixed, and source variables are given different storage locations during the execution of a program.

Superscalar, (V)LIW, and (super)pipelined processors can issue and execute multiple operations concurrently. An optimizing compiler can increase the efficiency of such processors by statically scheduling independent operations for concurrent execution. However, scheduling may result in source expressions executing out of source order. If assignments are executed out of order, the sequence in which source level values are computed will be different from that specified in the source program. Consequently, if the debugger inspects a variable, the value retrieved from the variable's location may not be the value expected, since some computations specified in the source were performed out of order.

Due to the increasing gap between processor and memory speeds, cache miss penalties have become increasingly expensive. One way in which this problem has been addressed, has been to include larger register files on chip, allowing the compiler to select frequently accessed values to be kept in registers. Since there are typically many more program values than there are physical registers, a register may be assigned to different values during execution. Consequently, at a breakpoint, the register assigned to a source variable at some point in time may be holding another variable at the time the breakpoint is encountered.

Supported in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330. Work furnished in connection with this research is provided under prime contract MDA972-90-C-0035 issued by DARPA/MO to Carnegie Mellon University.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Previous work on debugging optimized code has been concerned with a number of issues, e.g., which user variables are up to date at a breakpoint (starting with [11,14]), where to find the most accurate value for a user variable ([10]), how to map source code breakpoints to stopping points in the target code ([10,4]), how to deal with specific optimizations([15]), how to deal with implementation issues ([12,7]), and how to present the information to a user ([6]).

Our work focuses on the debugger issues raised by code reordering and storage location reuse due to register assignment. We implemented the techniques described in this paper for the iWarp C compiler, which is based on the PCC2 compiler. In this paper, we present measurements of the effects of these optimizations on the ability of a debugger to recover source level values for a few benchmarks selected from a suite of numerical programs[13].

2 Background

In the next section, we briefly introduce the key concepts of our symbolic debugger. Then we discuss those features of the compiler that are relevant for this paper.

2.1 Debugger model

Our debugger model supports the base operations of control breakpoints, data inspection, and execution continuation. Control breakpoints are either synchronous, such as source level breakpoints, or asynchronous, such as program faults or user interrupts. In this paper, "execution stops at instruction *I*" means that the execution stops *before I* completes. That is, either an asynchronous breakpoint occurred during the execution of *I* (e.g., an exception), or a synchronous breakpoint was placed at *I*. The topic of mapping source statements to machine instructions has been studied by other researchers[9]. The strategy used by the debugger may restrict which machine instructions can produce breakpoints, but since we want to allow the user to interrupt the execution of a program at any moment, we do not impose any restrictions on where a breakpoint can happen. Therefore, our algorithms and our evaluation is based on the assumption that any instruction can be a breakpoint.

Data inspection is limited to source variables. The debugger does not change the state of a program except for setting breakpoints; data modification by the user is not supported. When the user inspects a variable, the value stored in the variable's location may be irrelevant because the variable has not been initialized during the execution of the program. There are two possible policies that a debugger can adopt:

1. Warn the user that a variable is uninitialized.
2. Let the user beware, do not notify the user.

In the absence of support provided by the runtime system (e.g., path descriptors [15]) or the architecture (e.g., memory tags), the first option requires that the debugger obtains program flow analysis information from the compiler. If no definition of a user variable *V* reaches a point *L* in the source, then *V* is uninitialized whenever the program breaks at *L*. This data flow problem is known as reaching definitions[3]. Note that even if policy one is adopted, the debugger cannot help in the case that definitions reach on some but not all paths to *L*.

When the debugger is invoked as a result of encountering a control breakpoint, the address in the object code where the breakpoint occurs is called the *object breakpoint*, and the source statement where the breakpoint is reported is called the *source breakpoint*.

At a breakpoint, the debugger must determine if a variable is *resident*. A variable V is called resident if the debugger can find a storage location that holds the value of V , otherwise V is *nonresident*. There are several methods a debugger can use to answer the residency question, they are discussed in Section 3. A register promoted variable V is called *evicted* if the register assigned assigned to V may be holding the value of a variable other than V at the breakpoint; this occurs if V 's register has been reassigned to another variable. An evicted variable must always be reported as nonresident by the debugger (unless the debugger attempts recovery).

Finding a variable's residence is only the first step. If the variable is resident, there is no guarantee that the storage location holds the value that the user expects for this source breakpoint. A source variable whose run time value at a breakpoint is different from its expected source value due to re-ordering is called a *noncurrent* variable. A variable V is *endangered* if the debugger detects that V may be noncurrent. Only source variables can be noncurrent or nonresident; compiler-generated temporaries can never be inspected by the user, so the debugger never has to display those values. Also, note that noncurrency only applies to resident variables since only resident variables have a runtime value.

Consider the source code in Figure 1 and the object code generated shown in Figure 2. Variables d and f have been assigned the same register R4, and variable c has been assigned register R3. No other storage location holds c , d , or f . Note that register R3 is also used to hold an expression temporary at instruction I3. Furthermore, on entrance to this block of code, d is dead, and register R4 (which holds the value of d upon exit of this basic block) contains the value of f . Upon exit, f is dead.

Now let us consider the task of a symbolic debugger at different breakpoints. For example, each of the floating point addition operations may cause an exception. If an exception occurs at the fpadd instruction I3, the debugger reports that execution halted at statement S1. At this breakpoint, d is reported nonresident, since R4 holds a value belonging to variable f .

Now consider a floating-point exception during execution of I5. This is reported as occurring at statement S3 in the source code. At this breakpoint, d is still nonresident; c is also nonresident because of the assignment to R3 at instruction I3. a is noncurrent because its assignment from statement S1 has been delayed by the code scheduler so that I6 is executed after I5 (the breakpoint).

Similarly, when we analyze the situation at a breakpoint caused by instruction I7, which is reported as a breakpoint at statement S2, c and f are reported as nonresident. R3 still contains the expression temporary computed by instruction I3, and d is noncurrent, since the assignment of statement S2 has already been performed at I5, before the breakpoint.

Note that if no assignments to d reach this block of code in the source, d can be reported as uninitialized rather than noncurrent or nonresident, at any of the breakpoints.

```

a = b+c;    /* S1 */
c = e+g;    /* S2 */
d = a+f;    /* S3 */

```

Figure 1: Example source code. All variables are floating point.

The debugger must detect the set of noncurrent and nonresident variables and report them as such in response to a user query. That is, it is acceptable that the debugger cannot display the value of a variable in response to a user query, but the debugger is not allowed to provide misleading information. The debugger may attempt to recover the value of noncurrent or nonresident variable, but recovery may not always be successful. (Notice the difference between noncurrent and nonresident variables: for a noncurrent variable, the variable's location contains either an old or future value. A nonresident

```

I1: R1 <-- load b
I2: R2 <-- load g
I3: R3 <-- fpadd R1, R3 -- b+c
I4: R1 <-- load e
I5: R4 <-- fpadd R3, R4 -- d = a+f
I6: a <-- store R3 -- a =
I7: R3 <-- fpadd R1, R2 -- c = e+g

```

Figure 2: Object code generated for source of Figure 1.

variable is a variable where the debugger cannot determine the home location, and therefore no value can be presented.)

2.2 Compiler framework

The iWarp C compiler (release 2.5) performs local code compaction for the iWarp processor. iWarp is an LIW machine, with 128 registers, of which 94 are available to the compiler. In a single cycle, the iWarp can execute a floating point multiplication, a floating point addition, 2 integer operations or memory accesses, as well as a loop termination test[5]. Compaction may cause function calls to be reordered with respect to other operations.

Local variables that are not aliased are promoted to a register by the optimizer. These variables along with compiler temporaries are allocated registers from an infinite pool of virtual registers. Virtual registers are assigned physical registers after code scheduling, using graph coloring. Live ranges are not split, and promoted variables have *no* home locations in memory. Therefore, a promoted variable resides in its assigned register throughout its live range. The assigner attempts to assign caller saved registers to live ranges that do not span function calls. Register subsumption or coalescing[8] is performed to minimize the number of register moves. This optimization assigns the same register to two virtual registers whose live ranges do not conflict, but are connected by a register move.

The code scheduler and register assigner of the iWarp C compiler create two problems for a debugger. First, because of code scheduling, the debugger must detect which assignments and function call operations have executed (or not executed) out of order with respect to the source stopping point, and how source level values have been affected. Second, because registers may be reassigned, the debugger must detect which of the promoted variables are resident in their assigned registers at a breakpoint. In this compiler, two types of instructions evict variables. A register promoted variable may be evicted because its assigned register is re-assigned to another variable. Or, if the variable was assigned a caller saved register, the variable may be evicted because its value is killed by a function call. Since promoted variables do not have home locations in memory, recovery of their values is difficult.

3 Detecting nonresident and noncurrent variables

Our algorithms for detecting nonresident and noncurrent variables are implemented for the iWarp C compiler, but the same techniques can be used for other processors and other languages. The C compiler was modified to pass information describing the results of register allocation and code scheduling to the debugger. The results of register allocation and assignment are described with two tables, one that

maps register promoted variables to virtual registers and another that maps virtual registers to physical registers. The intermediate representation (IR) of the program is annotated with information describing the code generated for each IR operation, and the annotated IR is consulted by the debugger.

When a breakpoint occurs at an object breakpoint O , this object breakpoint is mapped to the *IR breakpoint operation*, the operation in the IR, for which a synchronous breakpoint was reached or within which an asynchronous breakpoint occurred. The debugger performs data flow analysis on the object to detect the set of nonresident variables at O . (The decision to determine this set on demand is motivated by implementation concerns; it is perfectly possible to perform this analysis before program execution and to record the result for each possible object breakpoint.) Then the debugger determines the set of noncurrent variables, by consulting the annotated IR. The approach of annotating the IR for detecting noncurrent variables is similar to Hennessy's[11], however, our annotations model the physical registers of the target machine as well as the instruction-level parallelism exposed by the concurrent execution of multiple operations. Details of our approach are described in [1] and [2].

3.1 Detecting nonresident variables

There are two strategies for a debugger to determine which variables are nonresident. It can make a conservative approximation, or it can try to obtain the exact solution. One conservative approximation is to assume that a variable is resident only during its live range. (It must be resident during the live range, otherwise there is a compiler error!). That is, a variable is considered nonresident after its last use. The attraction of this approach is that the compiler must maintain the live range information for register allocation. The drawback is that a variable may stay in its register after its last use if the register allocator has no immediate need for the variable's register.

The second strategy is to determine when a variable has been evicted. Let $R(V)$ denote the register assigned to a register promoted variable V . A variable V becomes evicted when $R(V)$ is targeted by an instruction that writes the value of another variable or of a temporary. After its eviction, a variable is nonresident (unless recovery is undertaken).

Information about eviction is available only from analyzing the object program. Therefore to detect evicted variables, our debugger performs data flow analysis on the object program. Machine operations that target a variable V 's assigned register $R(V)$ but do not correspond to source assignments to V , are marked as causing V to become evicted.¹ On the other hand, machine operations that target $R(V)$ and *are* source level assignments to V are marked as causing V to become resident. All variables are considered resident at the source node of a program's control flow graph. Data flow analysis is then employed to track the eviction of variables along the flow of instructions: a variable V is evicted at a point O in the object if it is evicted on any path leading to O .

3.2 Detecting noncurrent variables

To detect noncurrent variables, the debugger must detect which assignments have executed out of sequence with respect to the source breakpoint, since it is these operations that affect source level values.² Therefore, the annotated IR must record the canonical (source-order) sequence of assignments as well as the order in which they are executed in the object. The source order of assignments is captured by annotating each assignment in the IR with a sequence number. The ordering defined by the sequence numbers captures the canonical execution order of the IR assignments. The IR operations

¹If a variable V is assigned a caller saved register, then a function call operation is also considered as targeting V 's register.

²Function calls also affect source level values and are also considered. For conciseness, we only mention assignments.

on the right hand side of an assignment expression E are marked with the same sequence number as E .

The order in which IR operations are executed in the object is determined by the code scheduler, and the code scheduler must pass this information to the debugger. Each IR operation may translate into multiple operations, which are placed by the scheduler into machine instructions. Therefore, each IR operation is annotated with a list of basic block schedule offsets. Each offset identifies a machine instruction in the current basic block.³ The offset of the last machine operation generated for an IR operation determines when the IR operation completes execution.

Using sequence numbers and offsets of machine operations, the debugger can determine which operations have executed out of source order at a breakpoint. Let B be the IR breakpoint operation, and O be the block offset of the object breakpoint. There are two ways that an IR assignment operation A can be performed out of source sequence:

1. A executes before the breakpoint operation B in the canonical execution order but was scheduled to execute after the breakpoint O .
2. A executes after the breakpoint operation B in the canonical execution order but was executed prematurely before the breakpoint O .

Having detected which assignments have executed out of sequence, the debugger must detect how source variables have been affected. Pointers and aliased variables complicate the analysis of noncurrent variables. The debugger must ensure that if it cannot precisely determine the currency status of a variable, it makes only inconsequential errors. That is, the debugger may not announce a variable V as current, if it is possible that V is noncurrent. Consider an assignment into a location pointed to by p :

```
...
a =                /* B */
*p = <expr>       /* S */
```

If the store of S is moved above the store of B (the compiler determined that a and $*p$ are not aliased), then the location pointed to by p is noncurrent if a breakpoint occurs at B . If the debugger can recover the address of the location stored into by $*p$, it can precisely detect which variable is noncurrent at the breakpoint. However, if this is not possible (for example p may be nonresident at the breakpoint), the debugger must report any variable that is *potentially* aliased to $*p$ as endangered (potentially noncurrent). The compiler's memory disambiguation or aliasing analysis can improve the debugger's chance to determine the currency status of potentially aliased variables. Any variable that the compiler certifies as nonaliased will not be noncurrent due to $*p$.

3.3 Noncurrency caused by register subsumption

A register move operation copies the contents of one register to another. The optimization of register subsumption[8] or coalescing attempts to eliminate register move operations by assigning the same physical register to the source and destination virtual registers of the move operation. However, eliminating a move operation that corresponds to a source level assignment affects debugging. Consider the following source code:

³Since scheduling is performed at the basic block level, we use an offset from the beginning of the basic block instead of an instruction address.

```

y = z+w;    /* S1 */
...
x = y;      /* S2 */

```

Assume for the sake of exposition that code generation has not reordered execution. Assuming that both x and y are promoted to registers, the code selector will generate a move operation for the assignment in S2. Coalescing will assign the same register to both x and y (assuming their live ranges do not conflict), eliminating the move operation generated for S2. In effect, S1 and S2 are performed at the same time, and r will hold the value of both x and y after execution of S1. If a breakpoint occurs somewhere between S1 and S2, S2 will have executed too early. Consequently, x will be a noncurrent variable at such a breakpoint. To model this situation, the IR of S2 is marked as being executed at the same time as S1 so that the noncurrency detection algorithm will detect x as being noncurrent between S1 and S2.

4 Results

We have analyzed the effects of register allocation and instruction scheduling on nine numerical programs. In this section, we report results from two programs selected as representatives of this set: *bessi* (modified Bessel function I of integer order), *gaussj* (Gauss-Jordan elimination), and *ludcmp* (LU decomposition for solving a system of linear equations).

4.1 The effects of register allocation

Figures 3 and 5 illustrate the effects of register allocation on the debugger's ability to recover source variables, by showing the average number of register allocated variables that are nonresident at a breakpoint. The leftmost column shows the average number of register promoted variables, while the other columns show how the number of nonresident variables is reduced by using data flow analysis to find reaching and evicted variables. The second column from the left shows the number of variables that are nonresident if the debugger uses a variable's live range as the range in which a variable is resident. These are the results that would be obtained if the debugger used a simplistic approach to detecting nonresident variables. The third column shows the results of augmenting live range information with reaching analysis to find uninitialized variables. The fourth column shows the number of variables that are nonresident using data flow analysis to find evicted variables, and the fifth shows the effects of using reaching analysis to exclude uninitialized evicted variables.

The results from this figure show that employing data flow analysis techniques in the debugger reduces the number of nonresident variables. Comparing the second column with the fourth, and the third column with the fifth, shows the impact of the eviction data flow analysis, while comparing the second column with the third, and the fourth column with the fifth shows how reaching analysis helps.

4.2 The effects of instruction scheduling

Because of reordered pointer assignments and function calls, the precise number of noncurrent variables cannot be determined. Therefore, the effects of instruction scheduling cannot be analyzed based on the number of noncurrent variables. Instead, the analysis is based on the number of breakpoints that contain noncurrent variables.

Figures 4 and 6 illustrate the effects of instruction scheduling by showing the percentage of breakpoints that have noncurrent variables, and compares this with the percentage of breakpoints that have

nonresident variables. The leftmost column shows the percentage of breakpoints that contain non-current variables. A breakpoint contains noncurrent variables if there are reordered assignments or function calls at the breakpoint. The second column shows the results of using live range information to detect nonresident variables, while third column shows the results of using evicted and reaching data flow analysis to detect nonresident variables.

These results show that as far as breakpoints are concerned, nonresident variables pose much more of a problem to symbolic debugging than noncurrent variables. Results from the other numerical programs that we looked at are consistent with those shown here. The percentage of breakpoints with noncurrent variables ranged from 15-30%, while the percentage of breakpoints with nonresident variables ranged from 65-100% when the simple live range approach is used and 40-95% when the reaching and evicted variables data flow analysis is used. Programs that have an average number of register promoted variables of over 10, have a high percentage of breakpoints with nonresident variables, 95% percent or more for the live range case, and 70% or more using data flow analysis.

5 Concluding remarks

Previous work on debugging optimized code has been mostly concerned with noncurrency due to reordering or elimination of assignments. Our investigations indicate that nonresidency of register promoted variables is a serious problem that must be addressed by a symbolic debugger for optimized code.

The techniques used by the debugger to determine residency significantly impact the ability of the debugger to allow user inspection of variables. If the debugger relies solely on the compiler's view of a variable's live range to determine residency, the debugger misses many opportunities to find variables. Those variables that are dead but still in a register are reported as nonresident. However, if the debugger analyses the object code, it can determine when a variable is evicted, and only evicted variables are reported as nonresident. Register promoted variables are evicted by reuse of their assigned storage locations - either explicitly by an instruction that target the register that holds the value, or implicitly by a function call if the value is kept in a caller-save register.

Acknowledgements

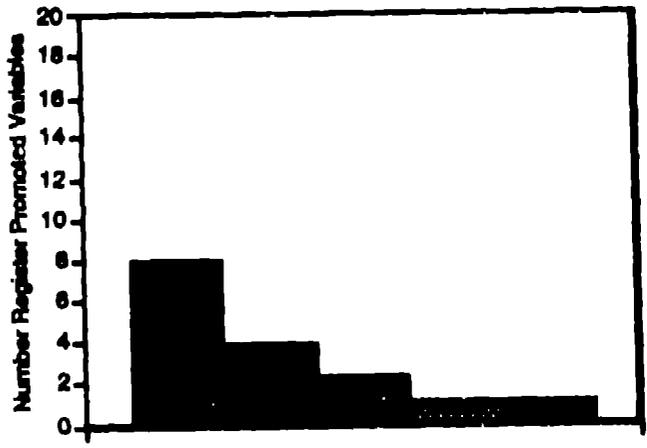
We appreciate the contributions by the Intel iWarp compiler team in providing us with a compiler framework. We especially thank James Reinders for numerous discussions of the compiler internals.

References

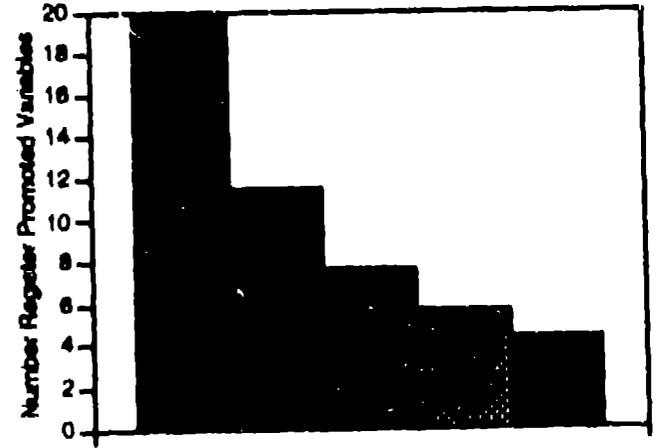
- [1] A. Adl-Tabatabai. Symbolic debugging of optimized C code. Technical report, School of Computer Science, Carnegie Mellon University, 1992.
- [2] A. Adl-Tabatabai and T. Gross. Evicted variables and the interaction of global register allocation and symbolic debugging. In *Proc. 20th POPL Conf. ACM*, January 1993.
- [3] A. V. Aho, R. Sethi, and Ullman J. D. *Compilers*. Addison-Wesley, 1986.
- [4] T. Bemmerl and R. Wismueller. Quellcode debugging von global optimierten programmen. Presented at 1992 Dagstuhl Seminar, Feb. 1992. (In German).

- [5] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iwarp: An integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88*, pages 330–339, Orlando, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.
- [6] G. Brooks, G. Hansen, and S. Simmons. A new approach to debugging optimized code. In *Proc. SIGPLAN'92 Conf. on PLDI*, pages 1–11. ACM SIGPLAN, June 1992.
- [7] B. Bruegge and T. Gross. An integrated environment for development and execution of real-time programs. In *Proc. ACM International Conf. on Supercomputing*, pages 153–162, St. Malo, France, July 1988. ACM.
- [8] G.J. Chaitin. Register allocation and spilling via graph coloring. In *Proc. of the SIGPLAN 1982 Symposium on Compiler Construction*, pages 98–105, 1982. In SIGPLAN Notices, v. 17, n. 6.
- [9] M. Copperman. Debugging optimized code without being misled. Technical Report 92-01, UC Santa Cruz, May 1992.
- [10] D. S. Coutant, S. Meloy, and M. Ruschetta. Doc: A practical approach to source-level debugging of globally optimized code. In *Proc. SIGPLAN 1988 Conf. on PLDI*, pages 125–134. ACM, June 1988.
- [11] J.L. Hennessy. Symbolic debugging of optimized code. *ACM Trans. on Programming Languages and Systems*, 4(3):323 – 344, 1982.
- [12] P. Kessler. Fast breakpoints: Design and implementation. In *Proc. ACM SIGPLAN'90 Conf. on PLDI*, pages 78–84. ACM, June 1990.
- [13] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, 1991.
- [14] D. Wall, A. Srivastava, and F. Templin. A note on Hennessy's 'symbolic debugging of optimized code'. *ACM Trans. on Programming Languages and Systems*, 7(1):176–181, January 1985.
- [15] P. Zellweger. An interactive high-level debugger for control-flow optimized programs. In *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 159–171. ACM, 1983.

- Avg. number of register promoted variables
- Avg nonresident variables, residence determined by live range data flow
- Avg nonresident variables, residence determined by live range and reaching dataflow
- Avg nonresident variables, residence determined by evicted data flow
- Avg nonresident variables, residence determined by evicted and reaching data flow



Bessj



Gaussj

Figure 3: Effect of register allocation

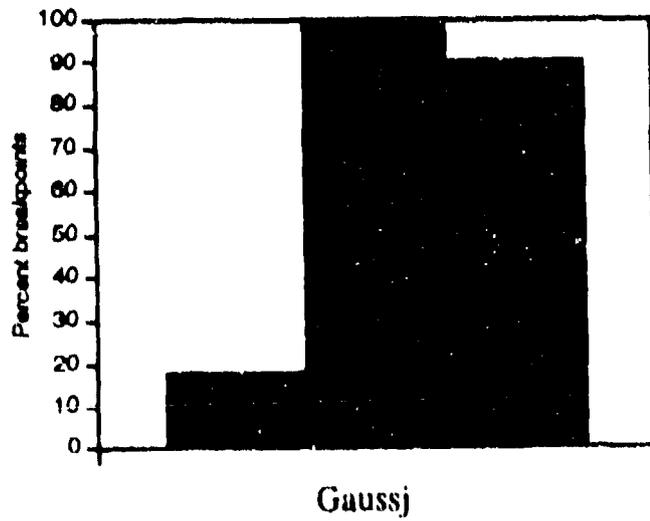
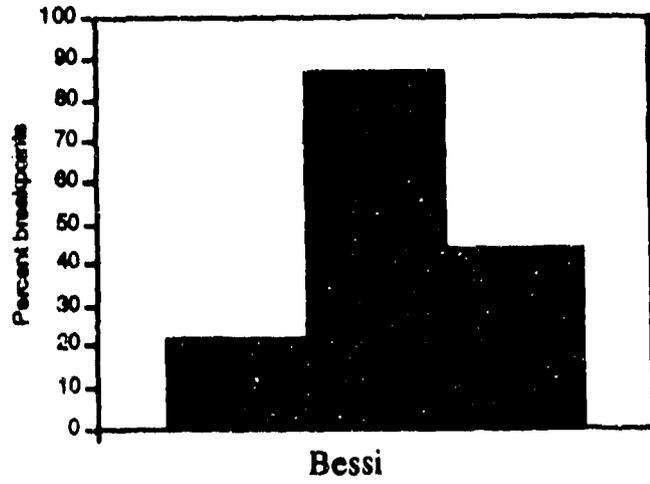
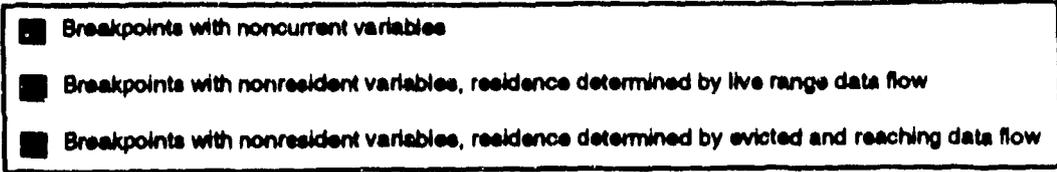


Figure 4: Effect of instruction scheduling

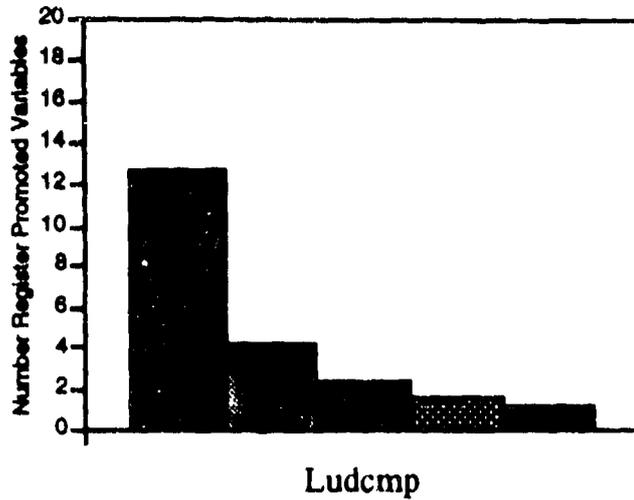
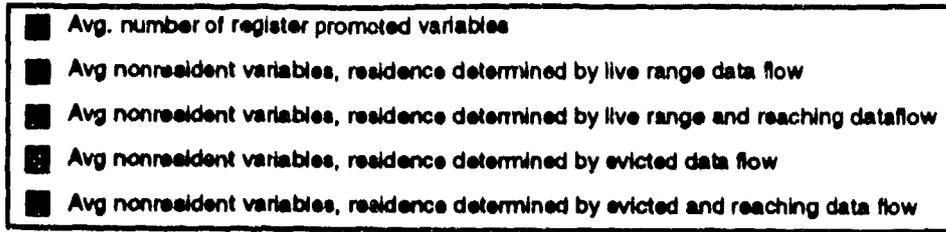


Figure 5: Effect of register allocation

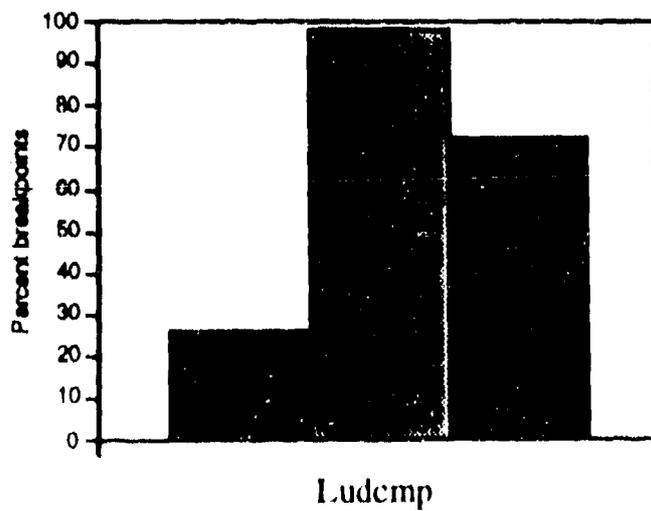
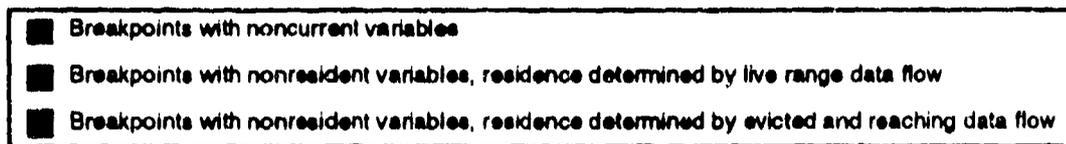


Figure 6: Effect of instruction scheduling

Debugging Optimized Code: Currency Determination with Data Flow

Max Copperman

max@cse.ucsc.edu

Board of Studies in Computer and Information Sciences
University of California at Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

Optimizing compilers produce code that impedes source-level debugging. Optimization can disturb the mapping between source statement boundaries and machine instructions. This paper presents a mapping that enables setting breakpoints at source statements and single stepping at the statement level in optimized code.

Optimization can cause the value of a variable to be *noncurrent* - to differ from the value that would be predicted by simulating the source code. If a debugger does not display a warning when the debugger user asks for the value of a noncurrent variable, the user will be misled. This paper describes a simple dataflow algorithm to determine a variable's currency, and shows how a debugger can use the results to describe the relevant effects of optimization. The determination method is more general than previously published methods.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging - *debugging aids*, D.2.6 [Software Engineering]: Programming Environments; D.3.4 [Programming Languages]: Processors - *code generation, compilers, optimization*

General Terms: Algorithms, Languages

Additional Keywords and Phrases: debugging, compiler optimization, reaching definitions, noncurrent variables

Original Source Code	After Constant Propagation	After Dead-Store Elimination
<code>x = expression;</code>	<code>x = expression;</code>	<code>x = expression;</code>
<code>...</code>	<code>...</code>	<code>...</code>
<code>x = constant;</code>	<code>x = constant;</code>	<code>...</code>
<code>...</code>	<code>...</code>	<code>...</code>
<code>y = x;</code>	<code>y = constant;</code>	<code>y = constant;</code>
<code>...</code>	<code>...</code>	<code>...</code>

Figure 1.1: Potentially Confusing Optimizations: Assume that the only use of `x` after the assignment of `constant` to `x` is the one shown. Constant propagation removes that use (shown in the second column). Subsequently, the assignment of `constant` to `x` may be eliminated as shown in the third column. If the debugger is asked to display the value of `x` anywhere after the eliminated assignment, typical debuggers will display `expression`. The user, looking at the original source code, may wonder why the displayed value is not `constant`, or may wrongly believe that the value being assigned to `y` is `expression`.

1 Introduction

Debugger users can set a breakpoint at a source statement (say, S), and when it is reached, can have the debugger display a variable, say, V . Most debuggers will display whatever is in V 's storage location. Assuming it has not eliminated V , optimization can introduce two problems for this scenario. After optimization, it may not be clear which instruction generated from S (if any) reflects the user's notion of being "at" S . Even if such an instruction is found, the value in V 's storage location may not be the value that the source code would lead one to expect. For example, due to a code motion optimization, an assignment to V may have been done earlier in the generated code than in the source code. This can confuse, mislead, slow down, and irritate debugger users. Figure 1.1 is an example of the latter problem, caused by constant propagation followed by dead store elimination.

Source-level debugging of optimized code is clearly desirable. It would eliminate recompilation steps and enable source-level performance debugging. In particular, it would replace or enhance machine-level debugging of optimized code when the bug causes the behavior of an optimized version of a program to differ from the behavior of an unoptimized version.

Programmers spend a lot of time on such bugs because they are difficult to locate. If the problem is a compiler bug, machine-level debugging may still be necessary, though source-level features may speed the process of narrowing down the search. But if the bug is in the application, it may be possible to find the bug without ever going to the machine level.

It often surprises engineers to find that the behavior of an optimized version of a program can differ from the behavior of an unoptimized version even when the compiler is correct. However, most of us have experienced a program behaving badly, but the bug going away when we run the program in the debugger. One typical cause is an assignment through a stray pointer that hits a

different piece of data because the program is loaded at a different address. Optimization can have the same effect: because the size of the code and data space differs from an unoptimized version, an assignment through a stray pointer can hit a different piece of data.¹

Because optimized code does cause difficulties in mapping between the source code and the machine code, if a debugger provides source-level debugging of optimized code, it should warn the user when its responses to queries may have been affected by optimization.

1.1 Overview

Section 2 describes a mapping between source statements and machine instructions that allows breakpoints to be set at the debugger user's notion of source statement boundaries. When such a breakpoint is reached, if the value in a variable's storage location is suitable to be displayed to the user, the variable is *current*. The remainder of the paper describes how to determine whether a variable is current at a breakpoint – the problem of *currency determination*, investigated by Hennessy [Hen82]. The fundamental idea behind this solution to the currency determination problem is the following: if the definitions of a variable V that “actually” reach a breakpoint B are not the ones that “ought” to reach B , V is not current at B . The definitions of V that actually reach B are those that reach B in the version of the program executing under debugger control. The definitions of V that ought to reach B are those that reach B in a strictly unoptimized version of the program.² Section 4 describes a dataflow computation that produces a set of pairs – the definition of V that ought to reach B along a path p is paired with the definition of V that actually reaches B along p . Given the set of such pairs for V at B (the *paired reaching set for V at B* , or PRS_B^V), V is current at B if for each pair, both positions of the pair are occupied by the same definition.

In order to determine a variable's currency:

1. The compiler must generate a set of debug records relating statements to code addresses; these debug records are ordered in two flow graphs, one representing the program before optimization and the other representing the program after optimization.
2. The flow graphs are used to compute paired reaching sets.
3. A paired reaching set is inspected to determine the currency of the variable.

¹Another typical cause in the use of an uninitialized variable that gets a different initial value in the debugger. A local variable V may get a different initial value in optimized code because some prior local may have been eliminated. This may cause V to be allocated at a different stack offset, and it will get its initial value from a different previous occupant of the stack.

²One compilation of the program is sufficient to provide the information with which to compute both the definitions that ought to reach B and those that actually reach B .

2 Breakpoint Model

In unoptimized code, the instructions generated from a statement are contiguous,³ and code is generated for every statement in the order in which it appears in the source code. Breaking at a statement S corresponds to having executed all “previous” statements, that is, having executed all code that was generated from statements on the path to S , and suspending execution at the first instruction generated from S . At that point, no “subsequent” statements have begun, that is, no code that was generated from any statement on the path from S (including code generated from S itself) has been executed, and the value in each variable’s location matches the value of the variable that would be predicted by a close reading of the source code.

Debugger users expect these characteristics to hold when execution is suspended at a statement boundary. Considerable optimization can take place without compromising these characteristics (for one example, invariant address calculations can be moved to loop pre-headers). Such optimization may be largely ignored by the debugger without impeding source-level debugging.⁴ The user needs to be informed only about optimization that affects source code variables and statement flow-of-control. Telling the user about optimization on compiler temporaries is likely to make the debugging job harder, not easier.⁵

2.1 Syntactic and Semantic Breakpoints

The machine instruction used as the breakpoint location for a statement should be chosen based on the user’s intent. The user may set a breakpoint in a loop to be able to poke around on each iteration. If the statement at which the breakpoint is set were moved out of the loop by optimization, it would be appropriate to set the breakpoint where it used to be. On the other hand, the user may have set the breakpoint to check the values of variables used in an expression in that statement. In that case, if the statement were moved out of the loop, it would be appropriate to set the breakpoint where it ended up, so the values the debugger displays are the actual values used in the expression.

Of course, the debugger does not know the user’s intent. If these situations are to be distinguished, two types of breakpoints are needed. Zellweger [Zel84] introduced the terms *syntactic* and *semantic* breakpoints. The order in which syntactic breakpoints are reached reflects the syntactic order of source statements; the syntactic breakpoint for statement n is never after the syntactic

³Code generated from looping or branching statements is typically not contiguous. However, this lack of contiguity is present in the source code as well as the generated code.

⁴Note that such code motion is relevant to trap location reporting. If an address computation is moved up out of a loop, and the computation traps, the user should be informed that the trap occurred in the statement that the address computation originated in.

⁵There are circumstances in which it is important for the debugger to reveal the effects of optimization at this level of detail, such as allowing the user to track down a code-generation bug. In such circumstances, it is appropriate to shift to machine-level debugging.

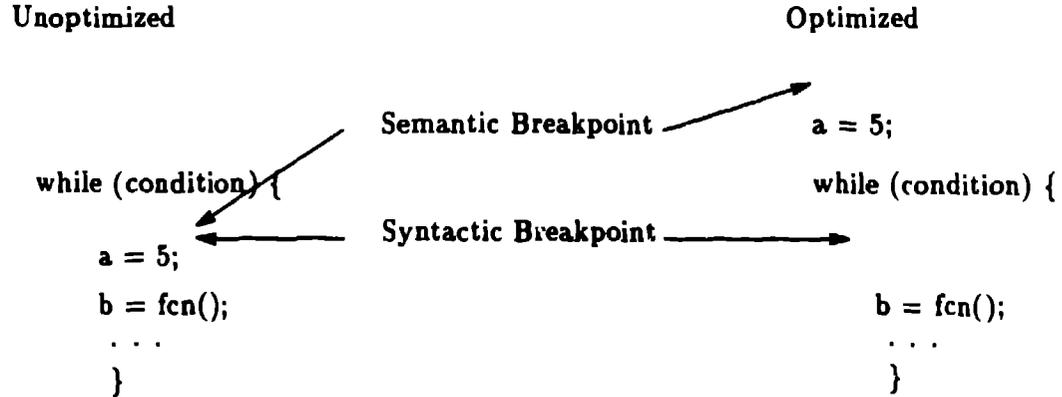


Figure 2.1: Semantic and Syntactic Breakpoint Locations

breakpoint for statement $n + 1$. It will be at the same location if the code for n is moved or eliminated. If the code generated from statement n is moved out of a loop, a syntactic breakpoint for n remains inside the loop. The semantic breakpoint for a statement is where the action specified by the statement takes place; where the code that implements the essence of the statement ends up. If no code motion or elimination has occurred, syntactic and semantic breakpoints are the same.

Figure 2.1 provides an example of the syntactic and semantic breakpoints for a loop from which optimization has moved an invariant statement.

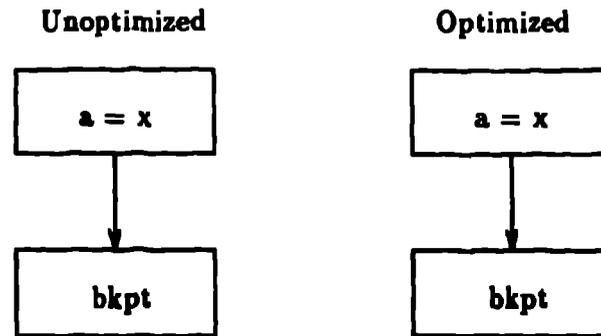
The proposed breakpoint model supports both syntactic and semantic breakpoints. Section 4 assumes only syntactic breakpoints are available.

2.2 Breakpoint Locations (Representative Instructions)

The instructions generated from a statement that are possible breakpoint locations for that statement are the statement's *representative instructions*. The first instruction generated from a statement that has an effect that is visible at the source level is an instruction at which the user may want to break, and thus is selected as a representative statement. If a statement has multiple effects that are visible at the source level, it will have one representative instruction for each. For an assignment, the representative instruction is the instruction that accomplishes the store of the result into the variable (whether it is a store instruction or a computation into a register). Choosing the store as the representative instruction for variable modifications is crucial to the correctness of the work presented in the remainder of the paper. For loops and branches, the branch instruction is the representative instruction.

The C statement `if ((i = j++) == k)` has three representative instructions (and therefore three possible breakpoint locations), one at the store into `j`, one at the store into `i`, and one at the branch to the `then` or `else` case.

For the duration of this paper, the term breakpoint refers to a source-level breakpoint, that is, the location of a representative instruction.

Figure 3.1: Variable *a* is current at *bkpt*

3 Currency

If the debugger user asks the debugger to display a value that optimization has caused to be different from the value that would be displayed at the same point in unoptimized code, the debugger should warn the user.

I call the value in a variable *V*'s storage location when execution is suspended at a breakpoint its *actual value*. *V*'s *expected value* at a breakpoint is the value that would be predicted by hand-simulating the program to the breakpoint.

In unoptimized code, at each breakpoint the expected value of every variable is identical to its actual value, but this is not the case for optimized code. Hennessy [Her82] introduced the terms *current*, *noncurrent*, and *endangered*, which describe the relationship between a variable's actual value and its expected value at a breakpoint based on a static analysis of the program.

Informally, a variable *V* is *current* at a breakpoint *B* if its actual value at *B* is guaranteed to be the same as its expected value at *B* no matter what path was taken to *B*. Examples of current variables are given in Figures 3.1 and 3.2. In the examples, *bkpt* represents the breakpoint.

V is *noncurrent* at *B* if its actual value at *B* may differ from its expected value at *B* along every path to *B* (though the two values may happen to be the same on some particular input). Figures 3.3 and 3.4 show examples of noncurrent variables.

V is *endangered* at *B* if there is at least one path to *B* along which *V*'s actual value at *B* may differ from its expected value at *B*. Figure 3.5 is an example of an endangered variable.

In Figure 3.5, *a* is said to be current along the left-hand path and noncurrent along the right-hand path.

This work builds on previous work that defined a vocabulary for discussing the problem. The definitions and discussion in the remainder of this section are largely as taken from [Cop92].

Because optimization may modify the program's flow graph, *path* must be defined in such a way that it makes sense in both the unoptimized and optimized versions of the program. There are two relevant relationships: the relationship between the optimized and unoptimized flow graphs

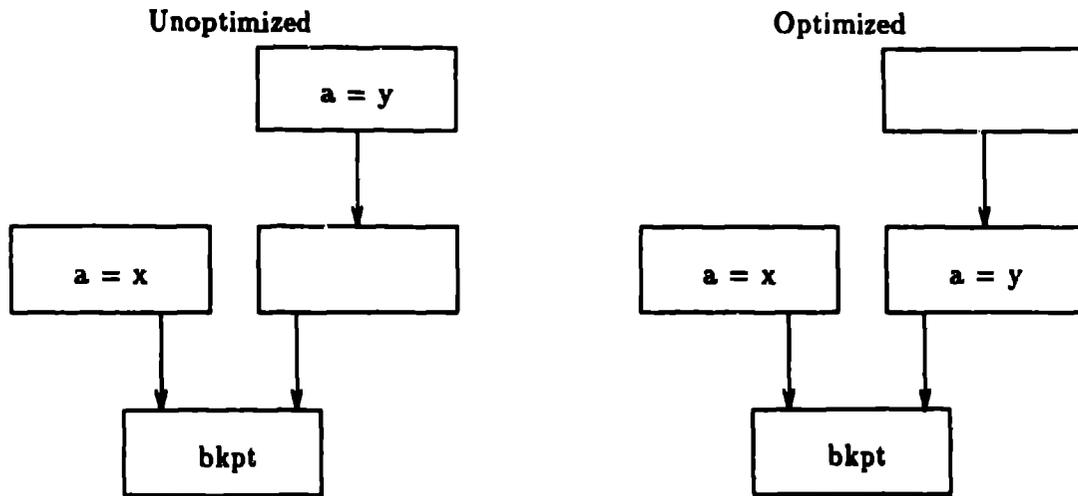


Figure 3.2: Variable **a** is current at **bkpt** in the presence of relevant optimization

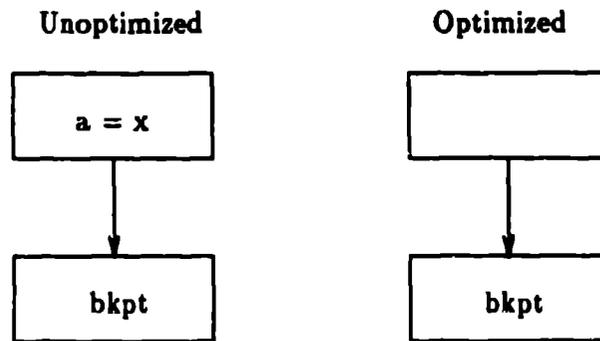


Figure 3.3: Variable **a** is noncurrent at **bkpt**

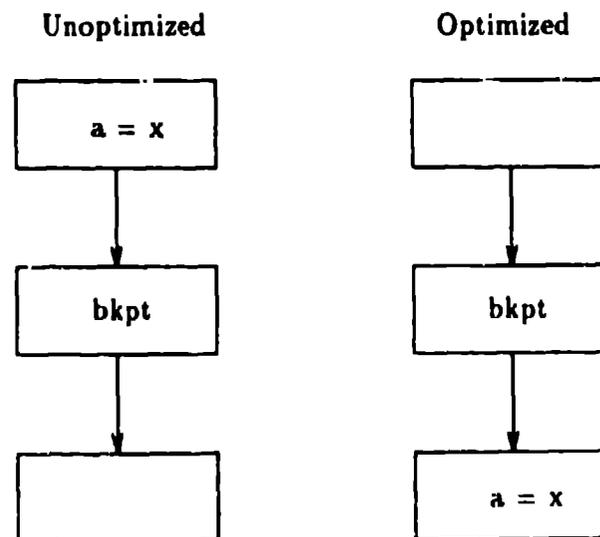


Figure 3.4: Variable **a** is noncurrent at **bkpt** due to code motion

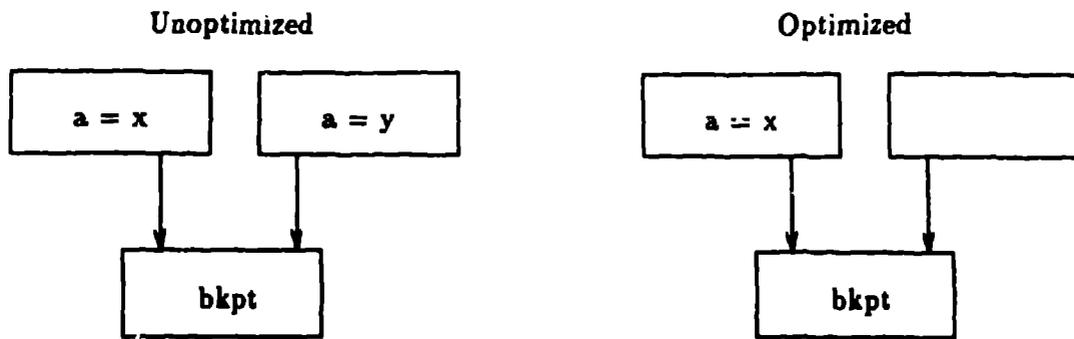


Figure 3.5: Variable a is endangered at $bkpt$

(as would be induced from generated code), and the relationship between the data structures used to determine a variable's currency; the pre-optimization flow graph and the post-optimization flow graph.

Definition 1: A *path* p is a pair $\langle p_u, p_o \rangle$ where p_u is the sequence of basic blocks visited in an execution of an unoptimized version of a program and p_o is the sequence of logical blocks visited in an execution of an optimized version of the same code on the same inputs.

The correspondence between basic blocks in p_u and logical blocks in p_o is as follows:

1. All of the code in block b_u in p_u may have been moved or eliminated by optimization. In that case b_u corresponds to block b_o in p_o , where b_o is an empty block that has been left in the post-optimization flow graph precisely to maintain the correspondence.
2. A basic block introduced by optimization that has a single successor (such as a loop pre-header), together with its successor forms a single logical block. If x is such a block and y is its successor, logical block y_o in p_o denotes both x and y , and corresponds to y_u in p_u .⁶
3. There may be one block b_o in p_o corresponding to a sequence of blocks in p_u , on condition that if the first block in the sequence in p_u is entered, execution will always proceed through the entire sequence. In this circumstance, the single block b_o is treated as a sequence of logical blocks corresponding to the sequence in p_u .
4. Multiple blocks b_1, b_2, \dots, b_n in p_o (not necessarily contiguous) may correspond to multiple instances of a single block b in p_u , on condition that one of the b_i is in p_o

⁶Since the Supercomputer Debugging Workshop, this correspondence has been found in some cases to introduce a conservative inaccuracy into the currency determination algorithm. A variable will be claimed to be endangered between the top of a loop and an assignment into the variable within the loop when it is in fact current in that region, if an assignment into that variable is moved down into the loop pre-header from above. No such inaccuracy occurs if an assignment is moved from the body of the loop to the pre-header.

iff b is at the same point in the sequence p_u .⁷

5. A block b_u in p_u has one corresponding block b_o in p_o otherwise.

These correspondences may be combined, so for example, blocks in an unrolled loop may be coalesced. Optimizations that modify the flow graph in other ways are not handled by the algorithm presented herein.

Both assignments to a variable and side effects on that variable modify the value stored in that variable's location. These terms do not distinguish whether the source code or generated code is under discussion. Furthermore, they do not distinguish between unoptimized generated code and optimized generated code. These distinctions are needed in this work because it compares reaching definitions computed on unoptimized code with reaching definitions computed on optimized code.

Henceforth the term *assignment* refers to assignments and side effects in the source code. It is convenient to have a term *definition* that can denote either an assignment or its representative instruction in unoptimized code. This does not introduce ambiguity because either one identifies the other, and the order of occurrence is the same in the source code and unoptimized code generated from it. In contrast, the term *store* denotes a representative instruction for an assignment in optimized code. As with definitions, an assignment corresponds to a store, but unlike definitions, the order of occurrence of assignments in the source code may differ from the order of occurrence of stores in the machine code.

An optimizing compiler may be able to determine that two assignments to a variable are equivalent and produce a single instance of generated code for the two of them, or it may generate multiple instances of generated code from a single assignment. Such optimizations essentially make equivalent definitions (or stores) indistinguishable from one another. We will be concerned with determining whether a store that reaches a breakpoint was generated from a definition that reaches the breakpoint. If definitions d and d' are equivalent, and store s was generated from d while s' was generated from d' , the compiler is free to eliminate s' so long as s reaches all uses of d' . To account for this, s needs to be treated as if it was generated from either d or d' .

Definition 2: A *definition of V* is an equivalence class of assignments to V occurring in the source code of a program that have been determined by a compiler to represent the same or equivalent computations, or the representative instruction generated from any member of such an equivalence class in an unoptimized version of the program.

⁷Note that while this is the correspondence needed for loop unrolling and inlining (procedure integration), the work as presented in this paper does not handle either of these optimizations. Section 7 describes limitations on the optimizations that are handled.

Definition 3: A *store into V* is the set of representative instructions occurring in an optimized version of a program that were generated from any member of the equivalence class denoted by a definition.⁸

We can now formally define some of the terms described previously.

Definition 4: A *variable V is current at a breakpoint B along path p* iff the store into V that reaches B along p_o was generated from the definition of V that reaches B along p_u .

Definition 5: V *is noncurrent at B along p* iff the store into V that reaches B along p_o was not generated from the definition of V that reaches B along p_u .

Definition 6: V *is current at B* iff V is current at B along each path to B .

Definition 7: V *is noncurrent at B* iff V is noncurrent at B along each path to B .

Definition 8: V *is endangered at B* iff V is noncurrent at B along at least one path to B .

3.1 Assignments Through Aliases

Definitions 4 through 8 assume a single definition or store reaches a breakpoint along any path. Consider an assignment $*P$ through a pointer. When execution is suspended at a breakpoint B , $*P$ may be an alias for V . $*P$ must be considered to be a definition of V that reaches B . If $*P$ is not an alias for V in some particular execution, the value that V contains at the breakpoint came from whatever definition would have reached if $*P$ were not present. Therefore, this definition must also be considered to reach B . For any language that allows aliasing, a static analysis cannot assume that a single definition reaches along a given path.

The presence of multiple definitions or stores along a single path requires more complex versions of Definitions 4, 5 and 8. For clarity of exposition, I have chosen not to cover aliasing in this paper. The required definitions may be found in [Cop92].

⁸A store is an equivalence class by the same equivalence relation applied to definitions (having been determined by a compiler to represent the same or equivalent computations)

4 Paired Reaching Sets

A paired reaching set PRS is a set of reaching definitions that includes information about what should reach and what does reach a given breakpoint. Such a set is relative to both a variable and a breakpoint: PRS_B^V is the paired reaching set of assignments to V that do/should reach B .

An element of a paired reaching set is a pair (d, s) where d is a definition and s is a store. Loosely, for a definition d of V and a store s into V , the pair $(d, s) \in \text{PRS}_B^V$ means d should reach B and s does reach B .

More precisely, given such a definition d and a store s , independent of whether s was generated from d :

$(d, s) \in \text{PRS}_B^V$ means there is a path p such that d reaches B along p_d and s reaches B along p_s .

- V is current at B iff $\forall (d, s) \in \text{PRS}_B^V, s$ was generated from d .
- V is endangered at B iff $\exists (d, s) \in \text{PRS}_B^V$ such that s was not generated from d .
- V is noncurrent at B iff $\forall (d, s) \in \text{PRS}_B^V, s$ was not generated from d .

Because I want to use familiar notation for familiar tasks, I will allow 'dotting into' a pair: if P is the pair (x, y) , $P.d$ is the definition element x and $P.s$ is the store element y . $(d, s).d = d$ and $(d, s).s = s$.

Paired reaching sets can be computed by the compiler or the debugger. If the compiler computes them, existing compiler data structures can be modified for the task. Previous work has described the data structures the compiler must provide the debugger if the debugger computes them ([Cop92], [CM91a], [Cop90]). The computation requires a pre-optimization flow graph giving control flow information and the order of representative instructions (definitions) as it exists prior to optimization, and a post-optimization flow graph giving control flow information and the order of representative instructions (stores) as it exists after optimization. The correspondence between basic blocks in the pre-optimization flow graph and logical blocks in the post-optimization flow graph must be available. The correspondence between definitions and the stores generated from them must be available.

In this computation, definition/store pairs will be the elements of Gen, In, and Out sets for each basic block. The Gen set Gen_B^V for a variable V and block B contains a single pair⁹ consisting of the definition of V occurring in B and the store into V occurring in B . If no definition of V occurs in B , the definition element of the pair is null. If no store into V occurs in B , the store element of the pair is null. An initial definition of V and store into V are associated with the source node of a flow graph component. Null entries do not appear in the In and Out sets of a block because some non null definition or store reaches every block.

Because I want to introduce unfamiliar notation for unfamiliar tasks, operation κ defines the interaction of a pair that reaches a block with the pair generated in that block. Given two pairs

⁹In the presence of aliasing a Gen set may contain multiple pairs.

(d, s) and (e, t) where d and e are definitions of a variable V , s and t are stores into V , and d or s may be null, Table 4.1 defines $(e, t) \kappa (d, s)$.

$(e, t) \kappa (d, s)$	d is null	d is non-null
s is null	(e, t)	(d, t)
s is non-null	(e, s)	(d, s)

Table 4.1: The definition of κ : $(e, t) \kappa (d, s)$

The κ operation corresponds to the kill operation in standard dataflow algorithms. Definitions kill definitions, and stores kill stores. If a block contains a store, its Gen pair contains that (non-null) store as its second element, and pairs leaving that block contain that store as their second element. If a block does not contain a store, its Gen pair contains a null second element, and pairs leaving that block contain the second element they arrived at that block with. The same goes for definitions. To make this appear similar to a familiar dataflow operation (which it is), the κ operation has been extended to a set of pairs (the In set) and a pair. The In set may be empty (in fact, every In set is initially empty), in which case if the Gen pair contains a null, the result is empty, otherwise the result is the set containing the Gen pair:

$$\emptyset \tilde{\cup} (\text{null}, \text{null}) = \emptyset$$

$$\emptyset \tilde{\cup} (d, \text{null}) = \emptyset$$

$$\emptyset \tilde{\cup} (\text{null}, s) = \emptyset$$

$\emptyset \tilde{\cup} (d, s) = (d, s)$. This is correct because the $\tilde{\cup}$ operation is used to define the Out set of a block. If the block generates both a store and a definition, its Out set will contain the pair consisting of that store and definition. If it contains a null in either position (or both), its Out set depends on the In set. Because every variable is defined to have an initial definition and store, propagation will eventually cause the block to have a nonempty In set.¹⁰ Given a nonempty set of pairs R and a pair S where S may contain nulls, $R \tilde{\cup} S$ is equivalent to the set of pairs produced by individual κ operations between each pair in R and the pair S : $R \tilde{\cup} S \equiv \{r \kappa S | r \in R\}$.

Algorithm PRS computes paired reaching sets at block boundaries for a flow graph component (a subroutine). *Start* is the start node of the flow graph component. *d-init* and *s-init* are initial definitions and stores representing the creation of a variable.

Algorithm PRS

Input:

the post optimization flow graph;

the pre optimization flow graph;

Output:

¹⁰ This could make the algorithm converge more slowly than a standard reaching definitions algorithm

The paired reaching sets of each variable at each block boundary.

Step 1:

```

0  for each variable  $V$ 
1     $\text{Gen}_{\text{Start}}^V = (d\text{-init}, s\text{-init})$ 
2    for each source block  $B$ 
3       $\text{Gen}_B^V = (\text{null}, \text{null})$ 
4      if a definition  $d$  of  $V$  is in  $B$  and reaches the exit of  $B$  in the pre-optimization flow graph
5         $\text{Gen}_B^V.d = d$ 
6      if a store  $s$  into  $V$  is in  $B$  and reaches the exit of  $B$  in the post-optimization flow graph,
7         $\text{Gen}_B^V.s = s$ 

```

Step 2:

```

8  for each variable  $V$ 
9    for blocks  $B$  that can be reached in the post-optimization flow graph,
10      $\text{In}_B^V = \text{Out}_B^V = \emptyset$ 
11     iteratively compute  $\text{In}_B^V$  and  $\text{Out}_B^V$  until convergence, according to the following,
12      $\text{In}_B^V = \bigcup_P \text{Out}_P^V$  for  $P$  logical predecessors of  $B$  in the post-optimization flow graph
13      $\text{Out}_B^V = \text{In}_B^V \dot{\cup} \text{Gen}_B^V$ 

```

End of Algorithm PRS

For ease of exposition, we assume that there is at most one definition of a variable in a block. A basic block B that contains n definitions of V can be transformed into a sequence of blocks B_1, B_2, \dots, B_n each containing a single definition of V , where B_i is the sole predecessor of B_{i+1} and B_{i+1} is the sole successor of B_i . Similarly, we assume that there is only one store into V in a block.

Algorithm PRS provides In and Out sets at block boundaries. Our goal is to determine a variable's currency at an arbitrary breakpoint Bp . Let B be the block containing Bp , and let L be the location of the definition of V or store into V in B , or *null* if there is none.

if $L \neq \text{null}$ and (L precedes Bp in B or $L = Bp$)

$$\text{PRS}_{Bp}^V = \text{In}_B^V$$

else

$$\text{PRS}_{Bp}^V = \text{Out}_B^V$$

V 's currency at Bp is determined by comparing the definition element with the store element in the pairs in PRS_{Bp}^V . If in every pair, the store was generated from the definition, V is current. If the store was not generated from the definition in any pair, V is noncurrent. If the store was generated from the definition in some pairs but not in others, V is endangered.

4.1 Aliasing

Algorithm PRS assumes that a single definition of a variable reaches a breakpoint along a path. As discussed in Section 3.1, this is not the case in the presence of aliasing.

Paired reaching sets can be constructed in the presence of aliasing. However, the proof of correctness of the algorithm was not complete at press time, so the material has been omitted.

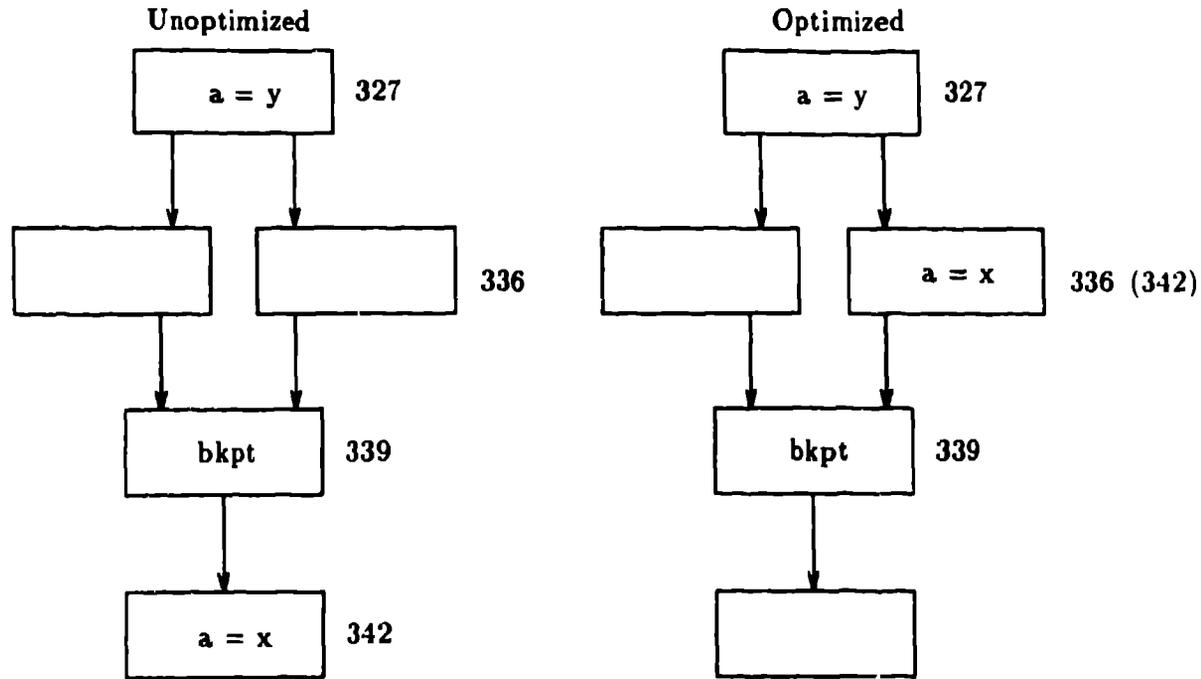


Figure 5.1: The display of `a` could be accompanied by this message: “Breakpoint 1 has been reached at line 339. `a` should have been set at line 327. However, optimization has moved the assignment to `a` at line 342 to near line 336. `a` was actually set at one of lines 327 or 342.”

5 When a Variable is Endangered

When the debugger is asked to display a variable, it determines whether the variable is current. If the variable is current, the debugger displays its value without comment. If the variable is endangered, in addition to displaying its value, the debugger can give the user some help in understanding why the value is endangered. The general flavor of what the debugger can do is given by the following sample message that might accompany the display of variable `a` when the optimization shown in Figure 5.1 has occurred.

“Breakpoint 1 has been reached at line 339. `a` should have been set at line 327. However, optimization has moved the assignment to `a` at line 342 to near line 336. `a` was actually set at one of lines 327 or 342.”

The information contained in this message is available from the paired reaching set PRS_{339}^a and the pre- and post-optimization flow graphs. The description of the effects of optimization will vary in specificity as the effects of optimization vary in complexity.

6 Running Time

The worst-case asymptotic cost of Algorithm PRS is dreadful, though polynomial. Let n be the number of blocks in the flow graph and m be the number of definitions of the variable in question. Recall that the blocks may be split according to Sections 4 (n is the number of blocks after splitting). The asymptotic worst case cost is $O(n^3m^2)$. If $n = m$, it is an $O(n^5)$ algorithm. However, we will see that in practice two factors of n and a factor of m can be replaced with constant factors, for an $O(nm)$ running time.

Computing paired reaching sets is done with an iterative algorithm that runs until it converges. The equations are

$$\begin{aligned} \text{In}_B^V &= \bigcup_P \text{Out}_P^V \text{ for } P \rightarrow B \\ \text{Out}_B^V &= \text{In}_B^V \cup \text{Gen}_B^V \end{aligned}$$

and these are computed iteratively over all blocks B until no In or Out set changes.

We are concerned with definitions of a single variable.

Within each iteration the computation of Out_B is cheaper than the computation of In_B . Computing In_B involves iterating over the (up to n) predecessors of B , and In_B is computed for each of n blocks, so the union operation is performed n^2 times. The union operation is a merging of sets containing at most m elements, which can be done in time proportional to m , so each iteration has worst case cost of n^2m .

In the worst case, each iteration could add one definition to one block, so the total number of iterations could be nm , for an $O(n^3m^2)$ total worst case running time.

If the graph is traversed in the right order, on average 5 iterations are sufficient for convergence [ASU86], replacing factors of n m with a factor of 5.

Two factors of n come from iterating over n blocks with n predecessors each. In practice, a fully connected flow graph is a rarity. Most blocks have one or two predecessors, though some have many (e.g., the block following a case or switch statement). The number of predecessors is some small constant c which gives us an $O(nm)$ running time.

I expect m to be fairly small on average, though the sizes of both m and n depend considerably on program characteristics (and coding style). In particular, because each subroutine is a flow-graph component, the cost increases with the size of the subroutines. Without a working implementation, I cannot yet say whether we will achieve speeds acceptable for interactive use. However, I take comfort in the fact that machine speeds double regularly.

7 Summary

In optimized code, statements may be reordered and the instructions generated from a statement may not be contiguous in the final executable code. If the statement to breakpoint location mapping commonly used for statements in unoptimized code is used for optimized code, a debugger user cannot in general break at what the user considers a statement boundary, or execute a single statement at a time. Section 2 describes a mapping between statements and breakpoints for optimized code that provides a reasonable approximation to what the naive user would expect. It provides exactly what the naive user would expect on unoptimized code. In optimized code, it isolates points that correspond well to the user's view of statement boundaries, and provides a granularity of breakpoint locations fine enough that the user can 'step' without executing more than a single statement. If a statement does not have multiple side effects, one 'step' executes the entire statement.

Optimization can cause the value in a variable's location to be endangered, which means it is unexpected and potentially misleading. A debugger must be able to determine a variable's currency if it is to issue a warning when asked to display an endangered variable. Hennessy [Hen82] [CM91b] and Coutant et al [CMR88] give solutions to special cases of the currency determination problem. Section 4 describes a general solution to the problem for a large class of local and global sequential optimizations, including common subexpression elimination, cross-jumping, instruction scheduling, other code motion, partial redundancy elimination, loop reordering, induction-variable elimination, and loop fusion.

The currency determination algorithm requires reaching-definitions information computed before and after optimization, but does not require knowledge of which of these optimizations have been performed. In addition, this reaching-definitions information allows the debugger to construct informative warnings as to why a variable is endangered.

The results described in this paper are conservative when a variable is current along all feasible paths but noncurrent along some infeasible path, in which case it will be claimed to be endangered.¹¹

There are important sequential optimizations that do not fall into the class delineated by Definitions 1, 2 and 3. These are optimizations that duplicate code in such a manner that a duplicate does not perform an equivalent computation to the original (as in loop unrolling and inlining). This work can be extended to handle this class of sequential optimizations, but the extensions are beyond the scope of this paper. Parallelizing optimizations have not been considered.

7.1 Future Work

Once a debugger user has found a suspicious variable (one that due to program logic, not optimization, contains an unexpected value), the next question is 'How did it get that value?'.

¹¹An infeasible path is one that cannot be taken in any execution

The sets of reaching definitions used for currency determination can be used in a straightforward manner to answer this question ('x was set at one of lines 323 or 351'). One direction for future research is whether reaching sets are adaptable to back-chaining such dependences efficiently. This has been called *flowback analysis* by Balzer [Bal69], and has been investigated by others ([MC91], [Kor88]).

Another research avenue is how a debugger can efficiently collect the runtime information needed to determine whether an endangered variable is in fact current or noncurrent on a particular execution. In conjunction or as an alternative, how can the information from the compiler be extended so that the debugger can compute and display the value that a variable would have had if optimization had not been performed? Finally, an exciting possibility is extending the breakpoint model and currency determination techniques to parallel code, which is rife with noncurrent variables.

References

- [AU77] A. V. Aho, J. D. Ullman, "Principles of Compiler Design," Addison-Wesley, Menlo Park, CA, 1977.
- [ASU86] A. V. Aho, R. Sethi, J. D. Ullman, "Compilers Principles, Techniques, and Tools," Addison-Wesley, Menlo Park, CA, 1986.
- [Bal69] R. M. Balzer, "EXDAMS - EXTendable Debugging and Monitoring System," *Proceedings of AFIPS Spring Joint Computer Conference*, Vol 34 pp. 125-134, 1969.
- [Cop92] M. Copperman, "Debugging Optimized Code Without Being Misled," UCSC Technical Report UCSC-CRL-92-01, January 1992. Submitted for publication to *ACM Transactions on Programming Languages and Systems*.
- [Cop90] M. Copperman, "Source-Level Debugging of Optimized Code: Detecting Unexpected Data Values," University of California, Santa Cruz technical report UCSC-CRL-90-23, May 1990.
- [CM91a] M. Copperman, C. E. McDowell, "Debugging Optimized Code Without Surprises," *Proceedings of the Supercomputer Debugging Workshop*, Albuquerque, November 1991.
- [CM91b] M. Copperman, C. E. McDowell, "A Further Note on Hennessy's "Symbolic Debugging of Optimized Code", UCSC Technical Report UCSC-CRL-91-04, February 1991. Submitted for publication to *ACM Transactions on Programming Languages and Systems*
- [CMR88] D. Coutant, S. Meloy, M. Ruscetta "DOC: a Practical Approach to Source-Level Debugging of Globally Optimized Code," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 125-134, 1988.
- [FM80] P. H. Feiler, R. Medina-Mora, "An Incremental Programming Environment," Carnegie Mellon University Computer Science Department Report, April 1980.
- [Hen82] J. Hennessy, "Symbolic Debugging of Optimized Code," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 323-344, 1982.
- [Kor88] B. Korel, "PELAS Program Error-Locating Assistant System," *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, pp. 1253-1260, September 1988.
- [MC'88] B. Miller, J. Choi, "A Mechanism for Efficient Debugging of Parallel Programs," *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 125-134, Madison, Wisconsin, 1988.
- [MC'91] B. Miller, J. Choi, "Techniques for Debugging Parallel Programs with Flowback Analysis," *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pp. 491-530, 1991.
- [PS88] L. L. Pollack, M. L. Soffa, "High Level Debugging with the Aid of an Incremental Optimizer," *Hawaii International Conference on System Sciences*, January 1988.
- [PS92] L. L. Pollack, M. L. Soffa, "Incremental Global Reoptimization of Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 14, No. 2, pp. 173-200, 1992.

- [Str91] L. Streepy, "CXdb A New View On Optimization," *Proceedings of the Supercomputer Debugging Workshop*, Albuquerque, November 1991.
- [WS78] H. S. Warren, Jr., H. P. Schlaeppli, "Design of the FDS interactive debugging system," IBM Research Report RC7214, IBM Yorktown Heights, July 1978.
- [Ze83a] P. Zellweger, "Interactive Source-Level Debugging of Optimized Programs," *Research Report CSL-83-1*, Xerox Palo Alto Research Center, Palo Alto, CA, Jan. 1983.
- [Ze83b] P. Zellweger, "An Interactive High-Level Debugger for Control-Flow Optimized Programs," *SIGPLAN Notices*, Vol. 18, No. 8, pp. 159-172 Aug. 1983.
- [Zel84] P. Zellweger, "Interactive Source-Level Debugging of Optimized Programs," Research Report CSL-84-5, Xerox Palo Alto Research Center, Palo Alto, CA, May 1984.
- [ZJ90] L. W. Zurawski, R. E. Johnson, "Debugging Optimized Code With Expected Behavior," Unpublished draft from University of Illinois at Urbana-Champaign Department of Computer Science, August 1990.

A Debugging Tool for Parallel and Distributed Programs

Andreas Weininger
Institut für Informatik
Technische Universität München
Orleansstr. 34
W-8000 München 80
weininge@informatik.tu-muenchen.de

1 Introduction

Debugging parallel and distributed programs is much more difficult than debugging sequential programs. The main reasons for this fact are the missing reproducibility of parallel programs, the added complexity because of more threads of control, and the great importance of the probe effect [Gai86] because of time dependencies in parallel programs. This paper shows an example how the problems mentioned above can be handled in a debugger for parallel and distributed programs. This example is the debugger *Source* for the parallel and distributed programming language ParMod.

ParMod [Eic86] is a set of language constructs for parallel programming. These constructs have been combined with different sequential programming languages for instance Pascal [Eic87] or C [WSF91]. Currently a new version based on Modula2 is developed. A ParMod program consists of several modules. A module may contain global and local parallel procedures. Tasks are created by asynchronous calls of parallel procedures. Tasks within a module may communicate through shared variables whereas different modules may only communicate through parameters passed to global procedures i.e. there is no shared memory between different modules. Figure 1 shows an example of a snapshot of a ParMod program run. Most imperative parallel programming languages are at least partially similar to these aspects of ParMod. Therefore most of the following considerations can be generalized to other parallel programming systems.

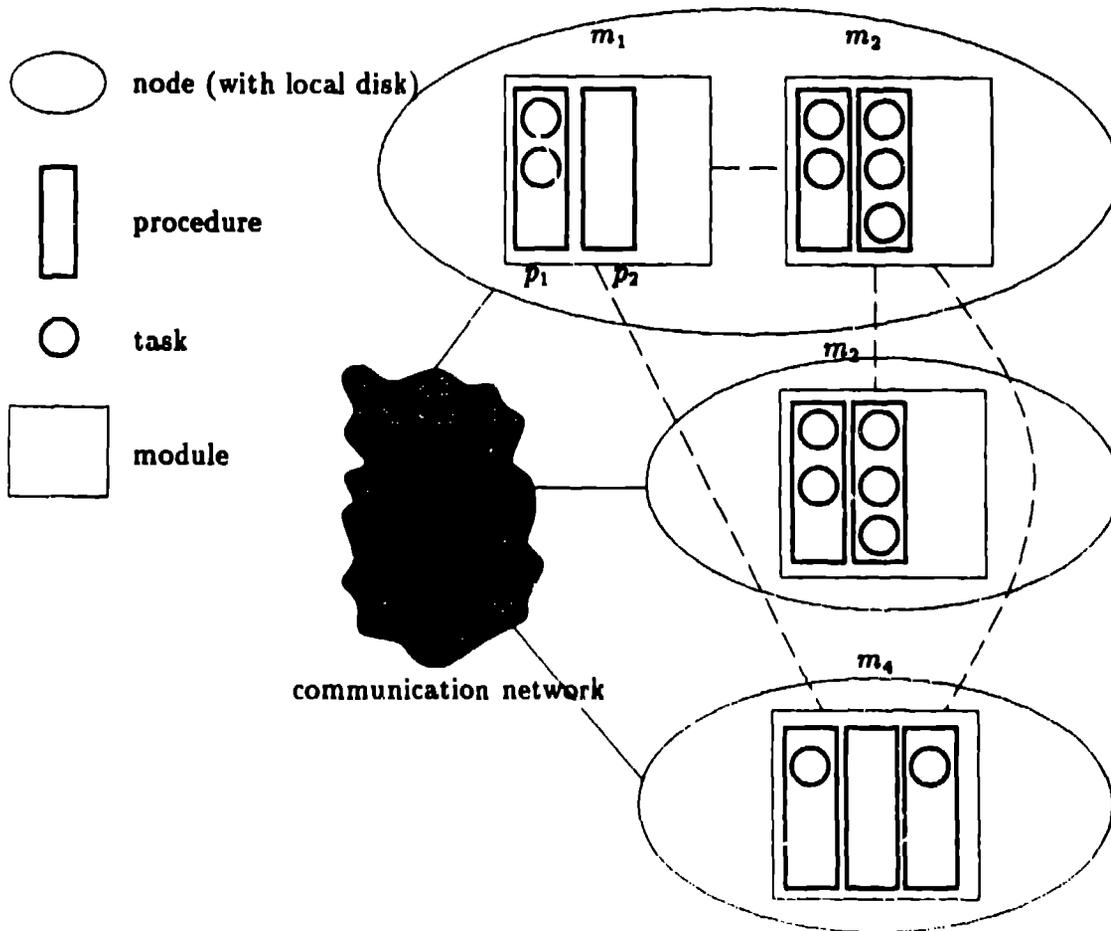


Figure 1: Example of a snapshot of a ParMod program run

To deal with the complexity of programming parallel systems, we have added a programming environment to ParMod. This environment includes the tools *Runtime* for performance analysis [EA88] [AFP90] and *Source* for debugging. The first version of the debugger *Source* was implemented for ParMod-Pascal [Wei88]. The ParMod-Pascal system is a simulator which allows unrestricted parallelism [FZ89]. Later *Source* was extended such that it can be used with the distributed implementation of ParMod-C on a network of UNIX workstation [Mäs91]. The following discussion is based on this version of *Source*.

2 Design Goals

Several goals guided the design of the debugger *Source*: First, we wanted to provide information on a high level of abstraction, the source code level of the ParMod language. The debugger should have the same graphical user interface for all ParMod implementations. *Source* should allow the reproduction of program runs. Therefore we have chosen a trace based approach. *Source* should provide a high degree of portability. This means

- as much hardware independence as possible. Therefore *Source* is a pure software solution.
- independence of any special dialect of ParMod. Even non-ParMod programs can be observed if they can generate the necessary trace information¹.
- communication and integration with other tools of the ParMod development environment especially with the performance visualization and analysis tool *Runtime*.

Another goal was to minimize the probe effect even if this is difficult to achieve for a pure software approach. A subgoal of this was to minimize the length of the trace which yielded to the principle: "What is expensive to trace should be expensive to specify for the user". Therefore most inter-task events are traced by default whereas most of the intra-task events, which should normally occur much more often, must be enabled explicitly by the user. The following sections show how we achieved our goals.

3 Architecture

Because of the reasons mentioned above, *Source* is a trace based debugger. But *Source* is a combined post-mortem and online debugger, i.e. it is possible to analyze a program during the program run and also after the program run. Figure 2 gives an overview of the architecture of *Source*, when it is used as an online debugger for a ParMod-C program running on a workstation network. Every ParMod module writes a local trace file on its node. This file is written locally to avoid central bottlenecks (cpu time, disk space). Another process reads the local file and sends the trace data via UNIX sockets to the

¹For instance the ANSA Testbench [ANS90] is currently extended to produce ParMod trace files

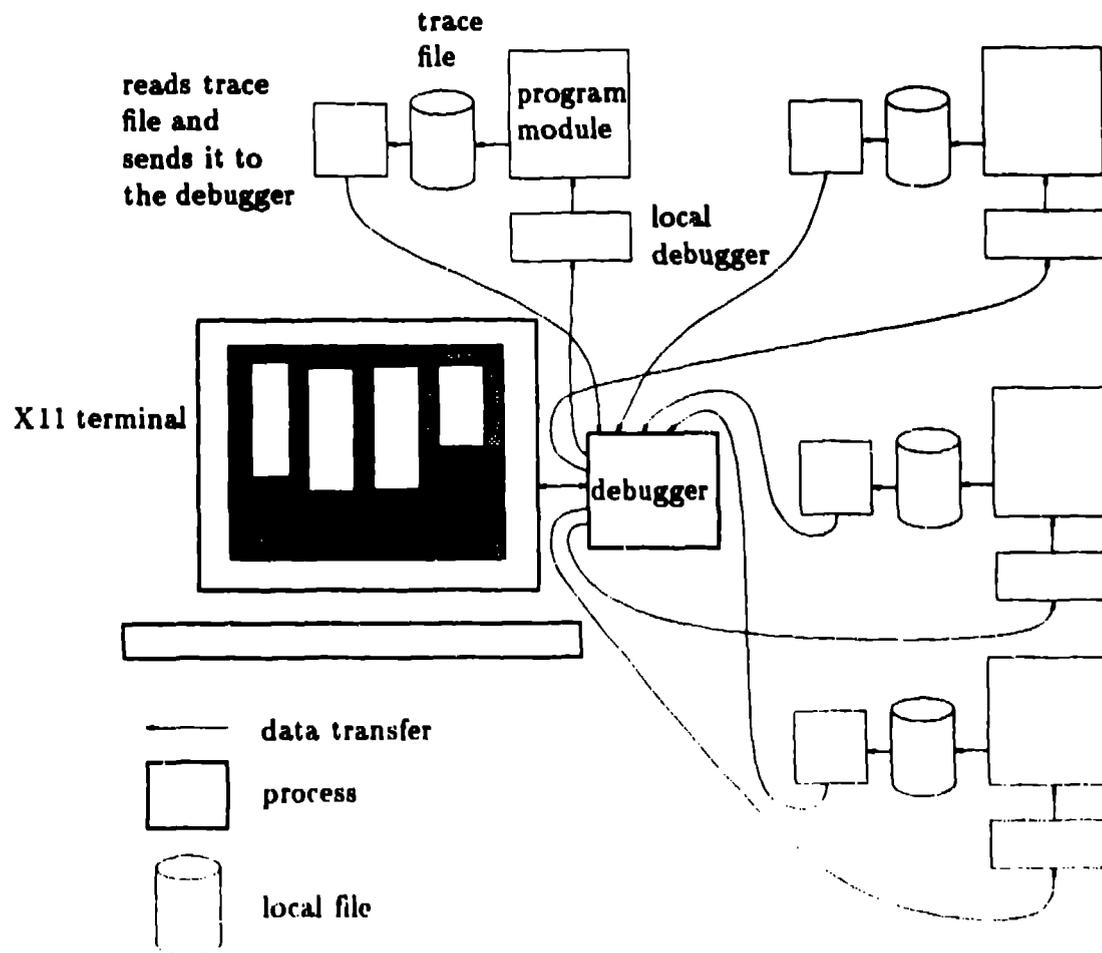


Figure 2: Architecture for online monitoring

central debugger which interacts with the user. This partitioning is chosen to minimize the probe effect on the module. If a node consists of several processors the process which reads the trace file can run on its own processor. If the central debugger has to communicate with a ParMod module this will be done via sockets and a local debugger. The local debugger has only to execute simple tasks like to trigger when more or less information has to be written to the trace file. Therefore the local debugger can be integrated in the runtime system of every program module.

Furthermore, *Source* is not only a debugger for the parallel aspects of a program run – communication and synchronization between tasks. Unlike many other trace based debuggers *Source* allows to examine variables and

control flow like sequential debuggers do, what we consider as necessary. How this is done is described below.

Another aspect of the design of *Source* is that it allows the integration with other tools of our parallel programming environment like the performance measurement and visualization tool *Runtime*. This is achieved through a common trace file format. The same trace can be used in both tools. This allows to detect a bottleneck with *Runtime*, and to analyse the program run at the location of the bottleneck in more detail with *Source*.

4 Trace Generation

When a ParMod program is compiled and linked with the debug option enabled, the compiler instruments the program for tracing certain events, including the creation and destruction of tasks, the entering and leaving of critical sections, and similar communication and synchronization oriented events. Overall there are about 40 different events of this kind. Additionally, the programmer may specify trace events with the following expression:

```
trace ( expression1 ) if ( expression2 )
```

The ParMod-compiler generates code from this expression which will write the value and an identification of *expression1* in the trace when *expression2* is true. The value of this expression is the value of *expression1* independent of the value of *expression2*. The purpose of *expression2* is mainly to limit the size of the trace. 'if (*expression2*)' can be omitted. This is equivalent to 'if (1)'. *expression1* may also be omitted. Then **trace** may only be used in a position where a statement is allowed and only the current line number is written in the trace. The purpose of this is to see more positions in the program run when the program is visualized. A suitable external representation of *expression1* is deduced from the type of *expression1*. This is an advantage of the integration of the trace statement in the programming language ParMod.

The following examples show how this construct can be used.

Examples:

```
await(trace(i) > 5) ...  
a[trace(j)if(j > n)] = trace(b[trace(i)]);  
trace();  
trace()if(z > z > n);
```

The trace is written in a very compact binary format. This reduces the trace size compared with the former ASCII trace format to less than 25

%. But the main reason was not to reduce the size of the trace but to reduce the probe effect. This is achieved since the costs for compressing at runtime are significantly lower than the i/o costs for writing 4 times more trace information.

5 Visualization

Source presents the user a global snapshot of the program on the screen. The user can move forward and backward in a program run from one snapshot to the other with a step size to be set. *Source* visualizes every task in a separate window. The headline of each task window shows information for the identification of the task (module, procedure, task number, caller) and about the current status of the task. The rest of a task window is divided into two parts:

- a possibly empty subwindow which shows the values of traced expressions which are valid in the current snapshot, and
- a subwindow which shows the source code with the current line highlighted.

All tasks of a module are grouped. In a module all tasks which execute the same procedure are also grouped. The user may move windows on the screen via a window manager but the debugger normally places the windows at useful default positions. If the screen is too crowded with too many task windows, the user can unselect modules or procedures which are not interesting in the moment. The user can also specify breakpoints and so called clock breakpoints, i.e. breakpoints which are triggered by time. He may also specify the module, the procedure, line numbers, expression identifications or expression values, when he sets a breakpoint. Clock breakpoints are useful for reaching a snapshot which corresponds to a position of a program run which is identified in *Runtime*.

A sample screen can be seen in figure 3. The window in the upper right corner is for entering a breakpoint, the window in the lower right corner is for selecting modules and procedures. The window in the middle at the lower border of the screen is the control window, in which commands for the debugger are entered.

6 Comparison with other approaches

Other debuggers for parallel and distributed programs have been described for instance in [ACM91] and [ACM88]. *Source* differs from most of these approaches by the possibility to move forward and *backward* in a program, by the integration with the other monitoring tools through a common trace format and the ability to inspect variables in a trace based tool.

7 Current status and future work

Currently the modifications necessary for online-debugging in the debugger *Source* are completed. The tracing in the workstation implementation of ParMod-C will be finished at the end of 1992. The implementation of a stand-alone reproduction system has just started.

We will use the debugging tools in our parallel programming course and for development of software systems which are implemented in ParMod-C. This includes a parallel database system and recursive numerical algorithms.

References

- [ACM88] *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, Wisconsin, May 5-6 1988.
- [ACM91] *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, Santa Cruz, California, May 20-21 1991.
- [AFP90] F. Abstreiter, M. Friedrich, and H.-J. Plewan. Das Werkzeug Runtime zur Beobachtung verteilter und paralleler Programme. SFB-Bericht 342/4/90 B, Technische Universität München, 1990.
- [ANS90] ANSA, Cambridge, UK. *Testbench Implementation Manual*, 1990.
- [EA88] Stefan Eichholz and Franz Abstreiter. Runtime observation of parallel programs. In *CONPAR88*, Manchester (UK), September 1988.
- [Eic86] S. Eichholz. ParMod – A language for programming with parallel modules. Technical Report TUM 18616, Technische Universität München, November 1986.

- [Eic87] Stefan Eichholz. Parallel programming with ParMod. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 377–380. Pennsylvania State University Press, May 1987.
- [FZ89] M. Friedrich and J. Zeiler. Simulation of hardware and multitasking for the parallel programming language ParMod. *Mircoprocessing and Microprogramming*, 1989.
- [Gai86] Jason Gait. A probe effect in concurrent programs. *Software – Practice and Experience*, 16(3):225–233, March 1986.
- [Mäs91] Wolfgang Mäsel. Online-Debugging von verteilten ParMod-Programmen. Master’s thesis, Institut für Informatik, Technische Universität München, May 1991.
- [Wei88] A. Weininger. Entwurf und Implementierung von Testhilfen für ParMod-Programme. Master’s thesis, Institut für Informatik, Technische Universität München, December 1988.
- [WSF91] Andreas Weininger, Thomas Schnekenburger, and Michael Friedrich. Parallel and distributed programming with ParMod-C. In H.P. Zima, editor, *Lecture Notes in Computer Science 591, Parallel Computing, First International ACPC Conference*, pages 115–126, Salzburg, October/November 1991. Springer.

Analyzing Traces of Parallel Programs Containing Semaphore Synchronization

D. P. Helmbold, C. E. McDowell, T. Haining

September 1, 1992

Abstract

One important kind of correctness and performance debugging tool for parallel programs determines and presents temporal ordering relationships between the various synchronization operations in the program. The particular ordering relationship we study is the “always happens before” relationship for arbitrary programs using semaphore synchronization. Our analysis is based on an execution trace of the program rather than the program itself. We have previously published a polynomial time conservative approximation of the “always happens before” relationship. Determining the exact “always happens before” relationship is intractable (co-NP Hard).

We built a random program generator and applied our algorithm to the random programs it generated. Our algorithm’s results were compared with the partial orders produced by an exponential time brute force algorithm which appears practical only for the relatively small programs generated. This process quickly identified traces where our algorithm failed to find some of the “always happens before” orderings. The findings from these experiments and the resulting modifications to our algorithm are described in this paper.

1 Introduction

Parallel programs with explicit synchronization can be notoriously difficult to debug. One important kind of correctness and performance debugging tool determines and presents the temporal ordering relationships between the various synchronization operations in the program. The particular ordering relationship we study is the “always happens before” relationship. Informally, event e_1 “always happens before” event e_2 if e_1 happens before e_2 in every execution in which e_2 occurs.

The complexity of determining the ordering relationships varies depending upon the type of synchronization and the branching characteristics of the program. There are polynomial time algorithms for simple models such as message passing with two-way naming or post and wait events with no clear in straight line programs (with no branches except bounded

loops) [NG92]. On the other hand, undecidability issues arise when arbitrary programs are considered. We have been studying this problem for arbitrary programs using semaphore synchronization. We avoid the undecidability problems by basing our analysis on a trace of the program rather than the program itself. Even with this restriction the problem is intractable (co-NP Hard), so we settle for a polynomial time conservative approximation of the “always happens before” relationship.

One possible application of the “always happens before” relationship is the detection of data races. If all accesses to a shared variable are ordered by the “always happens before” relationship then there is no race on the variable. Our conservative approximation of the “always happens before” relationship leads to the detection of more races than can occur, but guarantees the detection of every race that did occur in the execution of the program that was analyzed.

Care must be taken when generalizing from a trace to the entire program. One important situation where our trace results can be generalized is when we detect that there are no data races in the trace. This means that there are also no data races in the program (when executed with the same input).

In [HMWar] we described our basic approach. That algorithm takes an execution trace and produces a partial order representing the temporal orderings that always hold for the given program on the given input with two limitations:

1. It may fail to indicate that two events are ordered when they must always occur in a particular order (i.e. it is a conservative under-approximation of the “always happens before” relationship).
2. It may indicate that event e_1 “always happens before” event e_2 when there is a radically different execution where e_2 happens before e_1 . When this happens the radical difference between the executions is caused (directly or indirectly) by some other race in the program that will be detected by our algorithms. (I.e. if we report there are no races then there are none, but we may not report all of the races.)

Other researchers are pursuing this problem from the other end (e.g. [NM91]). Given a set of potential races (as might be reported by our analysis) their techniques identify a subset of the races that that can actually occur. We believe these to be complementary approaches and are continuing to refine our algorithms to reduce the above limitations.

In order to refine our algorithm it was necessary to determine when it failed (i.e. find a program and an execution trace containing two events e_1 and e_2 such that e_1 happens before e_2 in every execution of the program in which they both occur and our algorithm indicates that e_1 and e_2 are unordered). Given the limited set of “real” programs available to us in the programming language we currently support (IBM Parallel Fortran) our algorithm never fails (i.e. it finds exactly those orderings that hold for all executions given a particular input). We therefore built a random program generator and used our algorithm on the random programs it generated. Our algorithm’s results were compared with the partial orders produced by an exponential time “brute force” algorithm which appears practical only for the relatively

small programs generated. This process quickly identified traces where our algorithm failed to find some of the “always happens before” orderings. These experimental results have led to considerable modifications to our algorithms.

2 Overview of the Algorithm

Our algorithms use vector timestamps [Fid88] for each event to represent the partial order. We call a partial order *safe* if it contains a proper subset of the edges in the “always happens before” ordering. We first compute a very conservative safe partial order from an execution trace and then attempt to insert new edges into this safe partial order while maintaining the safety property. Edges are inserted into the partial order by manipulating the timestamps assigned to the events.

We only consider synchronization using counting semaphores with the two semaphore operations *signal* and *wait*¹. To compute the initial safe partial order we assume that any signal event could have been the signal that releases any wait. Our previous algorithm for inserting additional edges was based upon the following observations:

Observation 1 *If some wait event e_w on semaphore A is known to follow n other waits on semaphore A (given the safe partial order already computed) then we know that e_w must follow $n + 1$ signal events on semaphore A . Thus additional edges can be inserted into the partial order by increasing the (vector) timestamp for e_w so that each component is at least as big as the corresponding components in $n + 1$ of the timestamps for the signal events on semaphore A .*

Observation 2 *If one of the $n + 1$ signals, call it e_s , needed in observation 1 is known to be preceded by an additional wait event on semaphore A that is not one of the n wait events known to precede event e_w , then $n + 2$ signals occur before e_w whenever e_s occurs before e_w .*

This second observation implies that $n + 1$ signals other than e_s occur before e_w . In the program of Figure 1, the S_1 event in task D corresponds to the e_s event in Observation 2. We call this phenomenon *shadowing*, as the “shadow” cast by the preceding wait prevents e_s from satisfying the signals needed by e_w .

Our study of random programs has motivated changes in the algorithms so that they exploit the following additional observations:

Observation 3 *If wait event e_w on semaphore A is known to follow n waits on some other semaphore B (given the safe partial order already computed) then we know that e_w must follow n signal events on semaphore B .*

This extends observation 1 to apply to semaphores other than the one specified in the event e_w . The example that lead to this observation is shown in Figure 2.

¹These are more descriptive for English speakers than the original V and P of Dijkstra.

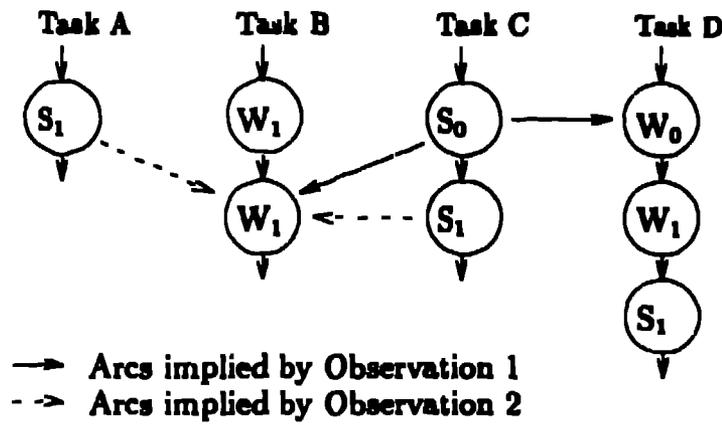


Figure 1: An Example of Observations 1 and 2. Note that the program can deadlock with Task D waiting on semaphore 1.

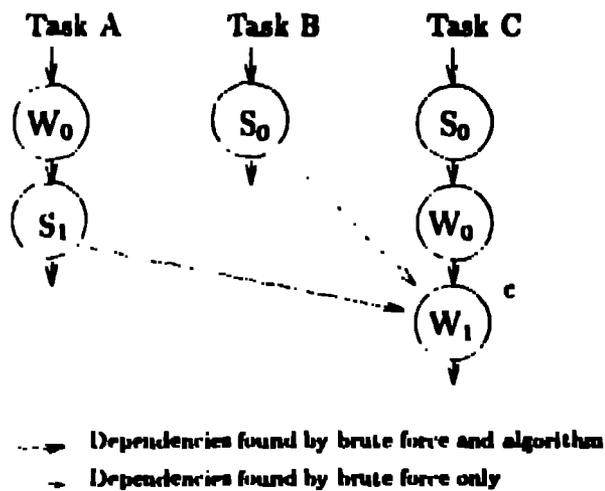


Figure 2: Example motivating observation 3. Event c is preceded by two wait on semaphore 0 events (W_0 's) and therefore must be preceded by two S_0 's.

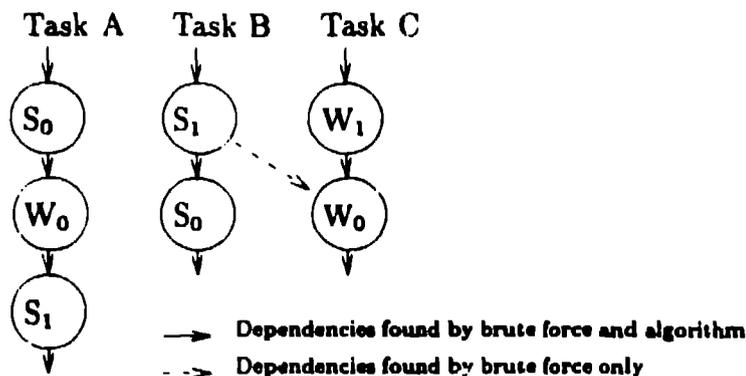


Figure 4: Example requiring two applications of observation 4.

A trace of an execution is simply a totally ordered list of signal and wait events together with an indication of which task performed each event. As events are entered in the trace only when the associated operation completes, every prefix of the trace contains at least as many signal events on each semaphore as wait events.

The “program” corresponding to the trace can be viewed as in Figures 1, 2, 3, and 4. There the events are grouped in vertical lists by task. Within each vertical list the events appear in the same order as they occurred in the trace. A “state” of the program corresponds to selecting a (possibly empty) prefix of each vertical list. Usually some program states are unreachable by any execution. For example, in Figure 4 the state where Tasks A and B have executed no events but Task C has executed its first event (W_1) is unreachable as the wait cannot complete until after semaphore 1 has been signaled.

The brute force analyzer performs a depth-first search on the program’s (exponentially large) state space to discover all of the reachable states. If event e_1 has been executed in every reachable state where event e_2 has been executed then the brute force analyzer indicates that e_1 “must happen before” e_2 . Performing this test for each pair of events allows the brute force analyzer to compute the “must happen before” relationship for the “program”.

If a real program makes conditional branches that could be affected by races, then the the brute force analyzer may overestimate the “must happen before” relationship. Our algorithms also have this drawback as noted in the introduction. However, even the overestimated “must happen before” relationship allows the race affecting the conditional branch to be flagged. Our experiments concentrate on how often the “must happen before” relationship computed by our other algorithms matches that returned by the brute force analyzer.

The random trace generator uses the parameters `MaxTasks`, `MaxSemaphores`, and `NumEvents`. For each trace pick `NumTasks` (between 2 and `MaxTasks`) and `NumSemaphores` (between 1 and `MaxSemaphores`) uniformly at random. Once these values have been set, the generator executes the loop in Figure 5 to output the trace of a non blocking execution.

```

while less than NumEvents generated do
  select T randomly between 1 and NumTasks
  select sem randomly between 1 and NumSemaphores
  if more signals than waits have been generated for sem
  then
    flip a coin
    if heads then output:  $W_{sem}$  by task T
    if tails then output:  $S_{sem}$  by task T
  else
    output:  $S_{sem}$  by task T
end while

```

Figure 5: Random program generator.

# of events	original	depth 1	depth 2	depth 3	total # of traces
30	54	3	0.26	0.05	5726
35	58	3.7	1.4	1.1	840
40	59	3.4	0.75	0.56	1070

Table 1: Percentage of random traces that failed to find at least one edge when compared to the actual “must happen before” partial order. Results are given for the original algorithm and for our revised algorithm with the depth of recursion set as indicated.

At the time of this writing we have done only a few tests aimed at determining the accuracy of the algorithm, but the numbers are very encouraging. The results are given in Table 1.

4 Conclusion

Our approach of using randomly generated programs to improve our algorithms has been extremely successful. When applied to small randomly generated straight line parallel programs, our new algorithm failed to find all “must happen before” edges less than 0.25% of the time. We believe that for real programs the failure rate will be significantly lower

References

- [Fid88] C. J. Fidge. Partial orders for parallel debugging. In *Proc. Workshop on Parallel and Distributed Debugging*, pages 183–194, May 1988.
- [HMWar] D. P. Hembold, C. E. McDowell, and J. Z. Wang. Determining possible event orders by analyzing sequential traces. *IEEE Transactions on Parallel and Distributed Systems*, to appear. Also UCSC Tech. Rep. UCSC-CRL-91-36.
- [Mat88] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Proceedings of Parallel and Distributed Algorithms*, 1988.
- [NG92] R. H. B. Netzer and S. Ghosh. Efficient race condition detection for shared-memory programs with Post/Wait synchronization. In *Proc. International Conf. on Parallel Processing*, 1992.
- [NM91] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. *SIGPLAN Notices (Proc. PPOPP)*, 26(7):133–144, 1991.

A Algorithm Details

Before each trace is analyzed, every event contained in that trace is assigned a vector timestamp. A timestamp contains one entry for each task. For a timestamp τ , $\tau[i]$ is the number of events completed by task T_i at the time the event associated with τ completed. When properly maintained, ordered and unordered event pairs can easily be distinguished by comparing their time vectors.

An event e with timestamp $\tau(e)$ precedes another event e' with timestamp $\tau(e')$ in the partial order if and only if every component of $\tau(e)$ is less than or equal to the corresponding component of $\tau(e')$. Events e and e' are unrelated in the partial order when both some component of $\tau(e)$ is greater than the corresponding component of $\tau(e')$, and some (other) component of $\tau(e')$ is greater than the corresponding component in $\tau(e)$.

Definition 1 For any two time vectors τ_1, τ_2 in Z^n

1. $\tau_1 \leq \tau_2 \iff \forall i(\tau_1[i] \leq \tau_2[i])$

2. $\tau_1 < \tau_2 \iff \tau_1 \leq \tau_2$ and $\tau_1 \neq \tau_2$

3. $\tau_1 \not\leq \tau_2 \iff \exists i(\tau_1[i] > \tau_2[i])$

The time vector τ_1 is earlier than time vector τ_2 (or τ_2 is later than τ_1) when $\tau_1 < \tau_2$.

At present, we use three algorithms to build the partial orders which approximate the ordering properties of events in a program trace. The first extracts the corresponding partial order from the event trace. The second modifies this partial order to ensure that it is valid for all possible executions in which the same events occur. The third uses the observations presented earlier in this paper to add back some ordering arcs which still are present in all executions.

Before defining our algorithms it is necessary to define a few common functions that we employ:

Definition 2 For any m time vectors τ_1, \dots, τ_m of Z^n

- $\overline{\text{min}}_k(\tau_1, \dots, \tau_m), k > 0$ is the vector of Z^n whose i th component is the k^{th} smallest element in the collection $\tau_1[i], \tau_2[i], \dots, \tau_m[i]$,
- $\overline{\text{max}}(\tau_1, \dots, \tau_m)$ is the vector in Z^n whose i th component is $\max(\tau_1[i], \dots, \tau_m[i])$.

Conventionally, we define $\overline{\text{min}}_0(\tau_1, \dots, \tau_m)$ to be $\vec{0}$, the all-zero vector.

As an example, $\overline{\text{min}}_3(\{[1, 2], [1, 3], [2, 4], [2, 5], [3, 2]\})$ is $[2, 3]$.

We also employ two special types of timestamps:

Definition 3 Given an event e performed by task T_i in a trace, let $\tau^\#(e)$ be the time vector containing the local event count for e (one more than the number of events previously performed by T_i in the trace) in the i th component and zeros elsewhere.

Definition 4 Given an event e performed by task T_i in a trace, let e^p denote the previous event performed by T_i in that trace (or the all zero vector if no such an event exists).

To obtain an initial partial order for an execution trace, on which further analysis can be based, we use an algorithm based on the one provided in Fidge [Fid88] and Mattern [Mat88]. This process associates a Wait event with an unused Signal event on the same semaphore, creating a partial order which pairs every Wait event with a Signal event which allowed it to precede. This generates a partial order that contains orderings that are not “must happen before” orderings. Instead, this partial order represents the causal orderings that did occur in a particular execution.

To generalize this partial order information to make it valid for all possible executions containing the same events, we use a process called *rewinding* (Algorithm 1) to decouple Wait events from specific Signal events. After rewinding, every Wait event has a time vector that reflects the assumption that any Signal (on the same semaphore) could have been the Signal that triggered the Wait.

Algorithm 1 (Rewind)

Repeat the following procedure until no further changes are possible.

```
for each event  $e$  in the trace
  if  $e$  is a Wait event on semaphore  $S$ ,
    let  $e_1^s \dots e_k^s$  be all the Signal events on  $S$ ;
    set  $v_s = \overline{\min}(\tau(e_1^s), \dots, \tau(e_k^s))$ ;
  else
    set  $v_s = \bar{0}$ , the all zero vector;
  end if;
  set  $\tau(e) = \overline{\max}(\tau(e^p), \tau^\#(e), v_s)$ ;
end for;
```

Unfortunately, the newly established safe order relation is too conservative because some “must happen before” ordering arcs are deleted as a part of the rewind procedure. At this point we apply an algorithm based on the observations described in Section 2.

The Expand algorithm (Algorithm 2) cycles through the entire timestamp representation of the trace, using a routine called *modify* to advance the timestamps of Wait events based on Observation 4.

It does this by considering the set of Signal events, R , and the set of Wait events, W , that exist in relation to two timestamps: τ_m and $\tau(\hat{e})$. Timestamp τ_m is an amalgamated timestamp representing all the events previously considered through recursive calls to *modify*. It will act as the event e_w in the current application of Observation 4. Event \hat{e} functions as the Signal e_s from Observation 4. The quantity *depth* indicates the number of times that Observation 4 is to be recursively applied.

Function *modify* builds up a set, T , of virtual timestamps returned by calling *modify* recursively on the events in R . It then employs Observation 3, taking the k^{th} component-wise minimum of T (where k is the number of Wait events in W) to determine the earliest time that \hat{e} can occur. It then returns \hat{e} 's new timestamp in the quantity τ_m .

Finally, the main loop takes the timestamp returned by *modify* for a specific Wait event, and uses $\overline{\max}$ to combine it with $\tau(e^p)$ and $\tau^\#(e)$.

Algorithm 2 (Recursive Expand):

Repeat the following procedure until no more changes are possible.

for each event e in the trace

if e is a Wait event on semaphore S ,

$\tau(e) = \text{modify}(\tau_{\text{max}}, \tau_{\text{in}}, e, \text{depth});$

set $\tau(e) = \text{MAX}(\tau(e), \tau(e^p), \tau(e));$

else

set $\tau(e) = \text{MAX}(\tau(e), \tau(e^p));$

end if;

end for;

Function $\text{modify}(\tau_{\text{max}}, \tau_{\text{in}}, e, \text{depth}):$

let $\tau_m = \tau(e)$

if $\text{depth} = 0$

return(τ_m);

end if;

for each semaphore σ ,

let $T = \emptyset;$

let $W(\sigma) = \{e_w : e_w \text{ is a Wait event on } \sigma, \text{ and}$
either $\tau(e_w) \leq \tau_{\text{in}}$ or $\tau(e_w) \leq \tau(e)\};$

let $R(\sigma) = \{e_s : e_s \text{ is a Signal event on } \sigma,$
 $\tau(e_s) \not\leq \tau_{\text{in}} \text{ and } \tau(e_s) < \tau_{\text{max}}\};$

for each e_s in R ,

$\tau = \text{MAX}(\tau_{\text{in}}, \tau(e));$

$T = \{\text{modify}(\tau_{\text{max}}, \tau, e_s, (\text{depth} - 1))\} \cup T;$

end for;

let $k = |W(\sigma)|;$

let $\tau_s = \overline{\min}_k(T);$

let $\tau_m = \text{MAX}(\tau_m, \tau_s);$

end for;

return(τ_m);

Compile-time Support for Efficient Data Race Detection in Shared-Memory Parallel Programs

John Mellor-Crummey*

Center for Research on Parallel Computation
Rice University
Houston, TX 77251-1892
johnmc@rice.edu

Abstract

To date, both on-the-fly methods for detecting data races during program executions and post-mortem methods for analyzing traces of program executions have made little use of compile-time analysis to reduce the number of accesses that must be examined. In this paper we describe program analysis for this purpose that has been implemented as part of a debugging system for the ParaScope Programming Environment. To demonstrate the effectiveness of our analysis techniques, we present measurements that compare the overhead of race detection with three levels of compile-time analysis ranging from little analysis to aggressive interprocedural analysis. Since the monitoring overhead of run-time techniques for data race detection is high, improvements achieved using compile-time support will play an important role in making run-time techniques for detecting data races practical.

1 Introduction

In an execution of a shared-memory parallel program, a *data race* is said to exist when there are two or more unordered accesses to the same shared variable and at least one is a write. In the presence of a data race, the program's execution behavior may depend on the temporal order of the accesses participating in the race. The result is that the program may exhibit erroneous behavior in some, but not necessarily all, executions.

Detecting data races in shared-memory parallel programs is a difficult problem. Strategies for detecting such race conditions can be generally classified as (1) static analysis — analysis of a program's text to determine when two references may refer to the same shared variable [1, 3, 4, 21], (2) post-mortem analysis — collection of a log of events that occur during a program's execution and post-processing the

log to isolate data races [2, 8, 17], or (3) on-the-fly analysis — augmentation of a program to detect and report data races as they occur during its execution [11, 12, 15, 16, 18, 19, 20].

Static analysis techniques rely on classical program dependence analysis and an analysis of a program's concurrency structure to determine when two references may potentially be involved in a data race. Static techniques are inherently conservative, which often leads to reports of data races that could never occur during execution. Experience with static analysis tools has shown that the number of false positives reported using these techniques is too high for programmers to rely exclusively on static methods for isolating data races. Combining static analysis with symbolic execution offers hope for reducing reports of infeasible races [23].

Post-mortem techniques for detecting data races have the advantage that they can limit reports to feasible races. However, to guarantee that only feasible races are reported, exhaustive execution trace logs are necessary. The size of such execution logs is a serious drawback for these methods since the logs can be enormous for parallel programs that execute for more than a trivial amount of time. A promising alternative to pure post-mortem analysis is a hybrid approach that uses abbreviated logs containing only synchronization information to compute guaranteed orderings. Such orderings can be used in conjunction with subsequent static analysis or on-the-fly monitoring to report race conditions.

On the fly techniques maintain additional state information during a program's execution to determine when conflicting accesses to a shared variable have occurred. The principal drawback of on the fly techniques is that in the general case, the space and time overhead of detecting races during a program's execution can be enormous.

For on the fly techniques to become widely accepted, their space and time overhead must be reduced to a level that programmers can tolerate in practice. Known

* This work was supported in part by National Science Foundation Cooperative Agreement CCR 91 20008.

on-the-fly techniques applicable to programs with an unrestricted concurrency structure have asymptotic, worst-case space overhead that includes a term proportional VT , where V is the number of shared variables and T is the maximum logical concurrency in the program execution [11]. However, when the structure of logical concurrency is restricted to that achievable with closed, nested fork-join constructs (e.g., nested parallel loops), the worst-case asymptotic space overhead can be reduced to $O(VN)$, where N is the maximum dynamic nesting depth of parallel constructs, and the asymptotic time for testing if a particular access participates in a data race can be reduced to $O(N)$ [15]. These tighter bounds offer the promise of efficient on-the-fly detection of data races for this restricted class of programs.

In addition to having asymptotically efficient methods for testing whether a variable access is involved in a data race, for on-the-fly techniques to be practical, the number of access checks to test for data races during a program's execution must be minimized. In this paper we focus on a compile-time strategy for automatically instrumenting a program to detect data races at run time. Our instrumentation system, implemented as a component of the Rice University's ParaScope Programming Environment, exploits sophisticated static analysis to reduce the number of data race access checks added to programs.

Section 2 briefly describes the ParaScope system and provides context for the implementation. Sections 3.1-3.3 describe three approaches to data race instrumentation that rely on increasing amounts of program analysis to reduce the number of access checks added to a program. Section 4 describes some preliminary experimental results that compare the effectiveness of the three different instrumentation strategies. Finally, section 5 briefly outlines our plans for enhancing the instrumentation system to exploit additional kinds of analysis to further prune the number of access checks added to programs.

2 ParaScope Environment

ParaScope is a programming environment for scientific Fortran programmers. It has fostered research on aggressive optimization of scientific codes for both scalar and shared memory machines [7]. ParaScope provides an infrastructure for management, analysis, and transformation of programs written in a Fortran dialect with extensions for shared memory parallelism. It provides a rich collection of analyses including dependence analysis, control flow graph analysis, computation of static-single assignment form, and global value numbering among others.

Through careful design, the compilation process in ParaScope preserves separate compilation of procedures to a large extent. Tools in the environment cooperate to minimize the number of times a procedure needs to be examined during compilation. In general, the existing compilation system uses the following 3-phase approach [7, 9, 13]:

1. **Local Analysis.** At the end of an editing session, ParaScope calculates and stores summary information concerning all local interprocedural effects for each procedure. This information includes details on call sites, formal parameters, scalar and array section uses and definitions, local constants, symbolics, loops and index variables. Since the initial summary information for each procedure does not depend on interprocedural effects, it only needs to be collected after an editing session, even if the program is compiled multiple times or if the procedure is part of several programs.
2. **Interprocedural Propagation.** The compiler collects local summary information from each procedure in the program to build an *augmented call graph* containing loop information [14]. It then propagates the initial information on the call graph to compute interprocedural solutions.
3. **Interprocedural Code Generation.** The compiler directs compilation of all procedures in the program based on the results of interprocedural analysis.

Another important aspect of the compilation system is what happens on subsequent compilations. In an interprocedural system, a module that has not been edited since the last compile may require recompilation if it has been indirectly affected by changes to some other module. Rather than recompiling the entire program after each change, ParaScope performs *recompilation analysis* to identify modules that have not been affected by program changes, thus reducing recompilation costs [5, 10].

ParaScope computes interprocedural REF, MOD, ALIAS and CONSTANTS. Implementations are underway to solve a number of other important interprocedural problems, including interprocedural symbolic and regular section analysis of arrays.

3 Data Race Instrumentation

Internally, ParaScope represents program modules in the form of abstract syntax trees annotated with semantic information. A program transformation subsystem supports arbitrary transformation and augmentation of

Fortran ASTs. This system serves as the framework for the data race instrumentation system that we have constructed as part of ParaScope. The data race instrumentation system uses the transformation subsystem to transform Fortran ASTs, adding calls to run-time support routines that enable a program to detect data races during its own execution.

The focus of the ParaScope data race instrumentation system is to detect data races that arise during execution of parallel loops. This restriction in the current implementation of the instrumentation system stems from the lack of full support in ParaScope for heterogeneous parallelism that arises from Fortran parallel section constructs (analogous to the more familiar cobegin-coend construct of other parallel languages). Wherever this deficiency of the analysis affects the instrumentation system, the term *parallel loop* will be used in place of the term *parallel construct* in order to remain faithful to the actual implementation.

To prepare a program for data race instrumentation, the following sequence of transformations are applied to put the code in a canonical form:

- Transform logical IF statements into block IF statements so that instrumentation can be readily be added to the consequent as necessary.
- Transform each ELSEIF construct into an IF-THEN construct nested inside an ELSE construct. If any access in the ELSEIF guard needs instrumentation, the system must have a place to insert the instrumentation so that the access check gets executed *iff* the guard will be executed.
- Hoist all function invocations out of subscript expressions. Since subscript expressions are duplicated into access check code, subscript expressions must be side-effect free.
- Move each statement label to a CONTINUE that precedes the statement. Since the system will insert data race instrumentation for a statement immediately before the statement, the system must ensure that it is impossible to reach a statement without executing its corresponding access check instrumentation.

Once the code is in a canonical form, the data race instrumentation can proceed. The data race instrumentation process consists of adding several different types of statements

- concurrency bookkeeping — calls to race detection run-time library routines to indicate the creation or termination of a logical thread¹.

¹We use the term *thread* to denote the basic unit of concurrency (e.g., an iteration of a parallel loop body)

- access checks — calls to the race detection run-time library READCHECK or WRITECHECK operations that test if an access participates in a data race,
- access history declarations — each variable that may be involved in a data race is allocated storage for an access history in which information is stored about the threads that access the variable, and
- access history initialization and finalization — access histories for all local variables must be initialized upon procedure entry and finalized before the procedure returns or halts.

The next three sections describe data race instrumentation strategies that rely on increasing levels of program analysis.

3.1 Basic Strategy

Without any sophisticated analysis, data race instrumentation must be very conservative. Each procedure must assume that it is called in the scope of a parallel construct. Therefore, references to its formal parameters (which are passed by reference in Fortran) and global variables must be instrumented since they could conflict with other accesses made in the context of an enclosing parallel construct. The system must also add access checks for references to local variables that occur inside the scope of a parallel construct in the procedure. This is necessary since without further analysis one cannot be certain that the variable is not the target of conflicting, concurrent accesses within the scope of the parallel construct.

Even if a procedure contains no parallel constructs, all local variables passed as actual arguments to user-defined procedures must have access history storage allocated in the current scope and references to that storage must be passed to each called procedure since any called procedure could contain a parallel construct inside of which it references its arguments. Local variables not passed to called procedures need not be instrumented nor have access history storage allocated if no parallel constructs are present in the current procedure.

Variable references passed to intrinsic functions require special handling. Intrinsic functions in Fortran read, but never modify their arguments. Since the bodies of intrinsic functions are not instrumented by this system, in some cases, the system must add instrumentation at the point of call to reflect that the intrinsic reads its arguments. In particular, a READCHECK for a variable reference passed to an intrinsic must be added at the point of call if

- the variable is a formal parameter or a global variable, or
- the variable is a local variable and the call to the intrinsic is inside the scope of a parallel construct in the current procedure.

For each statement, the instrumentation system accumulates the set of variable references that need data race instrumentation. If multiple array element references in the same statement have the same sequence of subscript expressions, only one access check is needed for all of the references. This is true even if the references are a mix of reads and writes — a single WRITECHECK will suffice since any access that conflicts with a read will certainly conflict with a write.

3.2 Intraprocedural Strategy

To detect all data races, not all references to shared variables inside parallel loops need be instrumented. In particular, references to variables that are not accessed by more than one thread of control do not need data race instrumentation.

Data dependence analysis is a deep compile-time analysis of program variables and their subscripts to determine when two variable references may refer to the same memory location. Compile-time dependence analysis computes a conservative superset of a program's run-time dependences. In ParaScope, a dependence graph contains an edge for each data dependence, where each node in the graph represents a variable reference. A dependence edge between references R_1 and R_2 is carried by a loop if the execution of R_1 in loop iteration i can potentially access the same memory location as the execution of R_2 in loop iteration j , $i \neq j$. Dependences that are not carried by loops are said to be *loop independent*.

Three types of carried data dependences are important for data race instrumentation. A *true* dependence (also known as *flow* dependence) signifies that a memory location read during some loop iteration may be overwritten in a later iteration. An *anti* dependence signifies that a memory location written during some loop iteration may be read in a later iteration. Finally, an *output* dependence signifies that a memory location may be written during more than one loop iteration. When a data dependence is carried by a parallel loop, two different instances of the loop body may access the same memory location in parallel resulting in a data race at run time.

In the absence of procedure calls, all variable accesses that may be involved in a data race must be the endpoint of some data dependence that is carried by a parallel loop. In this case, it suffices to add access checks

only for variable references that are endpoints of data dependences.

When procedure calls are present, but no interprocedural information is available, conservative assumptions are necessary to ensure correctness. When building a dependence graph, conservative assumptions must be made about the side effects of each procedure call. In particular, the dependence analyzer must assume that each procedure call modifies each of its actual parameters (in fact, the analyzer must assume that any time a reference to an array element is passed to a procedure, the procedure modifies the whole array) and all global variables. As before, access checks are added for each endpoint of a data dependence. Also, as in the basic instrumentation strategy presented in the previous section, each procedure must conservatively assume that it is invoked from inside a parallel construct which requires access checks for references to global variables and its formal parameters in addition to access checks at dependence endpoints.

3.3 Interprocedural Strategy

In the instrumentation strategies presented thus far, conservative assumptions are made in the presence of procedures. At each callsite, the system must assume that the called procedure modifies each of its actual parameters and all global variables. Furthermore, the system must assume that each procedure may be invoked from within a parallel construct.

These two assumptions lead the system to insert instrumentation conservatively. Interprocedural information can help the instrumentation system reduce the amount of data race instrumentation and its run-time overhead in two simple ways:

- If the system knows that a procedure is never called from within a parallel construct, no access checks are necessary in the procedure for references to its formal parameters except inside any parallel constructs contained in the procedure.
- If the dependence analyzer has interprocedural summary information about the side-effects (MOD and REF) of procedure calls in parallel constructs, it will not have to make the conservative assumption that all variables accessible to the procedure are modified. This can reduce the number of dependence endpoints, thus reducing instrumentation.

Additional improvements can be obtained in more subtle cases using the interprocedural analysis strategy described below. The description of the implementation strategy is presented for each of the three analysis phases in the ParaScope compilation system: local

parameters require instrumentation, but the third does not.

For each procedure, its final data race instrumentation set describes which formal parameters and global variables require access checks inside the procedure body. However, with only the information computed thus far, each caller must conservatively assume that each actual argument that it passes to a called procedure requires access history storage. In the example shown in figure 1, there is no way for the loop calling *f* to know that the second parameter to *f* requires access history storage but that the third does not since these requirements are dictated from below by *h*.

Which variables require access histories allocated can be computed in a single backward dataflow pass over the callgraph. The initial value of the *storage allocation set* for each procedure is a copy of the procedure's final data race instrumentation set. During a backward dataflow pass, the storage allocation sets flow to each procedure from all the procedures it calls (along an edge for each callsite inside the procedure). For each callsite, only the variables known to the caller are propagated through the callsite up to the caller. The sets propagated to a node along the callsite edges are unioned to achieve the final version of the storage allocation set for that procedure.

After applying this analysis to the program fragment shown in figure 1, the callsites for *f* and *g* will know that no access history storage is needed for *x* and *y* respectively. Furthermore, the call interface for each of the procedures needs to be expanded only for the parameters that actually require access history storage instead of for all variables would be the case if the basic and intraprocedural strategy.

After the storage allocation set is computed for each procedure, only the top-level program is aware of all of the common variables that must be expanded. A forward interprocedural dataflow pass is necessary to guarantee that each procedure has a consistent definition of which variables in each common block need to be augmented with access history storage. This pass creates a *common allocation set* for each procedure.

Code Instrumentation

After all of the interprocedural analysis is complete, the data race instrumenter uses the information collected to instrument the Fortran AST for the program. Each reference that is an endpoint of a dependence carried by a parallel loop has a corresponding access check. Also, each reference to a variable in a procedure's data race instrumentation set has access checking added. For each local variable in the procedure's storage allocation set, storage is allocated, and calls to run-time support

routines are added to initialize and finalize the storage upon entry and exit of the procedure respectively. Common block definitions are expanded with access history storage added for each variable in the procedure's common allocation set. Actual argument lists are expanded at callsites to pass access history storage only for parameters in the callee's storage allocation set. Calls to concurrency bookkeeping routines are added only for parallel constructs that carry a data dependence. Thus, if a data race can never occur in the context of a parallel construct, no concurrency bookkeeping is performed at run time.

4 Preliminary Results

In order to test the efficacy of the compile-time analysis described in the previous section it is important to apply the analysis to some real programs. To date, we have preliminary results with two programs.

The first program, *search*, implements a multi-directional direct search method for finding a local minimizer of an unconstrained minimization problem [22]. Search contains four parallel loop nests (each of which contain a call to the same evaluator function) surrounded by an outer serial loop that tests for convergence. The second program, *back*, tests the adjointness of a routine that computes a one-dimensional seismic inversion (used for oil exploration) with its associated adjoint code. The code contains seven parallel loop nests, four of which contain calls to substantial procedures.

Table 1 contrasts static and dynamic statistics for the search program. The table shows measures for both the uninstrumented code and for code automatically generated by the ParaScope data race instrumentation system using the three different data race instrumentation strategies. The first column in the table shows source lines comparing the size of the original uninstrumented program versus the size with each style of instrumentation. The dramatic increase in source line count reflects the addition of access checks, concurrency bookkeeping calls, declarations for access histories, calls to initialize and finalize each locally declared access history, as well as declarations and data statements that contain information that enables the ParaScope data race run-time support library to report errors by relating them back to the original source code. The next two columns respectively indicate how many READ-CHECK and WRITECHECK calls were added to the program for each instrumentation strategy. Compared to the basic strategy, the interprocedural approach reduces the combined number of access checks added to the code by 87%. Since the remaining access checks are

```

parallel loop i = 1, n
  call f(a[aindex[i]], b[i], x[i])
end loop
...
parallel loop i = 1, n
  call g(d[i], e[eindex[i]], y[i])
end loop

subroutine f(f1, f2, f3)
  call h(f1, f2, f3)
end

subroutine g(g1, g2, g3)
  call h(g1, g2, g3)
end

```

Figure 1: Different contexts have different instrumentation requirements.

analysis, interprocedural analysis, and module compilation.

Local Phase

As described in section 2, at the end of an editing session the ParaScope editing tools record initial information about a module for use during interprocedural analysis. Before support for data race instrumentation was envisioned in ParaScope, initial information recorded included a descriptor for each procedure specifying the names and types of formal parameters, initial MOD and REF information for formal parameters and common variables, callsite descriptors including name of the invoked procedure (or procedure variable) and the actual arguments. To support data race instrumentation, this information was reorganized so that it is not summarized at the procedure level, but rather collected at the loop level. Also, the information was augmented to contain a description of the loop nesting structure and an indication of which loops are parallel. Loop-level information is important for data race instrumentation so that interprocedural analysis can determine which procedures are (possibly transitively) invoked from within the context of a parallel loop.

Interprocedural Phase

As was the case before support for data race instrumentation was envisioned in ParaScope, a callgraph is constructed and interprocedural ALIAS, MOD, and REF summary information is computed for each procedure using the initial information collected during the local phase

Conceptually, at this point a null *data race instru-*

mentation set is created for each procedure. After interprocedural analysis is complete, this set will indicate which formal parameters and global variables require access checks for references to them inside the procedure body.

The interprocedural analysis driver then invokes the dependence analyzer for each procedure using the interprocedural solutions for MOD and REF to increase the precision of dependence information at callsites. Using the MOD and REF solutions, the dependence analyzer identifies when a data dependence involves side-effects of a callsite. Such dependences indicate accesses made by the called procedure (or its descendants in the callgraph) that may be involved in data races and instrumentation will be needed for any access to that variable in the called procedure (or its descendants).

For each dependence endpoint at a callsite (referring to an actual or global accessed as a side-effect of the call), the data race instrumentation set for the procedure is augmented to indicate that some context in which the procedure is called requires instrumentation for accesses to a particular formal parameter or global. When all of the callsites inside parallel loops in the program has been processed, the instrumentation sets are ready for dataflow propagation through the edges in the callgraph. Final values for the instrumentation sets result from forward dataflow propagation of all of the data race instrumentation sets along callsite edges in the callgraph. At each callsite, global variables and variables passed as actuals are propagate instrumentation requirements into the callee. During dataflow propagation, the instrumentation requirements flowing to a node (i.e., procedure) from each of its callsite edges are unioned to achieve the final version of the data race instrumentation set for that procedure.

A contrived example shown in figure 1 illustrates how context can impose different instrumentation requirements on a procedure's formal parameters. Assume that subroutine *h* modifies its first two arguments, but only reads the third. Interprocedural MOD summary analysis will indicate that subroutines *f* and *g* modify their first two arguments (via their call to *h*). In the context of the first loop, the ParaScope dependence analyzer cannot prove that accesses to *a[aindex[]]* are independent, but can prove that modifications to *b[i]* are independent. Since the third parameter to *f* is not in *f*'s MOD set, there is no loop-carried dependence involving this parameter. The context of this loop thus requires instrumentation inside *f* (and thus *h* as well) only for accesses to *f*'s first formal. In the second loop, the situation similar for the call to *g*: only accesses to its second formal parameter require instrumentation. After interprocedural dataflow propagation of the instrumentation sets, inside *h* its first and second formal

	static measures			dynamic measures		
	source lines	access checks read	access checks write	access checks read	access checks write	execution time
uninstrumented	569	0	0	0	0	36.9
basic	1073	70	17	111498737	15270102	602.3
intraprocedural	1065	68	17	100749809	15270102	552.6
interprocedural	697	4	7	23368976	15269825	247.8

Table 1: Data race instrumentation statistics for the search program.

	static measures				dynamic measures		
	source lines	instrumented loop nests	access checks read	access checks write	access checks read	access checks write	execution time
uninstrumented	2212	0	0	0	0	0	20.4
basic	3961	7	234	63	70085036	7295204	327.8
intraprocedural	3961	7	234	63	70085036	7295204	327.1
interprocedural	2518	3	14	20	15781629	4100000	91.1

Table 2: Data race instrumentation statistics for the buck program.

all in the computational kernel, the effective reduction of the dynamic access checks is not nearly as dramatic. In comparison to the basic strategy, the interprocedural strategy reduced the combined number of dynamic checks by 70%.

Table 2 contrasts the static measures for the buck program. Dependence analysis alone was able to determine that there are no dependences carried by three parallel loops, each encapsulated in its own procedure. However, since each of the loops contains accesses to the arguments of the enclosing procedure, access checks are still necessary inside the parallel loops since nothing is known about the contexts in which the procedures containing the loops are called, and whether accesses to the arguments could cause data races. With interprocedural information, the instrumentation system is able to determine that none of the procedures containing a parallel loop is called from within another parallel loop; therefore, all instrumentation can be omitted from the aforementioned three parallel loops immediately. For a fourth parallel loop that contains a call to a procedure, all instrumentation also was eliminated because the analysis was able to determine that the side effects of the procedure did not result in any carried dependences. (No instrumentation was needed inside the procedure called from within the fourth parallel loop either.) The interprocedural approach reduces the combined number of static access checks by 89% over both the basic and intraprocedural strategies. The interprocedural strategy reduced the number of dynamic checks 74% over in comparison to the other approaches.

All execution times reported in the last column of tables 1 and 2 are from sequential executions of the in-

strumented programs on a Sun² 4/490. Elsewhere we have shown that sequential executions suffice for detecting data races in programs with loop-based parallelism [15]. All programs were compiled with the Sun f77 compiler using -O optimisation. Comparing raw execution times of the uninstrumented and instrumented code varieties shows the run-time overhead for on-the-fly monitoring to be relatively high. These numbers offer a conservative picture of the overhead of on-the-fly monitoring since the ParaScope data race run-time library is written in a modular style of C++ and has not been tuned for performance; for instance, the concurrency bookkeeping routines invoke "malloc" for dynamic memory allocation of each thread label rather than a tuned special-purpose allocator. For the search program, the execution overhead (computed as (instrumented execution time - uninstrumented execution time)/uninstrumented execution time) of the basic strategy was a factor of 1532%, whereas the interprocedural approach reduced this to 571%. For the buck program, the execution overhead was 1507% for the basic and intraprocedural strategies. The interprocedural strategy reduced the run-time overhead for race instrumentation of buck to 347%.

5 Future Plans

The instrumentation system currently performs no interstatement analysis to remove redundant access check operations within a procedure. We are planning to explore the use of use global value numbers, information from control flow graph analysis, and analysis developed for

²Sun is a trademark of Sun Microsystems

recognizing reuse of array variables [6] to eliminate redundant access check operations.

Acknowledgments

Robert Hood implemented a large part of the program transformation system upon which the data race instrumenter is built and most of the intraprocedural instrumentation system prototype. Ken Kennedy, Mary Hall, Seema Hiranandani, and Chau-Wen Tseng provided most of the description of the interprocedural infrastructure in ParaScope that appears in Section 2.

References

- [1] R. Allen, D. Baumgartner, K. Kennedy, and A. Porterfield. PTOOL: A semi-automatic parallel programming assistant. In *Proc. of the 1986 International Conference on Parallel Processing*, pages 164-170, Aug. 1983.
- [2] T. R. Allen and D. A. Padua. Debugging fortran on a shared memory machine. In *Proc. of the 1987 International Conference on Parallel Processing*, pages 721-727, Aug. 1987.
- [3] W. F. Appelbe and C. E. McDowell. Anomaly reporting - a tool for debugging and developing parallel numerical applications. In *Proc. First International Conference on Supercomputers*, FL, Dec. 1985.
- [4] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Sublok. The ParaScope editor: An interactive parallel programming tool. In *Proc. Supercomputing '89*, pages 540-550, Reno, NV, Nov. 1989.
- [5] M. Burke and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, to appear.
- [6] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.
- [7] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4), Winter 1988.
- [8] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flow-back analysis. *ACM Transactions on Programming Languages and Systems*, 1991.
- [9] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the R² programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491-523, Oct. 1986.
- [10] K. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.
- [11] A. Dinning and E. Schonberg. An evaluation of monitoring algorithms for access anomaly detection. Ultracomputer Note 163, Courant Institute, New York University, July 1989.
- [12] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOP)*, pages 1-10, Mar. 1990.
- [13] M. W. Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, Apr. 1991.
- [14] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, Albuquerque, NM, Nov. 1991.
- [15] J. M. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proc. of Supercomputing '91*, pages 24-33, Albuquerque, NM, Nov. 1991.
- [16] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235-244, Palo Alto, CA, Apr. 1991.
- [17] R. H. B. Netzer and B. P. Miller. Detecting data races in parallel program executions. In D. Golerter, T. Gross, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing* MIT Press, 1991. Also in *Proc. of the 3rd Workshop on Prog. Langs. and Compilers for Parallel Computing*, Irvine, CA, (Aug. 1990).

- [18] I. Nudler and L. Rudolph. Tools for efficient development of efficient parallel programs. In *First Israeli Conference on Computer Systems Engineering*, 1986.
- [19] E. Schonberg. On-the-fly detection of access anomalies. In *Proc. ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 285-297, June 1989.
- [20] G. L. Steele, Jr. Making asynchronous parallelism safe for the world. In *Proc. of the 1990 Symposium on the Principles of Programming Languages*, pages 218-231, Jan. 1990.
- [21] R. N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362-376, May 1983.
- [22] V. J. Torczon. Multi-directional search: A direct search algorithm for parallel machines. Technical Report TR90-7, Department of Mathematical Sciences, Rice University, May 1989.
- [23] M. Young and R. N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499-1511, Oct. 1988.

Direct Manipulation Techniques for Parallel Debuggers

Cherri M. Pancake
(Visiting Scientist, Cornell Theory Center
Department of Computer Science
Oregon State University
Corvallis, OR 97331
pancake@cs.orst.edu

Abstract: Graphic displays offer the debugger developer a means of managing the density and complexity of the data generated during execution of parallel programs. In addition, graphical techniques can be used to simplify the ways in which the user interacts with the tool. Direct manipulation (using a mouse or other pointer device) of graphical objects reduces the number of physical and cognitive operations required of the user. This paper discusses how direct manipulation can be applied to debugger interfaces. Examples are drawn from prototype trace-based and breakpoint-style debuggers, but most can be applied to any parallel debugging tool.

Introduction¹

Graphical user interfaces, increasingly common among all types of software tools, are now a normal part of most production-level parallel debuggers. To date, however, "GUI" has been something of an exaggeration. These debuggers typically exhibit few graphical capabilities other than scrollable display windows and pushbutton controls. More sophisticated graphical techniques -- in the form of visualizations and direct manipulation mechanisms for interacting with them -- largely have been neglected.

¹ The research described here was carried out at the Department of Computer Science and Engineering, Auburn University (Auburn, AL 36849), and at the Center for Theory and Simulation in Science and Engineering, Cornell University (Ithaca, NY 14853). PF-View was developed with Sue Utter-Honig as part of a joint study funded by IBM Corporation; the XIPD project was supported by the Supercomputer Systems Division of Intel Corporation.

The criticisms of debugging tools voiced by the user community reflect this shortcoming. The complexity of parallel debuggers, the difficulty with which they are operated, and their inability to characterize program execution in useful ways are cited frequently as sources of dissatisfaction [11]. Graphical display techniques offer the debugger developer a means of managing the density and complexity of the data generated during execution of parallel programs [14]. Moreover, graphical mechanisms can be used to simplify the ways in which the user interacts with the tool.

This paper focuses on *direct manipulation*: the use of a mouse (or other pointer device) to interact via graphical objects displayed on the screen. The first section describes how direct manipulation mechanisms can enhance the ergonomic and cognitive² aspects of software tools. The discussion then turns to examples of how direct manipulation can be introduced into parallel debuggers. Specific examples demonstrate how direct manipulation facilitates user control over

- the rate at which information is displayed,
- the sequence in which information is displayed,
- the amount of information presented,
- the level of information presented, and
- the contents of debugger-controlled entities.

A final section draws some conclusions on the current state of direct manipulation techniques for parallel debuggers.

The Basis for Direct Manipulation

Recent advances in graphics technology have revolutionized the area of user interfaces. The widespread availability of graphics hardware, windowing platforms, and standard graphics libraries make it possible to develop software tools that can be ported across a variety of host machines and operating systems. In particular, the proliferation of inexpensive graphical display hardware and the subsequent popularity of window-based user interfaces have led to increasing demands on parallel debugger developers for graphical support.

Strictly speaking, graphical techniques are those which are non-textual in nature. They rely on shape, color, screened textures, etc., to represent logical and physical characteristics in figurative or symbolic form. It can be argued, however, that a short word (e.g., "run", "step", or "exit") in some cases conveys more direct meaning than a contrived icon. Consequently, the

² The terms ergonomic and cognitive are used here according to the taxonomy outlined in [Curtis]

definition of *graphical* will be broadened somewhat for the purposes of this discussion, encompassing symbolic use of simple words or acronyms as well as more traditional iconic, plotted, or rendered representations.

The importance of graphical representations for managing large and complex data domains -- such as those imposed by parallel debuggers -- has been dealt with elsewhere (e.g., [18, 21, 19, 24]). Well designed graphical displays can integrate substantial amounts of detail without sacrificing intelligibility. They capitalize on the fact that humans are visually oriented, and especially adept at recognizing visual patterns and deviations from those patterns. As Tufte [22] has demonstrated, visual displays can make quantitative information much more intelligible by

- (1) making large data sets coherent,
- (2) reflecting both the statistical and the logical nature of the data,
- (3) revealing data at varying levels of detail, and
- (4) encouraging the eye to compare and contrast elements.

Although all these characteristics affect the usefulness of a parallel debugger, the last two transcend display techniques to play key roles in managing user interaction.

The availability of a graphical interface platform, whether or not visualization techniques are employed, makes it possible to incorporate direct manipulation techniques. These offer several advantages (cf. [8, 6, 18, 13]):

- The "control language" which must be learned to operate the software tool is reduced.
- Direct manipulation controls are cited by users as being more enjoyable to learn and use than textual languages.
- Common typing errors are eliminated.
- The number of physical actions required to perform each operation is reduced.
- There are fewer opportunities for syntax errors.
- Since operations are selected by recognition rather than recall, a slower delay period intervenes before action initiation.

When visualization is present, the combination of direct manipulation with graphical representations further enhances the user environment. Individual operations are more intuitive, since the user no longer needs to make a conscious correlation between information displayed graphically and arbitrary textual strings. There is some evidence that this reduction in cognitive load reduces the number of semantic errors [8, 26, 16].

A user interface can implement direct manipulation techniques at several levels of implementation. For our purposes, each interaction mechanism can be categorized according to the directness of its physical and logical support. This taxonomy is summarized in Table 1.

Table 1. Direct manipulation mechanisms, classified by directness

	<i>Physical Directness</i>	<i>Logical Directness</i>
least direct		cascaded menus
	pointer device	menu
	touch screen	button
most direct	virtual reality glove	graphical object

Physical directness: The purest form of direct manipulation occurs when the user's hand is moved within a *virtual reality glove* to mimic the manipulation of a physical object. A less tactile interaction is achieved when a *touch-screen* registers the positioning of the user's hand over elements displayed on the screen. Alternatively, a *pointer device* can be employed to redirect hand motion so that cursor movement follows the general direction and proportion of user movements.

Clearly, specialized hardware is required to implement virtual reality and touch-screen mechanisms. At the present time, such facilities are inordinately expensive and somewhat disappointing in terms of reliability. The remainder of this discussion therefore concentrates on pointer-based mechanisms. To simplify terminology, references to "mouse" actions should be interpreted as generic; that is, applicable to any standard pointer device (mouse, tracking ball, joystick, stylus, etc.).

Logical directness: Manipulation is most direct when the user is permitted to "grab" a *graphical entity* from the display and move, resize, or otherwise affect its graphical attributes, thereby causing a change in the logical entity which it represents. Interaction is less direct when the user manipulates a *control button*, causing a command or function to be applied to the logical entity; the display is then updated to reflect the effects of the operation. At a still lower level of directness, the user selects a button in order to display a *menu list*, from which an item is chosen, causing application of a command, etc. Cascaded menus or popups reduces the level of directness even more.

The use of buttons and menus to manage interaction has been well established, and is often dictated by the style policy guidelines of the windowing platform (e.g., [10, 20]). Iconic

buttons, which are simply a graphical extension of text-string buttons, do not raise the level of logical directness, although they can improve recognition in situations where multiple words would be required to represent the command textually [16]. For this reason, the examples given in subsequent sections focus on direct manipulation of graphical objects projected in some larger representational framework rather than icons *per se*.

Direct Manipulation to Control Order of Display

Once visualization has been employed to present of debugging information, graphical techniques can be extended so that the user manipulates the representational images displayed on the screen. Unlike more traditional ways of supporting user choices -- via selectable buttons and menus which make use of textual labels -- this approach allows the user to control the progress of debugging without the need to mentally interpret and apply arbitrary word sequences, numeric identification codes, etc.

Consider, for example, how the speed and direction of execution are controlled in a parallel debugger. A command-driven debugger, such as Intel's IPD [7], requires that the user correctly recall and apply one of several commands (run, rerun, step, halt, continue, etc.) and their syntactic variations (e.g., "step -i" vs. "step -c"). The adoption of a window-based interface simplifies this through the provision of labeled buttons or menu lists. In CONVEX's CXdb [2], for example, a row of button controls offers faster interaction sequences for functions like those of IPD. In this case, the user moves the mouse to position the cursor over the button, then presses a mouse button indicating his or her selection. (Note that in a typical debugging scenario, the user subsequently must position the cursor over another button in order to discontinue execution.)

Button- and menu-based interfaces offer the advantage that user interaction is more economical than typed commands, in terms of the number of physical movements required. What's more, opportunities for errors are reduced since the user is no longer responsible for syntax. Semantic errors can also be minimized, by de-sensitizing buttons or menu items when their selection would be inappropriate. However, constant mouse movement and fine-grained positioning is required -- particularly when buttons and menus are located at opposite extremes of the debugger window (a design policy typical of most graphical interface platforms).

Such motions can be streamlined through the incorporation of graphical direct manipulation techniques, which invest mouse actions with explicit control over functionality. At the simplest level, mouse buttons may be employed to shortcut menu or button selections. In the trace-based debugger illustrated in Figure 1, for example, button presses replicate the actions of

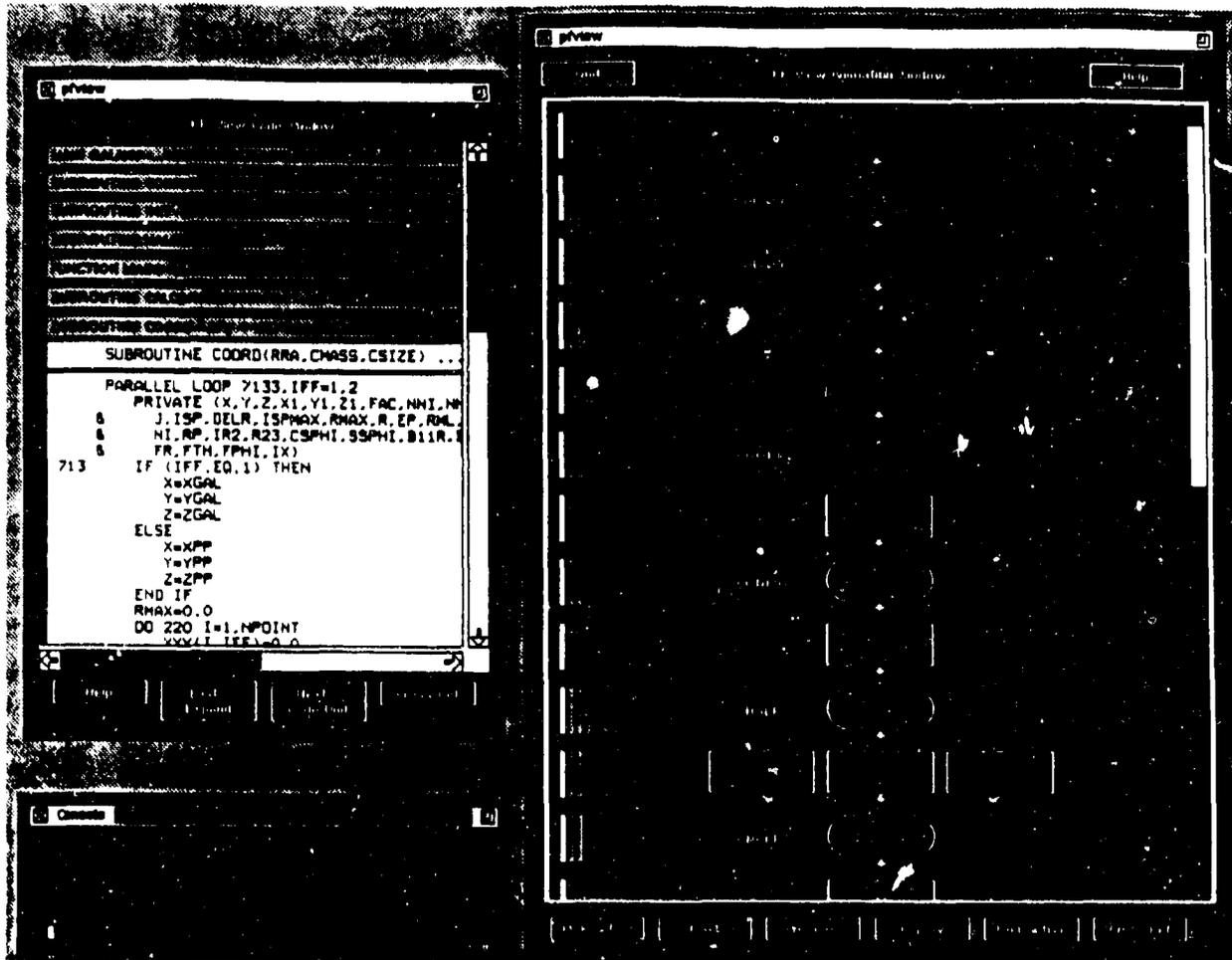


Figure 1. High-level execution replay from PF-View [25].

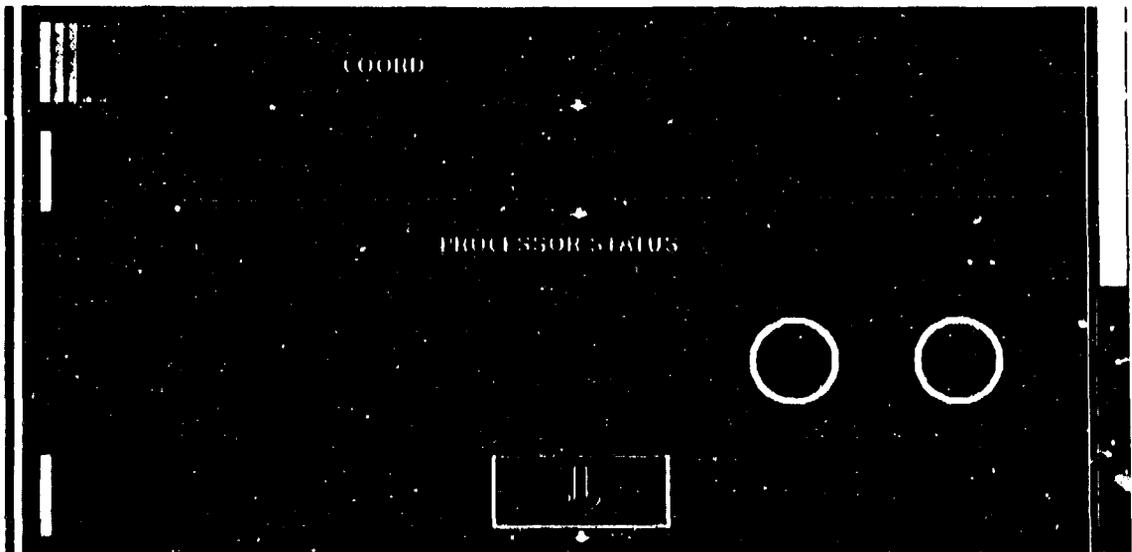


Figure 2. Low-level execution replay from PF-View [25].

the buttons located at the bottom of the window. To step forward through the program, the user can either activate the "next" control or click the lefthand mouse button anywhere in the display area. Backward movement is specified similarly, via the "previous" control or the middle mouse button.

Direct Manipulation to Control Level of Detail

The concept can be expanded to provide more manipulative power. The PF-View tool permits the user to change the level at which program events are animated by positioning the cursor on a higher-level icon and clicking the righthand mouse button. Figure 2 illustrates the effect of "expanding" a high-level parallel loop representation to reveal more detail about its execution. In this example, six processors were participating in execution of the loop (green circles), but two became suspended (red circles with icons) as they tried to gain access to shared variables guarded by locks. The white outlines -- indicating the holder of the lock and all current contenders -- appeared when the user clicked the middle mouse button over one of the suspended processors to gain access to even lower-level information. Had the user clicked instead in the background area of the expanded display, a return to the high-level animation would have been effected.

Direct manipulation can also be used to control the status of debugging filters or other controls. Figure 3 presents a sample display from a prototype breakpoint-style debugger for the Intel iPSC/860. A closeup (Figure 4) shows how the status of processor nodes is animated: here, nodes are arranged in a mesh, but the topology can be altered to reflect the logical communications patterns of the program under study. The color of each element changes during execution to reflect the execution state of the corresponding node. Moreover, as the display legend indicates, the user is free to click on processors directly with the mouse. Selecting a node's graphical image brings it in or out of the current focus of interest, thereby controlling the nature and quantity of debugger information reported during execution. In practice, the graphical mechanism means that the user is no longer forced to memorize (a) arbitrary process numbers (assigned by the previous debugger without regard to communications patterns), or (b) the set of those numbers which form the current context for applying debugger commands.

Once direct graphical control is introduced, rubber-banding capabilities can be added to streamline the specification of debugger operations. In the XIPD prototype user, for example, the user can press and hold the mouse button while dragging the cursor across the display. An outline appears which can be manipulated to encompass the desired number of nodes. When the button is released, the operation is applied to all nodes contained within the outline area. To

```

31:printf( out_left.message, "Converge to node %d", my_node);
32:printf( out_right.message, "Distribute from node %d", my_node);
33:out_left.From = out_right.From = my_node;
34:
35:if ( my_node & 0x01)
36: {
37:   count( 0, out_left, sizeof( Thready), send_to_left, 0);
38:   count( 0, out_right, sizeof( Thready), send_to_right, 0);
39:   create( 0, in_one, sizeof( Thready));
40:   create( 0, in_two, sizeof( Thready));
41: }
42:else
43: {
44:   create( 0, in_one, sizeof( Thready));
45:   create( 0, in_two, sizeof( Thready));
46:   count( 0, out_left, sizeof( Thready), send_to_left, 0);
47:   count( 0, out_right, sizeof( Thready), send_to_right, 0);
48: }
49:
50:printf( "Node %d: message one is %s\n", my_node, in_one.message);
51:printf( "Node %d: message two is %s\n", my_node, in_two.message);
52:

```

Figure 3. Prototype user interface from XIPD [12].

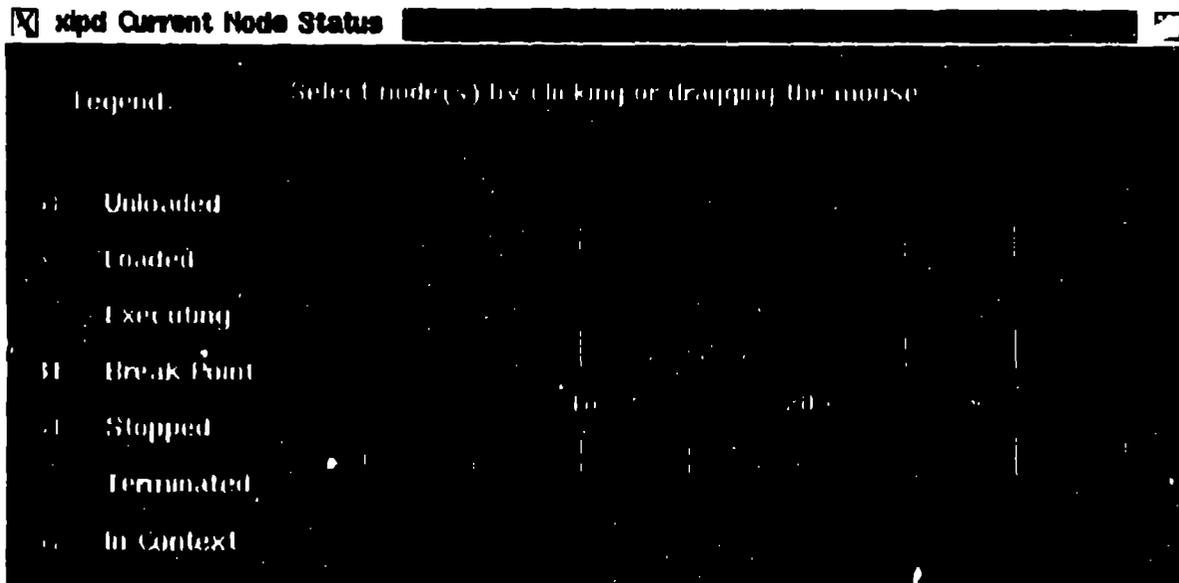


Figure 4. Process status display is manipulatable in XIPD [12].

provide consistency as well as ease-of-use, the same type of node diagram is used to manage several XIPD operations:

- to specify what nodes should be loaded with a given executable
- to add/remove behavior reporting filters
- to control the set of messages to be reported
- to select what processes should be killed

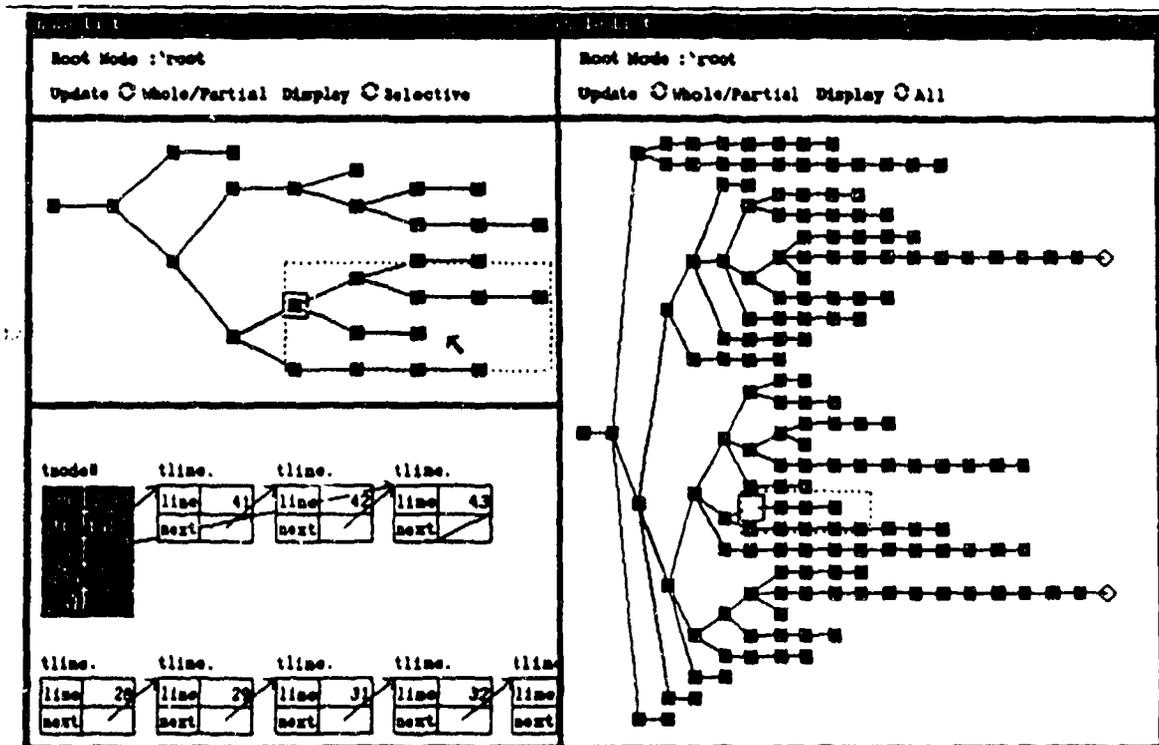


Figure 5. Navigating data structures in VIPS [17].

Direct Manipulation to Control Program Values

Another application of direct manipulation is to support the navigation of data structures. To date, serial debuggers (e.g., [17, 15]) have provided more flexible data traversal mechanisms than do parallel tools, but the same techniques apply in both cases. The example in Figure 5 illustrates how arbitrary linked data structures can be portrayed for examination and interactive update. The user first views a high-level (i.e., iconic) representation including all nodes in the tree or graph. Selecting a node icon with the mouse results in a lower-level display of the

portion of the tree immediately surrounding that node. The data is visible in this representation and can be edited interactively. It is also possible to view selectively just a portion of the graph by indicating particular pointer values (the highlighted boxes at the lower left in Figure 5); a special window shows just the portion of the list controlled by those pointers.



Figure 6. Graphical control of message queues in XIPD [12].

The graphical mechanisms become even more expressive when the user can grab elements and move them or delete them from the display, provoking a corresponding change in the underlying program data. In a parallel debugging environment, such techniques allow the user to examine and control interprocessor messages. Figure 6 portrays another display from XIPD, this time reporting the contents of message queues in terms of source node, destination node, and whether the node is blocked pending a send or a receive operation. By clicking on one of the message symbols (top display), the user can popup a supplementary window with message specifics. Depressing the control key while the message icon is clicked instructs the debugger to delete the pending message from the queue. (A two-handed control sequence was chosen so that messages would not be deleted inadvertently.)

Conclusions

Direct manipulation techniques can yield significant benefits for parallel debuggers. Graphical displays accommodate the volume and complexity of program behavior data much better than their textual counterparts. The addition of direct manipulation enhances user interaction even more. Such facilities support faster operation than do keyed sequences. They also sidestep many opportunities for syntactic and semantic errors, thereby maximizing interface effectiveness.

It should be noted that the examples presented here are from prototype tools developed in research environments. To date, the so-called graphical debugging tools marketed by parallel computer vendors do not exploit graphical direct manipulation. They employ window-based platforms to interact with the programmer, but the information displayed within the windows is textual (though multiple fonts, reverse-video, or other highlighting techniques may be utilized) and user interaction is managed through menus and pushbutton controls, labeled with text strings.

The examples discussed also fail to reflect the full range of possibilities for direct manipulation. Parallel debuggers are not keeping pace with other interactive software in their use of the new technology. Tools for visualizing scientific data, for example, provide more flexible mechanisms for editing and reformatting graphical layouts and also offer interactive graphical languages for specifying how the raw data should be processed for display (cf. [23, 5, 4]). Program development environments have progressed even further; they now rely on languages and approaches that are inherently visual, not just graphical translations of textual systems. Such concepts have not yet been adapted to parallel debugging tools.

To yield substantial benefits for tool developers, the new techniques must be assessed from cognitive as well as ergonomic perspectives. Clearly, direct manipulation mechanisms can only be effective if the graphical representation and the ways in which it is manipulated correspond well to the user's mental model of program and debugger behavior [9, 8, 13]. If a new, visual language must be committed to memory, graphical control will be frustrating and even counterproductive. Just as in debugger visualization, the challenge is to arrive at mechanisms that are both intuitive and fast.

References

1. Ambler, Allen L., and Margaret M. Burnett: *Influence of Visual Technology on the Evolution of Language Environments*. IEEE Computer, 22 (10): 9-22 (1989).
2. CONVEX Computer Corporation: *Convex CXdB User's Guide*. Convex Press, 1991.

3. Curtis, Bill: A Review of Human Factors Research on Programming Languages and Specifications. Proc. Human Factors in Computer Systems, pp. 212-218 (1982).
4. Dickinson, Robert R., Richard H. Bartels and Allan H. Vermeulen: The Interactive Editing and Contouring of Empirical Fields. IEEE Computer Graphics & Applications. 9: 34-43 (May 1989).
5. Dyer, D. Scott: A Dataflow Toolkit for Visualization. IEEE Computer Graphics & Applications. 10, 60-69 (July 1990).
6. Hutchins, E. L., J. D. Hollan and D. A. Norman: Direct Manipulation Interfaces. In User Centered System Design: New Perspectives on Human-Computer Interaction, ed. D. A. Norman and S. W. Draper. Lawrence Erlbaum Associates (1986).
7. Intel Supercomputer Systems: iPSC/2 and iPSC/860 Interactive Parallel Debugger Manual. Intel Corporation, 1991.
8. Morgan, K., R. L. Morris and S. Gibbs: When Does a Mouse Become a Rat? Or Comparing Performance and Preferences in Direct Manipulation and Command Line Environments. The Computer Journal. 34 (3): 267-271 (1991).
9. Norman, Donald A.: Some Observations on Mental Models. In Mental Models, ed. D. Gentner and A. Stevens. Erlbaum Associates (1983).
10. Open Software Foundation: OSF/Motif Style Guide. Prentice Hall (1991).
11. Pancake, Cherri M.: Software Support for Parallel Computing: Where Are We Headed? Communications of the ACM. 34 (11): 52-64 (1991).
12. Pancake, Cherri M.: Visual Techniques for Breakpoint-Style Parallel Debuggers. In preparation.
13. Pancake, Cherri M.: Graphical Support for Parallel Debugging. To appear in NATO Advanced Research Workshop on Software for Parallel Computation, ed. J. Kowalik. Springer-Verlag.
14. Pancake, Cherri M. and Sue Utter: Debugger Visualizations for Shared-Memory Multiprocessors. In High Performance Computing II, ed. M. Durand and F. El Dabaghi, pp. 145-158. Elsevier Science (1991).
15. Pazel, D. P.: DS-Viewer: An Interactive Graphical Data-Structure Presentation Facility. IBM Systems Journal. 28 (2): 307-323 (1989).
16. Rohr, Gabriele: Understanding Visual Symbols. Proc IEEE 1984 Workshop on Visual Languages, pp. 184-191 (1984).
17. Shimomura, Takao and Sadahiro Isoda: Linked-List Visualization for Debugging. IEEE Software. 6: 44-51 (May 1991).
18. Smith, Randall B: The Alternate Reality Kit: An Animated Environment for Creating Interactive Simulations. Proc. IEEE 1986 Workshop on Visual Languages, pp. 99-106 (1986).

19. **Smith, W. J. and J. E. Farrell: The Ergonomics of Enhancing User Performance with Color Displays. Proc. Society for Information Display, Vol. 2, pp. 5.1.1-5.1.16 (1985).**
20. **Sun Microsystems, Inc. OPEN LOOK Graphical User Interface Application Style Guidelines. Addison-Wesley (1990).**
21. **Tufte, Edward R.: Envisioning Information. Graphics Press (1990).**
22. **Tufte, Edward R.: The Visual Display of Quantitative Information. Graphics Press (1983).**
23. **Upson, Craig *et al.*: The Application Visualization System: A Computational Environment for Scientific Visualization. IEEE Computer Graphics & Applications. 9: 30-42 (July 1989).**
24. **Utter, Sue and Cheri M. Pancake: Advances in Parallel Debuggers: New Approaches to Visualization. Cornell Theory Center Technical Report CTC89TR18 (1989).**
25. **Utter-Honig, Sue and Cheri M. Pancake: Graphical Animation of Parallel Fortran Programs. Proc. Supercomputing '91, pp. 491-500 (1991).**
26. **Woods, David D.: Visual Momentum: A Concept to Improve the Cognitive Coupling of Person and Computer. International Journal of Man-Machine Studies. 21: 229-244 (1984).**

Direct Manipulation Techniques for Parallel Debuggers

Cherri M. Pancake
(Visiting Scientist, Cornell Theory Center)
Department of Computer Science
Oregon State University

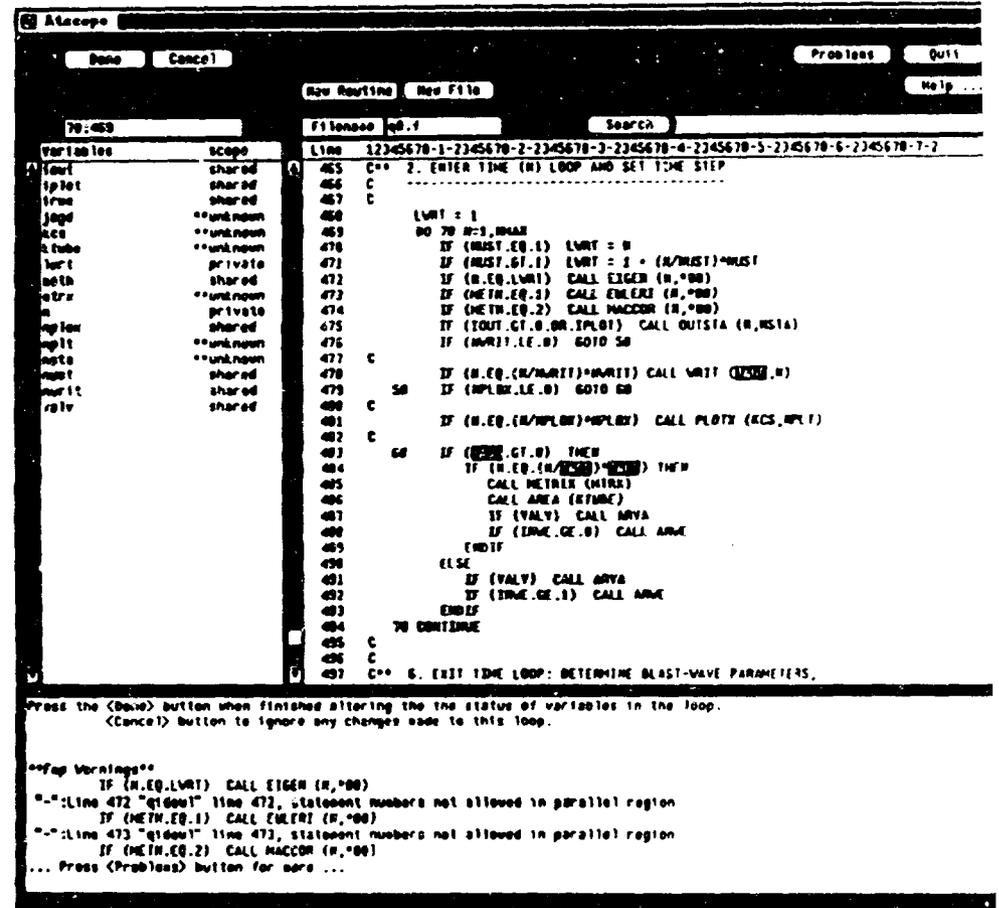
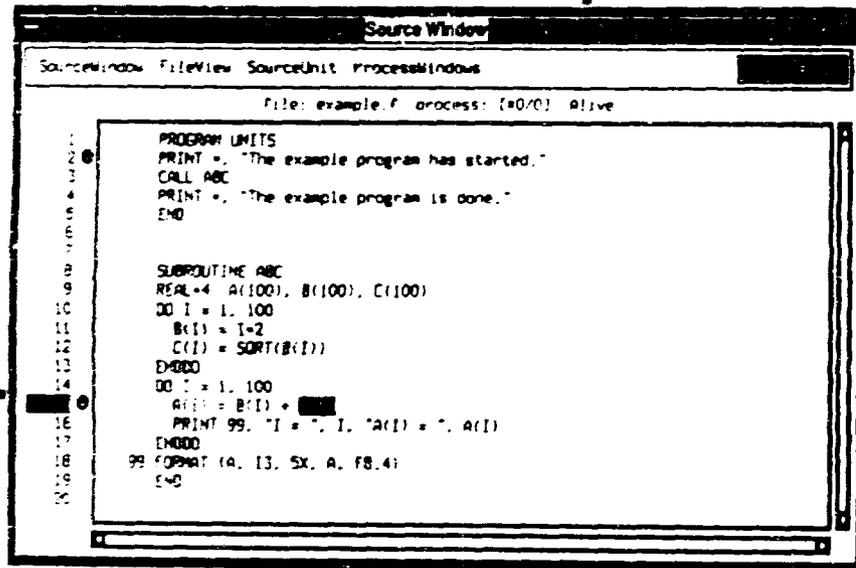
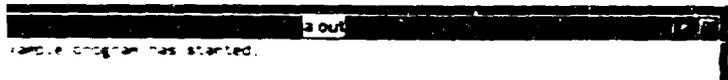
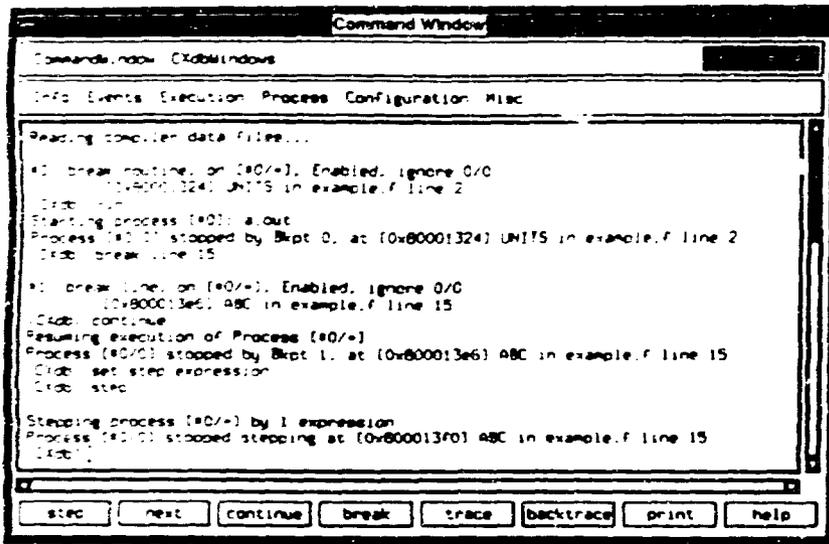
How Useful Are Today's Parallel Debuggers?

- **Complaints from the user community**
 - "too hard to learn"**
 - "tedious to use"**
 - "the information is microscopic"**
 - "won't give me the information I really need"**
 - "too hard to make sense of the data"**
 - "doesn't really help me find the errors"**
- **Recent survey of distributed-memory MIMD system users**
 - 80% have never even tried to use the parallel debugger available**
 - 90% still rely primarily on hand-coded instrumentation**
- **A number of users have developed their own specialized tools**

Challenges for the Parallel Tool Developer

- **Technological challenges:** stabilize an inherently unstable environment
intrusiveness problem
nonreproducibility problem
- **Data reduction challenges:** reduce execution data to manageable size
filter out redundant or unnecessary data
extrapolate higher-level "events" from low-level data
- **Cognitive challenges:** present information in meaningful way
must relate to programmer's concept of program
must be clearly applicable to task(s) at hand
should be straightforward to learn
operations should be intuitively obvious from displayed info
- **Ergonomic challenges:** make debugger use efficient physically
should minimize number of keystrokes
should minimize possibilities for manual errors
should minimize amount of mouse motion required

"Graphical" Debuggers (Convex CXdb, Cray ATscope)



How Graphical Techniques Help

- **Graphical techniques: non-textual in nature**
 - rely on graphical attributes (shape, color, screened texture, etc.)**
 - figurative or symbolic representation of objects, characteristics**
 - iconic, plotted, rendered elements**
 - "iconic words"**
- **Graphical displays of quantitative data**
 - can make large data sets coherent**
 - can reveal data at varying levels of detail**
 - can reflect both the statistical and the logical nature of the data**
 - can encourage the eye to compare/contrast elements**
- **Graphics can also be used to manage user interaction**

Graphical Interactions with the User

- **Direct manipulation mechanisms**
 - require mouse or other pointer device**
 - user manipulates graphical objects displayed on screen**
- **Direct manipulation can support many debugger activities**
 - control over direction and speed of execution**
 - control over level of information presented**
 - control over amount of information presented**
 - control over contents of program values**
- **Advantages**
 - smaller "control language" --> reduced learning time**
 - fewer physical actions required to perform operation**
 - recognition rather than recall --> slower action initiation delay**
 - elimination of common typing errors**
 - fewer opportunities for syntax errors**
 - no display/text correlation --> reduced cognitive load**
 - reduced cognitive load --> fewer semantic errors**
 - users claim they are more enjoyable than text-based techniques**

How "Direct" Is Manipulation?

<i>Logical Directness</i>		<i>Physical Directness</i>
cascaded menus menu button graphical object	least direct	pointer device touch screen virtual reality glove
	most direct	

Controlling Program Execution (Intel IPD)

```
(3:0) > context (all:0)
(all:0) > break gauss.f()#175
(all:0) > b
(all:0)
```

Bp #	Type	File name	Procedure	Breakpoint Condition	Bp context
1	C Bp	gauss.f	shadow	Line 175	(all:0)

Now you enter `run` followed by `wait`. The `run` command starts the program from the beginning; `wait` displays user process information when it reaches the breakpoint.

```
(all:0) > run ; wait
Context          State          Reason          Src/Obj Name    Procedure        Location
-----          -
*(all:0)         Breakpoint    C Bp 1          gauss.f         shadow           Line 175
```

Now, once again, display the current value of `nbrnodes`, and then, on node 3, use the `assign` command to temporarily reassign the value of `nbrnodes` to 3, instead of 4.

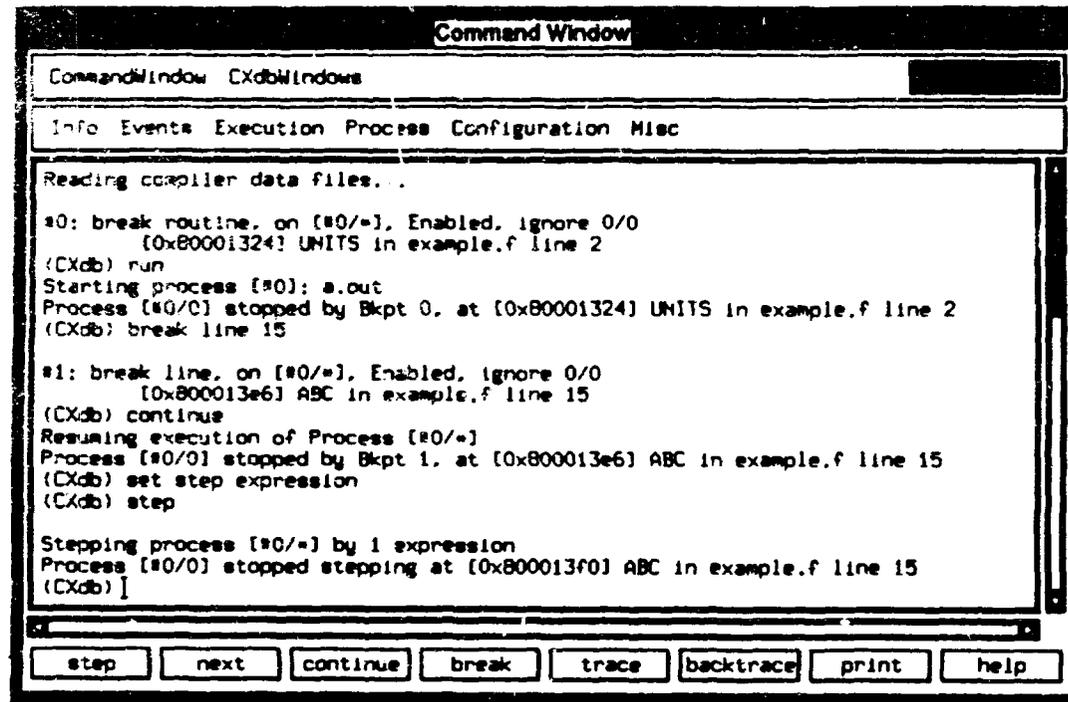
```
(all:0) > disp nbrnodes

** gauss.f()shadow()nbrnodes **
***** (all:0) *****
nbrnodes = 4

(all:0) > context (3:0)
(3:0) > assign nbrnodes=3
(3:0) > disp nbrnodes

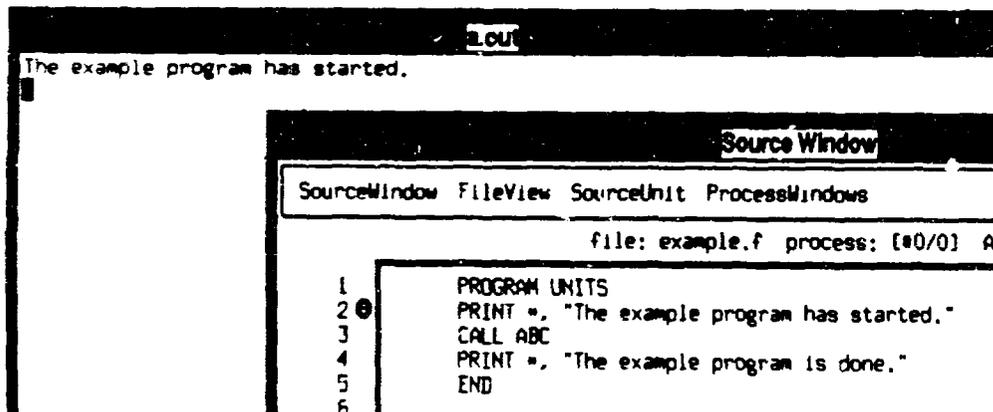
** gauss.f()shadow()nbrnodes **
***** (3:0) *****
nbrnodes = 3
```

Controlling Program Execution (CONVEX CXdb)



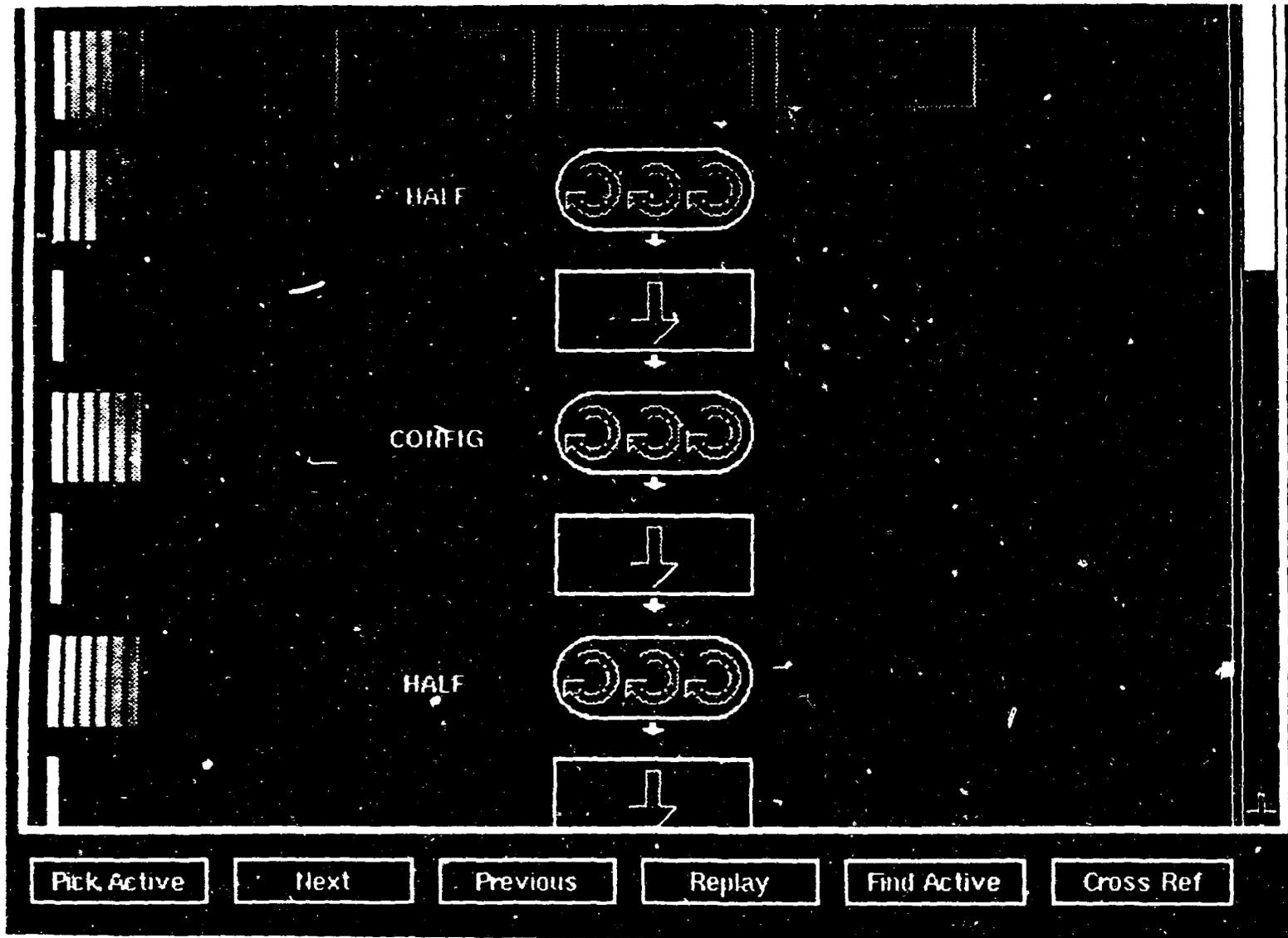
```
Command Window
CommandWindow CXdbWindows
Info Events Execution Process Configuration Misc
Reading compiler data files...
#0: break routine, on [#0/*], Enabled, ignore 0/0
    [0x80001324] UNITS in example.f line 2
(CXdb) run
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80001324] UNITS in example.f line 2
(CXdb) break line 15
#1: break line, on [#0/*], Enabled, ignore 0/0
    [0x800013e6] ABC in example.f line 15
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 1, at [0x800013e6] ABC in example.f line 15
(CXdb) set step expression
(CXdb) step
Stepping process [#0/*] by 1 expression
Process [#0/0] stopped stepping at [0x800013f0] ABC in example.f line 15
(CXdb) ]
```

step next continue break trace backtrace print help

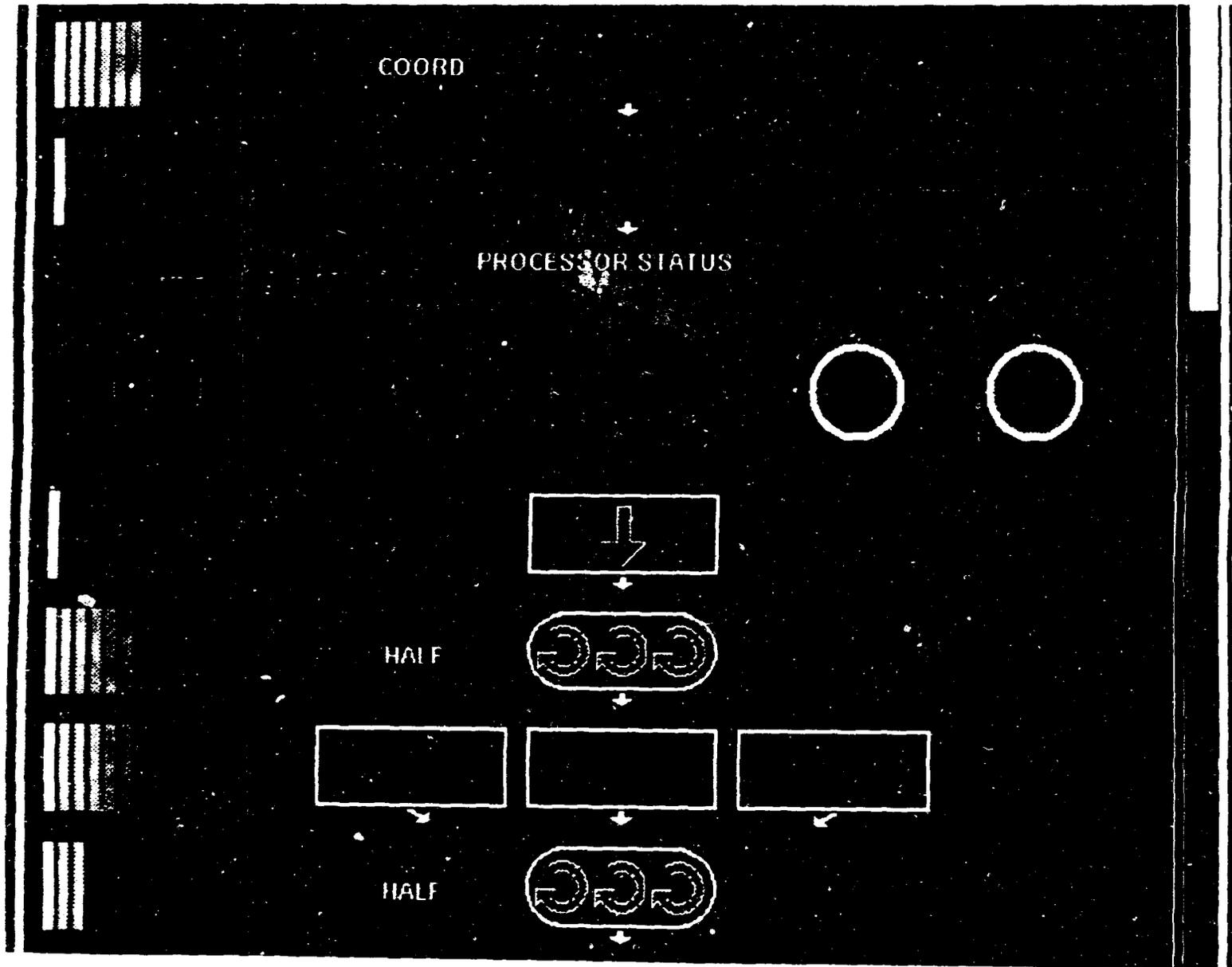


```
Source Window
SourceWindow FileView SourceUnit ProcessWindows
file: example.f process: [#0/0] Alive
1 PROGRAM UNITS
2 PRINT *, "The example program has started."
3 CALL ABC
4 PRINT *, "The example program is done."
5 END
6
```

Controlling Direction/Speed of Execution (PF-View)



Controlling Level of Information (PF-View)



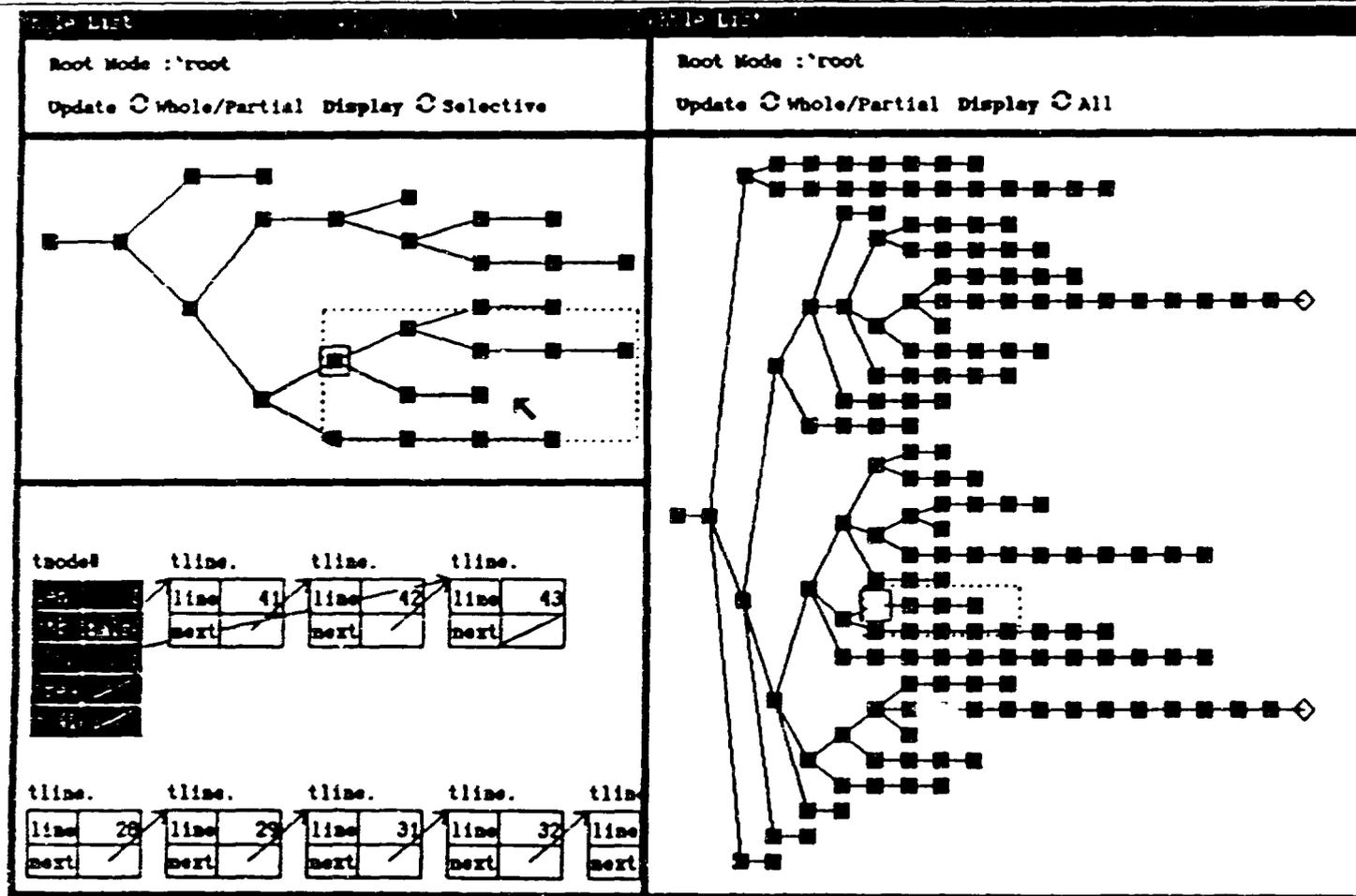
Controlling Amount of Information Presented (xipd)

xipd Current Node Status 

Legend: Select node(s) by clicking or dragging the mouse

	Unloaded								
	Loaded								
	Executing								
31	Break Point								
	Stopped								
	Terminated								
	In Context								

Navigating Data Structures (VIPS)



Controlling Message Exchanges (xipd)

3 original messages

Ctrl-click to delete send or receive

```
graph LR; subgraph Node1; N1((1)); N2((2)); N3((3)); end; subgraph Node2; N4((4)); N5((5)); N6((6)); end; N0((0)); N1 --> N3; N1 --> N4; N2 --> N5; N2 --> N6; N0 --> N0;
```

Change viewpoint by clicking or dragging

- all communications
- sends only
- receives only

Message Queue Information

Source: 1 Destination: 4

Type `ou01` Length 2

Text 23

Show As

- decimal
- hex
- ASCII

OK Cancel

Conclusions

- **Direct manipulation techniques can improve parallel debugger usability**
 - exploit pattern recognition capabilities**
 - minimize hand movements to perform complex operations**
 - displays can be manipulated directly rather than indirectly**
 - sidestep many opportunities for syntactic/semantic errors**
- **Open areas for further research**
 - flexible mechanisms for editing/reformatting graphical layouts**
 - graphical specifications of how raw data should be processed**
 - approaches that are inherently visual – not translated from text**
- **Dual goals: must meet cognitive as well as ergonomic needs**

Problem Areas

- **Direct manipulation must be assessed according to ergonomic value**
- **Must also count cognitive costs**
- **Effective only if**
 - graphical representation corresponds to mental model**
 - ways it is manipulated correspond to mental model**
- **Consistency is critical**
 - (example) xipd uses same graphical manipulations to control**
 - where to load given executable**
 - add/remove behavior reporting filters**
 - domain of message info reporting**
 - which processes should be killed**

Transparent Observation of XENOOPS Objects

S. Bijnens, W. Joosen, P. Verbaeten
Department of Computer Science K.U.Leuven
Celestijnenlaan 200A
3001 Leuven
Belgium
e-mail: stijjn@cs.kuleuven.ac.be

Keywords : transparent debugging, object-oriented parallel systems, distributed memory multiprocessors, reflection, meta-objects.

Within our research team we are building XENOOPS, a prototype execution environment for distributed memory multiprocessors which supports the development of complex parallel applications.

The construction of such applications requires the use of a tool to debug the various components of an application and to observe their behaviour (interaction) on a distributed memory computer. In this paper, we will outline the basic concepts behind our debugging tool, and mainly focus on the mechanism to realise a transparent and dynamic observation of object-oriented parallel applications.

Our approach is based on the concept of computational reflection, which leads to a clean separation between application and observation aspects. The code that realises application dependent observation will be generated by a preprocessor for C++.

1. Introduction

XENOOPS, an acronym for an eXecution ENvironment for Object Oriented Parallel Software[1], supports the development of adaptive parallel applications. Such applications correspond to parallel computations in which the workload distribution changes as the computations evolve. In this context dynamic load balancing can realise a relevant performance gain by reducing imbalances in the workload distribution as they occur. It is our believe that the production of parallel software for distributed memory computers will be accelerated if application writers adopt the benefits of the object-oriented methodology. Object-oriented programming organizes programs as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

This paper will focus on the basic mechanism to realise a transparent and dynamic observation at the object level. By transparent we mean that no changes to the application code are required, and that the semantics of the application code are -apart from performance- not influenced. By dynamic we mean that the debugger may be activated at any time, without restarting the application.

Section 2 describes the XENOOPS model and illustrates some of its specific advantages like reusability with regard to load balancing.

Section three discusses the required debugging functionalities within an object oriented framework, which is essentially different from low level event tracing.

Our debugger design is based on the concept of computational reflection[2], which is described in section four. In contrast to the common ad hoc 'print statement insertion' debugging techniques, we use the concept of reflection to develop a modular and transparent way of adding debugging and/or monitoring functionalities to a system. The notion of reflection provides an extra dimension of abstraction (meta-abstraction) that complements data abstraction and super-abstraction. In object-oriented systems meta-

Section 5 treats the realisation of the proposed mechanism by describing a preprocessor which generates application dependent observation code for XENOOPS applications in C++. To illustrate the proposed concepts, we describe an example from the domain of computational fluid dynamics (CFD) [4].

In section 6 we will illustrate some specific debugging policies that can easily be realized with the proposed mechanism.

Section 7 will describe the support that is offered by XENOOPS to realize the mechanism. Section 8 will compare our approach to dynamic debugging with related work. We summarise in section 9.

2. The XENOOPS model

The primary objects of our system are *work units*. Each of them encapsulates a fraction of the work to be executed by the application¹. These objects are mobile to enable a load balancing system to reallocate work at run time. Object migration[5] is initiated by the invocation of the *Migrate* operation on a work unit. Basically, the *Migrate* operation uses *Pack* and *UnPack* operations provided by the programmer, to achieve efficient migration while respecting the semantics of the applications data. Two additional operations (*Split* and *Join*) provide a mechanism to control object granularity: the *Split* operation will divide a work unit into several migratable units, the *Join* operation on a group of work units will consolidate them into one.

On each node, the work units are stored in an *object table*. Two active objects (objects with an own thread of control) manipulate this table:

- First, the *Calculator* executes the typical application code by selecting a work unit from the table, then performing an operation (for example one iteration step in a CFD application[4]), sending results and/or waiting for results from other work units, and finally putting the work unit back in the table. This component corresponds to the application's algorithm and will be provided by the application programmer.
- The second active object is the *load manager*, which runs simultaneously with the calculator. Dynamic load balancing will be realised by transparently migrating work units between nodes.

Figure 1 illustrates the XENOOPS model.

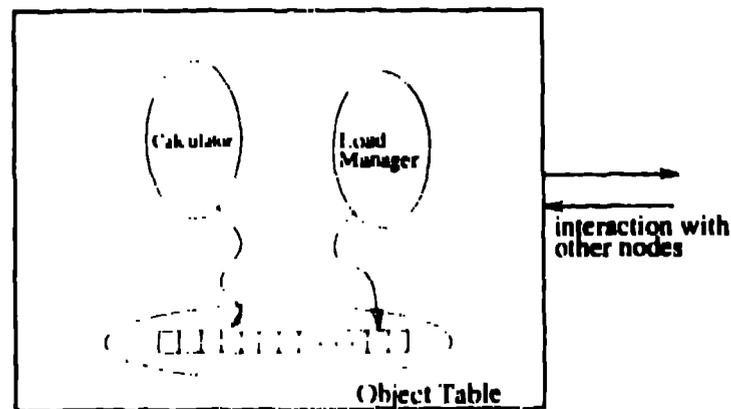


Figure 1 XENOOPS objects in a single node

¹ For instance, in the case of data parallel applications like CFD, a work unit corresponds to the data of a fraction of the discretised physical domain.

balancing component and simplifies the task of the application writer if he wants to integrate, test and optimise load balancing strategies for a particular problem. Disabling the load balancing component can be done in a straightforward way.

3. Debugging functionality

The task of observing the behaviour of parallel applications requires an environment in which to execute potentially errant code under controlled conditions.

In an object oriented system all interaction between objects is accomplished by method invocation. The process of testing object-oriented systems involves two phases. First, *component level testing* is done. Classes (templates for creating objects) are debugged individually. This way the correctness of each operation on an individual object is tested. Secondly, the integration of the individual components requires testing of their interaction. In particular, method invocations and their effect on state changes of interacting components must be controlled. We will focus on the second phase, because the first one can be accomplished by classical sequential debuggers.

Debugging functionalities can be classified by considering that debugging involves several agents and can be expressed in terms of interfaces between these agents. The *User* initiates a debugging session with a *System* that must be debugged. A *Tool* must realise the above as unobtrusively as possible.



Figure 2: Debugging

The interface between the user and the tool specifies the debugging functionality. Several approaches exist[6]: post-mortem debugging, instant replay, flowback analysis[7] and others. These functionalities are often integrated with a graphical interface.

The interface between the tool and the system needs a mechanism to realise the required debugging functionality.

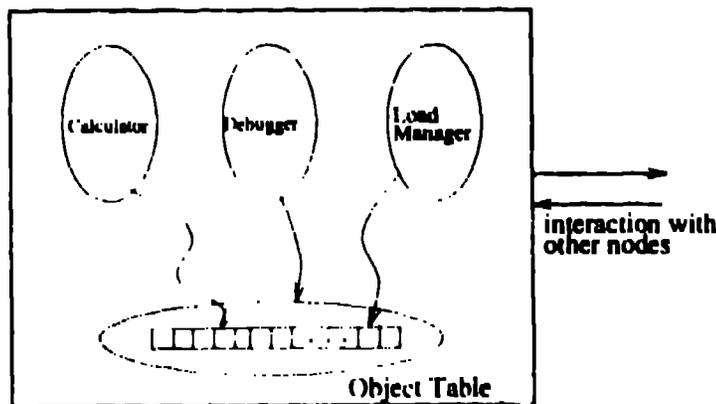


Figure 3: Debugging in XENCOOPS

The interaction between the tool and the XENCOOPS run-time system is accomplished by adding on each node a third active component : a debugger. This debugger can control and manipulate work units, stored in the object table. The mechanism for achieving this interaction in a transparent way will be described in the next section. This mechanism can be used to realise a broad spectrum of functionalities.

added to the system in a transparent way. Finally, if debugging is required another active component can be added (dynamically) without any modification of the other two components. In the next section we describe the mechanism for achieving this transparency

4. Modelling transparent debugging as reflective computation

Most computational systems exhibit not only object-computation, i.e. computation about their problem domain, but also reflective computation, i.e. computation about their own execution. Examples of reflective computations are :

- the gathering of performance statistics,
- the collection of information for debugging purposes, stepping and tracing facilities,
- self-optimisation, self-modification and self-activation.

The decomposition of a computation into object-computation and reflective-computation introduces more *modularity* into computational systems. The computation at the object-level manipulates data representing the problem domain. The computation at the reflective level takes care of the internal organisation of the computational system and its interface to the outside world. It manipulates data representing the actual object-level computation.

In [2], the concept of a meta-object is introduced: a meta-object is an object that controls and manipulates another object. In other words, a meta-object is an object 'about' another object, able to observe and to control it. Reflection is achieved by setting up a 'causal connection' between the meta-object and its corresponding object. This means that the meta-object and the object are linked in such a way that a change in one of the two leads to an effect upon the other.

Within the XENOOPS system, a meta-object will be created dynamically for every object, that's worth monitoring. We apply this rule to work units, as they are the key objects, and the *User* is especially interested in their behaviour. In particular, the *User* is interested in: the operations that are invoked on a specific work unit, the caller and the resulting state changes.

The debugging component will interact with the meta-objects. This interface is specified by a set of operations.

Typical operations invoked by the debugger on a meta-object can be classified as :

1. operations to realise the dynamic creation and destruction of meta-objects:
 - *Constructor(object)* creates a meta-object for a specific object.
 - *Destructor()* destroys a meta-object (when it is not relevant any more for debugging purposes).
2. operations to realise the debugging functionality:
 - *Trace_Method_Invocation()*: to trace method invocations on the object. This operation will intercept all invocations on the object and react in a appropriate way. The meta-object will inform the debugger about the event. The type of information that will be exchanged depends on the debugging functionality (post mortem, instant reply ...). Note that the proposed mechanism is general and does not restrict itself to a specific functionality.
 - *Get_State()* : to read the state of the object.

These operations are automatically generated by the XENOOPS debugging system using compile-time heuristics. For example, an operation like *Get_State()* makes use of the *Pack()* and *Unpack()* operations on an object. These operations are already supplied by the programmer for implementing the *Migrate()* operation (object migration).

This way we are building a debugging system on top of the application's object-system. It is a general and modular framework. The mechanism of meta-objects makes it possible to transparently add debugging

specific parallel application.

5. Realization of the Mechanism

5.1 General approach

A meta-object-class must be generated for every class, from which an object can be instantiated that is worth controlling (e.g. work units). A preprocessor generates automatically a meta-object-class that will be dynamically integrated into the system during a debugging session.

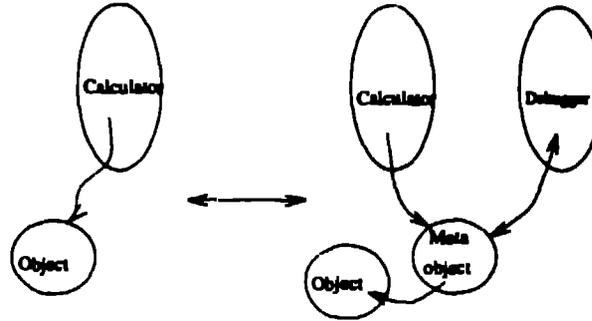


Figure 4: Dynamic debugging

Transparency is achieved by intercepting all relevant invocations called by the object system. (In Figure 4, only the Calculator is illustrated for simplification.) The meta-object will intercept all method invocations, inform the debugger and delegate the invocation to the object itself. The object system consisting of the calculator, the load manager and the object table does not notice the existence of meta-objects: a meta-object behaves like an ordinary object because it offers the same functionality.

We use the inheritance relation to achieve transparency.

A preprocessor realizes the necessary code to achieve transparent debugging:

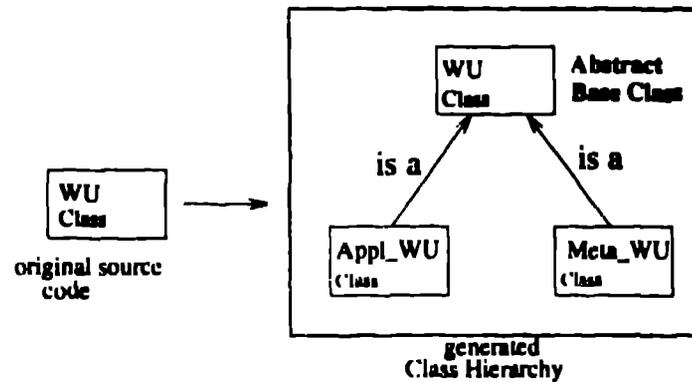


Figure 5. Preprocessor

The preprocessor generates a specific hierarchy. This way we achieve causal connection in a transparent way. The meta work unit object "is a" work unit object (by inheritance). So the other system components (calculator, load manager) only work with objects of type work unit (WU) and will not notice the difference whether the work unit is a real application work unit (Appl_WU) or a meta-object (Meta_WU), as they can invoke the same operations. This way the debugger can intercept all method invocations (through a meta-object) and state changes while no other system component will notice it.

The XENOOPS prototype is implemented in C++[8], and we use language specific features to realise the proposed concepts. The application programmer implements the functionality of work units.

The generated components are :

- the application work unit class (Appl_WU) is a copy of the original work unit class provided by the programmer.
- the meta work unit class (Meta_WU) is generated by the preprocessor and incorporates all debugging functionalities. This class is made a friend[8] of the application work unit class. This way the meta object can access private data members of the work unit itself (casual connection).
- The work unit class (WU) is a virtual base class (abstract class) with the same interface as the application work unit class but all methods are declared as pure virtual[8].

```

class WU{

    //original code for a work unit provided
    //by the CFD application programmer.

private:
    // the discretized physical domain.
    STATE my_data;

    Message Pack();
    void Unpack(Message);
public :
    // Constructor
    WU();

    // Destructor
    ~WU();

    // Iteration over the physical domain
    // that the work unit covers

    void Do_an_Iteration();

    // communication functions that involve
    // the swapping of edge values between
    //neighboring WU

    void Send_Update( WU);
    void Receive_Update(WU);

    // migration
    void Migrate( Node);

    //granularity control
    void Split(int);
    void Join( LList <WU>);

};

```

(Original WU class

```

class Appl_WU : public WU {

    // Only difference with the original WU class
    // is this friend statement, which allows the
    // meta object the access of private data.

    friend class Meta_WU;

private:
    STATE my_data;

    Message Pack();
    void Unpack(Message);

public :
    // Constructor
    Appl_WU();

    // Destructor
    ~Appl_WU();

    void Do_an_Iteration();

    void Send_Update( WU);

    void Receive_Update(WU);

    // migration
    void Migrate( Node);

    //granularity control
    void Split(int);
    void Join( LList <WU>);

};

```

Generated Appl_WU Class

```

private :
    // pointer to the Appl_WU, that
    // must be debugged.

    Appl_WU* my_appl_WU;

public :

    // Constructor (called by the debugger)
    // the parameter of type WU is the Appl_WU
    // that must be debugged.
    Meta_WU( WU);

    // Destructor
    ~Meta_WU();

    // Appl. Specific member functions
    // Called by Calculator or Load Manager
    void Do_an_Iteration();
    void Send_Update( WU);
    void Receive_Update(WU);
    void Migrate( Node); // migration
    void Split(int); //granularity control
    void Join( List <WU>);

    // Debug Specific member functions
    // Called by Debugger:
    void Trace_Method_Invocation();
    void Get_State();
};

```

Generated Meta_WU class

```

// to achieve dynamic binding

class WU {

private :
    // no private members
public :

    // Constructor
    WU();

    // virtual Destructor
    virtual ~WU();

    // all member functions are declared
    // as pure virtual (=0)

    virtual void Do_an_Iteration()=0;
    virtual void Send_Update( WU)=0;
    virtual void Receive_Update(WU)=0;

    // migration
    virtual void Migrate( Node)=0;

    //granularity control
    virtual void Split(int)=0;
    virtual void Join( List <WU>)=0;

};

```

Generated virtual WU class

The object system only expects objects of type WU. Method invocations are intercepted through dynamic binding, which means that method binding happens at run-time depending on the object's actual type. The ability to call a variety of functions using exactly the same interface -as provided by virtual functions- is also called polymorphism.

The meta object intercepts all invocations on the object. A specific debug policy must be specified by the User. The code for the Pre- and Post-actions included by the preprocessor are generated from the specification of the debug policy.

```
void Meta_WU::Send_Update(WU neighbor, DATA msg){
    // This function is called by the Calculator

    // PRE-action
    Pre_Action_Send_update(dest, msg,id);

    // Delegate invocation of the original
    // operation on the Appl_WU
    my_appl_WU ->Send_Update(WU dest);

    // POST-action
    Post_Action_Send_Update_end();
};

void Meta_WU::Get_State(){
    // This function is called by the Debugger
    //if the User wants to see the state of a
    //specific WU.

    // the marshallng of the WU into a message.
    // the pack-operation is already
    //available for realizing migration.
    msg = my_appl_WU->Pack();

    // Send the information to the User
    Send_to_Host(msg);
};
```

Implementation of some Meta_WU methods

Post-Mortem debugging/ Instant replay

This technique is very easy to realize. In the case of Post-Mortem debugging, the Pre- and Post actions just have to register the invocation.

To realize Instant Replay, the task of the Pre-action will be the registration (in the meta work unit) of a history of invocations that include enough information to realize the replay.

Snapshots

The realization of a consistent global snapshot of the distributed computation, essentially requires to find a set of local snapshots such that the causal relation between all events (invocations) that are included in the snapshots is respected. This means that: if an event is contained in the global snapshot and this event is caused by another event, then the latter event must be in the global snapshot too. This causal dependency can be realised with the use of vector clocks[9] included in the meta objects.

The proposed mechanism for transparent debugging with the use of meta objects can only function optimally if XENOOPS offers some advanced support.

7.1 Synchronization

The XENOOPS model provides on each node of the multiprocessor three active objects. Because these objects have their own independent life (thread) and simultaneously interact on shared data (work units), race conditions can occur. For example, when the Calculator has invoked the `Do_an_Iteration()` operation and the Debugger invokes simultaneously the `Get_State()` operation, inconsistent data will be passed to the debugger. Another race condition occurs when the load manager wants to migrate a work unit that is currently selected by the Calculator. These examples show that there *exist* already some synchronization constraints on a work unit's implementation provided by application programmer and that the debugging concept just *inherits* these constraints and *adds* some more.

In the literature, different strategies for handling synchronization constraints exist. One means (currently adopted in XENOOPS) of controlling parallel execution of methods is to specify the allowable control paths through each object (e.g Path Expressions [10]). The purpose of path expressions is to constrain parallel activities, which means they usually impose sequencing rather than indicating parallelism.

Notation	Meaning
<code>m1.m2</code>	<code>m1</code> and <code>m2</code> can be run in parallel
<code>{m1}</code>	0 or more <code>m1</code> in parallel
<code>m1 + m2</code>	<code>m1</code> and <code>m2</code> must execute serially

It should be noted that, because the specific specification policy used, is orthogonal to our debugging concept, we are not limited to one of them. Another way of specifying synchronization constraints can be realized by the use of *synchronization counters* [11].

7.2 Scheduling

Scheduling between the active components must be provided. One might expect active components to be mapped on the processes that are offered by the operating system kernel. The major difference between a MIMD parallel system and a traditional operating system lies in the fact that the processes running on a given node do not really *compete* for the processor, but *cooperate* to improve the global performance of one single application. To realize this cooperation the existing active objects (in our model a fixed set) have at least implicit knowledge about each other.

XENOOPS defines a control hierarchy between active objects :

1. For the case with two active objects (load manager and calculator), we decided to localize the scheduling control in the load manager. This way, the application writer will not have to deal with scheduling in the default case.
2. For the case with a calculator, load manager and debugger, we recursively apply the approach to the given situation: the debugger will control the scheduling between himself and the lower part of the hierarchy (load manager controlling the calculation).

Since the debugger controls the CPU allocation, he can give himself the highest priority if necessary. For example, when the invocation of `Get_State()` operation is blocked because of the `Do_an_Iteration()` has been invoked on a work unit, the debugger will have the highest priority if the `Do_an_Iteration()` operation finishes. This way an invocation of `Get_State()` will always be the nearest consistent state in the future.

The selected control hierarchy between calculator and load manager can be justified by the observation that, when only running the calculator (an unbalanced execution of the application), nothing will have to be specified. Then the calculator is the top of the hierarchy, and will have maximal use of the CPU.

Our work has been partially inspired by research projects in the area of object oriented programming languages (OOPL) and operating systems (OOOS). We will now refer to both.

The dynamic approach of transforming functionalities during the objects' lifetime is fairly commonplace. For example, in nature, a butterfly begins life as a caterpillar, molts into a chrysalis and reappears as a butterfly. In object oriented jargon this process can be stated as 'an object that dynamically changes its type'. In class based object-oriented languages this dynamic behavior can be captured by meta-objects [12].

Another area of study are OOPLs that use delegation based inheritance and do not even support the concept of a class. Upon receiving a message (= method invocation), an object compares it to its known methods. If no match can be found, it 'delegates' responsibility for the message by passing the message to another object (prototype). An object may dynamically select a prototype. This changes the way its messages are processed, effectively changing its 'class'. The language SELF uses prototypical inheritance [13].

The Actor model also supports a 'become statement', which results in the fact that incoming messages can be handled differently. [14].

Some object oriented operating systems also use a dynamic approach for modifying the behavior of resources. The Muse distributed operating system [15] provides an open and self-advancing dynamic environment. Muse provides reflective computing that presents facilities for self-modifying an object with its environment. Objects reside in the context of a collection of meta-objects to handle dynamic system behavior and to provide an optimal execution environment for the object.

Computational reflection is also employed in Choices [16], a family of object-oriented operating systems. For example, when Choices boots, few operating system facilities are available. Therefore the initial heap manager uses a simple algorithm that has few features and places few requirements on the operating systems. As the boot progresses and both virtual memory and process-switching facilities become available, the default heap manager is changed to a multi-threaded allocator that provides an appropriate balance of time and space usage properties for a multi-threaded kernel. Thus, the load manager dynamically changes its behavior.

9. Conclusion

We proposed a mechanism to control and observe interacting objects. Debugging functionalities are dynamically applied. Method invocations on relevant objects are intercepted by a meta-object and all interactions are accomplished in a transparent way. This dynamic approach minimizes the resource utilization for debugging.

The proposed mechanism is integrated in the XENOOPS environment, an execution environment for adaptive parallel programs. The debugger isn't restricted to a specific debug policy and/or graphical environment.

Acknowledgements

We would like to thank Herman Moons for his thorough reading and many remarks during the process of developing the debugging concept. The presented research are results of the Belgian Incentive Program "Information Technology" - Computer Science of the future, initiated by the Belgian State - Prime Minister's Service - Science Policy Office. The scientific responsibility is assumed by its authors.

- [1] Y. Berbers, W. Joosen, H. Moons, and P. Verbaeten, "The XENOOOPS Project" pp. 144-146 in *Proceedings of the 1991 International Workshop on Object-Oriented Programming in Operating Systems*, Palo Alto, CA, U.S.A. (1991-10).
- [2] P. Maes, "Computational Reflection" *Technical Report 67-2*, Vrije Universiteit Brussel (1987).
- [3] P. Wegner, "Concepts and Paradigms of Object-oriented Programming" *OOPS Messenger ACM*, Vol.1 (1) (August 1990).
- [4] R. D. Williams, "Supersonic fluid flow in parallel with an unstructured mesh" *Concurrency: Practice and Experience*, Vol.1, pp. 51-61, John Wiley (September 1989).
- [5] W. Joosen, Y. Berbers, M. Snyers, and P. Verbaeten, "Transparent Object Migration in Adaptive Parallel Applications" *Proceedings of EWPC'92, the European Workshop on Parallel Computing*, W. Joosen and E. Milgrom, Eds., pp. 300-311 (March 1992, Barcelona, Spain).
- [6] W. H. Cheung, J. P. Black, and E. Manning, "A Framework for Distributed Debugging" *IEEE Software*, pp. 106-115 (January 1990).
- [7] B. P. Miller and J. Choi, "A Mechanism for Efficient Debugging of Parallel Programs" *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 135-144 (June 22-24, 1988).
- [8] M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company (1990).
- [9] R. Schwartz and Friedeman Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail" *Interne Bericht Nr. 215/91*, Universitat Kaiserslautern (November 1991).
- [10] R. H. Campbell and A. M. Habermann, "The Specification of Process Synchronisation by Path Expressions" *Lecture Notes in Computer Science*, Vol.16, pp. 89-102, Springer-Verlag, Berlin (1974).
- [11] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roison, and X. Rousset de Pina, "Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications" *JOOP*, pp. 11-21, Laboratoire de Genie Informatique (October 1990).
- [12] B. Foote and R. Johnson, "Reflective Facilities in Smalltalk-80" *OOPSLA '89 Proceedings*, pp. 327-33 (1989).
- [13] C. Chambers, D. Ungar, and E. Lee, "An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes" *Proceedings of OOPSLA '89*, ACM (1989).
- [14] G. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, The MIT Press series in artificial intelligence.
- [15] Y. Yokote, F. Teraoka, A. Mitsuzawa, N. Fujinami, and M. Tokoro, "The Muse Object Architecture: A New Operating System Structuring Concept" *ACM Operating System Review*, Vol.25 (2), pp. 22-46 (April 1991).
- [16] P. Madany, N. Islam, P. Kougiouris, and R. H. Campbell, "Reification and Reflection in C++: An Operating Systems Perspective" *Internal Report, Department of Computer Science*, University of Illinois at Urbana-Champaign.

A Parallel Software Monitor for Debugging and Performance Tools on Distributed Memory Multicomputers

Don Breazeal Ray Anderson

Wayne D. Smith Will Auld Karla Callaghan

Intel Corporation Supercomputer Systems Division
15201 N.W. Greenbrier Parkway
Beaverton, OR 97006

Abstract

Program monitors must fulfill a number of requirements to be effective. Performance, reliability, generality, portability, and retargetability are all desirable and necessary attributes. On massively parallel distributed memory machines, these requirements present special challenges. As new, increasingly parallel and exotic architectures are created, program monitors must be able to move to these massive systems without massive re-implementation.

This paper describes the Tools Application Monitor (TAM), a parallel server that acts as a software monitor for parallel debugging and performance tools. The TAM provides a public interface that allows tools such as debuggers to control and monitor the behavior of numerous application processes with simple function calls. The TAM provides the basis for the next generation of programming tools targeted toward Intel Paragon™ systems.

1. Introduction

Software for parallel computers has been an area of active research since the introduction of the first parallel systems in the mid-1980's. Parallel systems present special challenges for developers of software tools such as debuggers and performance monitors. Some of those challenges include the following:

1. Tool Complexity

Parallel performance and debugging tools are more difficult to develop than their sequential counterparts because the state of a parallel application is more difficult to describe and control. The added complexity increases the development time and maintenance requirements for parallel tools.

2. Massive Amounts of Data

Performance monitoring, in particular, can require the collection and reduction of large amounts of data from large numbers of concurrent processes.

Debugger display of program data, when scaled to hundreds or thousands of processes, can result in an overwhelming amount of data as well.

3. Performance

Tool performance issues require special attention on distributed memory machines because of the potentially large number of processes to be monitored.

4. Machine Dependence and Programming Model Dependence

The relative lack of hardware and software standards for parallel systems makes it difficult to retarget parallel tools for new systems.

Also, the wide variety of competing programming paradigms for parallel computation such as control vs. data parallelism and explicit vs. implicit message-passing makes writing general purpose parallel tools difficult.

To address these problems, Intel is developing a parallel

Supported in part by
Defense Advanced Research Projects Agency
Information Science and Technology Office
Research in Concurrent Computing Systems
ARPA Order No. 6402, 6402-1, Program Code No. 8E20 & 9E20
Issued by DARPA/CMO under contract #MDA972-89-C-0014

Tools Application Monitor (TAM), initially targeted for use on Intel Paragon™ systems. The TAM is a parallel software monitor that will serve as the lowest level in a layered tools architecture. Parallel debugging and performance tools will reside at a higher level and will rely on the TAM for machine dependent process monitoring and control. A parallel interface library will connect the layers and allow parallel tools to perform operations such as "single step" or "read memory" on numerous application processes with a single function call. This approach allows the tool developer to concentrate on what functions to provide and how to present them to the user rather than the machine dependent aspects of parallel process control.

The paper is organized as follows: in section 2 we describe the target system, an Intel Paragon running Paragon OSF/1. Section 3 describes the motivation for the TAM. Section 4 lists the TAM design goals. Section 5 provides an overview of the TAM. Section 6 describes the instrumentation techniques used in the TAM. Section 7 provides an example of a debugger implemented using the TAM. Section 8 gives some background on related work, and section 9 summarizes our conclusions.

2. Paragon OSF/1

The Paragon OSF/1 operating system is a true distributed Unix operating system. Each node of the parallel machine runs a Mach 3 microkernel and an extended OSF/1 server. The operating system provides a single-system image, meaning that there is a single name space for process id's, file systems, and all other system resources. Parallel applications are gang-scheduled in system *partitions*, which are logical divisions of node resources. A special partition, the *service partition*, is used to execute login shells and non-parallel Unix processes. This partition is dynamically load-balanced and processes may migrate freely among the nodes in the service partition.

This single-system image, or global pid space, allows processes on distinct nodes to signal each other, wait for each other, create pipes between each other, and to trace one another via the Unix `ptrace()` system call[12]. These features are useful for single-process Unix debuggers in that when a process under debug migrates away from the debugger (or vice versa), the user is unaware of the migration and the debugger continues to work as normal. We have demonstrated this capability using the GNU debugger, `gdb`[15].

Parallel applications running under Paragon OSF/1 communicate via typed message passing, compatible with the message passing on previous generation Intel machines running the NX operating system[11].

On the Paragon system, parallel applications are invoked in the same manner as any other Unix program. However, parallel applications are linked with a special library containing routines that turn the initial Unix process into a parallel application. The initial Unix process, (also known as the *controlling process*), may become a parallel application either implicitly or through explicit function calls, depending on how the programmer builds the application. In either case, the controlling process executes a parallel fork operation, possibly followed by an `exec`, to create the parallel portion of the application. This "load model" presents a significant challenge in bringing the application under monitor control. Operating system extensions were required to implement this.

3. Motivation for the TAM

Given the single-system image presented by Paragon OSF/1, one could implement a front-end tool that directly calls `ptrace()` to monitor multiple processes running on multiple nodes. However, this approach has two basic flaws:

1. For hundreds or thousands of processes, the front-end tool would have to call `ptrace()` once for each operation on each process. This would create an unacceptable bottleneck.
2. The operations offered by `ptrace()` are too primitive for efficient remote use on a parallel system. A massively parallel monitor requires high-level operations that can be done locally, on the same node as the monitored process, to minimize intrusion on the message passing network caused by numerous low-level monitor requests.

To effect a solution that overcomes these flaws, the monitor must be implemented to perform high-level operations in parallel.

For example, single stepping by source line, implemented using the breakpoint mechanism, requires repeated memory reads and writes, executions, and register reads in order to set a breakpoint, run to it, and get the program counter, respectively. Doing this remotely requires multiple messages between processors; doing it remotely for hundreds or thousands of processes creates a serious bottleneck and serializes the implementation. Thus, the TAM implementation moves much of this work out to the nodes, as in the `lpt` debugger[5].

4. Design Goals/Requirements

This section describes the requirements that constrained the design of the TAM.

- **Reliability**

This is an obvious requirement, but important and difficult to achieve in a parallel remote monitor, where an operation might succeed for some processes but fail in differing ways for others. This overrides all other requirements.
- **Performance**

This is another obvious but difficult requirement. We arbitrarily set a limit of 1 second response time for interactive commands such as read memory, get stack tracebacks, and so forth.
- **Portability / Retargetability**

Parallel architectures are evolving rapidly. In a sense, this stifles the evolution of parallel programming tools, since tool builders are constantly scrambling to put the basic tools back onto the latest architecture. The TAM is intended to live past the current generation of Intel systems.
- **General Interface**

A goal for the TAM interface is to provide tool builders outside Intel the capability to port existing tools to the interface as well as create new tools using the interface. Research into parallel programming tools is a growing field, and this interface must enable researchers to concentrate on the important problems without having to reinvent the monitor for each tool.
- **Integration of Monitoring Facilities**

By combining monitoring facilities into a single monitor, any programming tool may make use of the facilities traditionally tied to any other tool. For example, a debugger can run to a breakpoint, then turn on profiling. All tools get a consistent picture of the program.
- **Monitor Any Program**

The TAM must be able to monitor any application that can run on the machine, without special compilation or linking to special libraries. A monitor failing this requirement is useless for production codes.

5. Design of the TAM

The TAM is comprised of two major components: an interface library, and a server. Multiple instances of the TAM server exist, one on each node in the partition where the application runs. The individual TAM servers are started when a call to the interface library loads an application, and each server is responsible for monitoring all processes in the application on its local node. Services provided by the TAM include traditional Unix monitoring activities such as reading and writing the application data space, management of breakpoints, watchpoints and tracepoints, program single stepping, and profiling, as well as features specific to a parallel message passing system such as inspection of message queues, data reduction and event logging.

One of the TAM servers has special responsibilities in that it monitors the controlling process of a parallel application; this TAM is called the *parent TAM*. When the controlling process begins to create the parallel part of the application, the parent TAM is responsible for starting a TAM server on each node of the partition. It also establishes the communication ports between itself and the rest of the TAMs. These ports are used to implement remote procedure calls (RPCs) in the parallel server.

Figure 1 illustrates the design of the TAM and its interaction with the front-end tool and the application being monitored. In that figure, bold arrows represent the RPC/Mach port network connecting the TAM servers, thin arrows represent the message passing network connecting application processes, and thin lines without arrowheads represent the trace() communication between each TAM server and the application processes under its control.

5.1 Interface Library

The TAM interface library, *libtam.a*, provides an external C Language interface to the services of the TAM. The interface hides the implementation of the RPC interface from the caller (the RPC mechanism is described in detail in section 5.4). In addition, the interface provides facilities for managing program data returned as a result of a data request as well as error reporting mechanisms.

The data buffering mechanism is based on the one used in the IPD parallel debugger on the iPSC[®]/2 and iPSC[®]/860[5]. It has been re-implemented and extended to satisfy the general requirements of the TAM. The mechanism has been improved through dynamic buffer allocation, removing the arbitrary 1K-per-node limit on data requests. A tool can then manipulate these buffers, via the interface routines, to collect identical data sent from different nodes so that it need only be displayed once. Redundant data occurs often in practice, for example, when many processes

are stopped at the same breakpoint and their stack traceback is identical. The buffer mechanism provides only primitive facilities for managing data returned from the TAM. As a result, much of the responsibility for this management is left to the front-end tool, such as releasing the buffers to be de-allocated when they are no longer needed. The TAM library requires the front-end tool to live up to these responsibilities through strict error checking.

Any TAM request for data involving multiple nodes may result in data and/or error information being returned as some nodes fail and others succeed. Error information may vary from node to node just as program data can. To return each of the error numbers with the list of nodes associated with each error is exactly the same problem as returning program data with the list of nodes which produced it. When an error occurs, a TAM library routine can be called to return an error message string and a list of nodes on which the error occurred.

5.2 TAM Servers

Each TAM server consists of four major components:

1. A node in the broadcast spanning tree.
2. A process list.
3. An external interface (RPC).
4. A selectable tracing and logging facility.

Each TAM server must respond to two external stimuli: TAM RPCs and debugger events generated by monitored processes. Since the primary debugging instrumentation is done using the ptrace() system call, debugger events result in a signal, SIGCHLD, being sent to the monitor. The TAM servers block awaiting an RPC request. If a debugger event occurs, control transfers to a signal handler, which calls a "handle event" RPC on the local RPC port and returns to receive the request and handle the event.

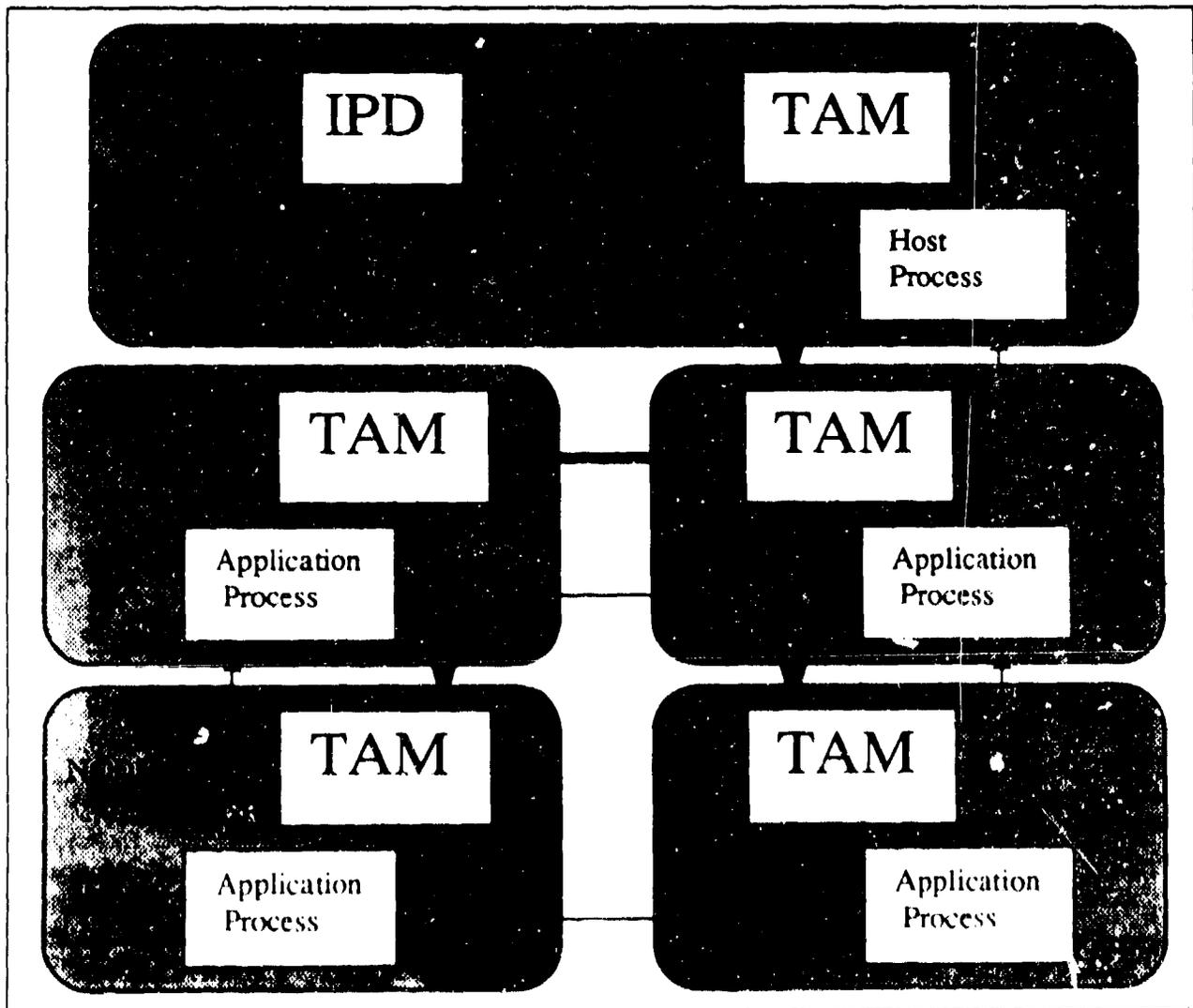


Figure 1. TAM Usage Model

5.3 Loading a Parallel Application for Debug

One of the more system-dependent parts of the TAM is the "load program" interface. As described in section 2, loading a parallel application on Paragon is a relatively straightforward operation. Loading it for debug, however, is more complicated.

A parallel application is invoked by running a single Unix process. This process calls a routine that allocates a partition of nodes. The parent TAM uses internal breakpoints to stop the process on return from that routine so that it may start a TAM server running on each node of the partition. Each of these servers notifies the OS of the ID of the application it wishes to monitor. The parent TAM may then resume the initial *controlling process* so that the application processes are loaded on the nodes as well. As these start up, the operating system suspends them and notifies each local TAM that a new traced process has been created, and the TAM sends an event message back to the interface library.

5.4 RPC Implementation

The TAM library provides an interface for remote procedure calls to the TAM. The RPC implementation is generated using the Mach Interface Generator (MIG)[9][10]. MIG reads the RPC specification and outputs C code to implement the RPCs using Mach ports.

RPCs are synchronous with respect to the interface library; an interface routine will not return to the front-end tool until the RPC is complete. However, each TAM server executes the RPC asynchronously. The TAM server receives the request, forwards it to the TAM servers downstream from it in the spanning tree, executes the request if it is part of the request, then receives any errors returned from downstream and forwards those, plus its own, upstream. If the request is for program data, a subsequent call to receive the data is required. Data is received and stored in buffers in the TAM interface library.

There were three communication mechanisms available to implement the TAM communication network: Mach ports, sockets, and NX message passing. Sockets were dismissed almost immediately because the software overhead would degrade performance below the specified limits. NX message passing is very attractive because of its high speed and the absence of a need for initialization, but because the monitor has to run in the same application space as the monitored processes (so that they are gang-scheduled together), a name-space conflict between the application and the monitor would be created. Thus, Mach ports were the only choice with the necessary performance and security features.

Mach ports have nearly the speed of NX message passing once the connection has been made, with the added attrac-

tion of the security that only processes that have been granted send rights to that port may send to it. However, the initialization required for the TAM communication network to use this paradigm is significant.

In order to broadcast TAM requests to potentially thousands of TAM servers, the communication network is created in the form of a spanning tree, rooted at the node where the front-end tool resides. This allows the interface library to send a request to at most $\log(n)$ nodes, where n is the number of nodes allocated for the application, and have them forward it in parallel using the contention free algorithm described in [1]. Although some commands may be directed to only a subset of the nodes, all commands follow this complete path out and back, since the overhead of calculating different paths based on the node list outweighs the extra message passing.

The pathways used to send event information and program data to the front-end currently circumvent the spanning tree network. These initial connections are candidates for future optimization. The impact of such connections is not yet clear.

6. Instrumentation and Examination Techniques

The TAM provides instrumentation traditionally used by debuggers and performance monitors. In this section we describe the monitoring techniques used by the TAM for these types of tools, but we categorize them in this way for reference only. In the TAM environment these facilities are interchangeable.

6.1 Debugging

The TAM uses `ptrace()` to perform debugging operations. The significance of this is that the TAM is able to make the connection through the operating system to trace parallel application processes started on a remote node. Briefly, modifications to the OSF/1 server were required to:

- allow the TAM to be notified of any new (traced) processes in a specified process group and to trace them
- allow the TAM to wait for sibling processes

OSF/1 provides two potential mechanisms for monitoring processes: `ptrace()`, using the Unix facilities of the OSF/1 server, and Mach exception ports, using the facilities of the Mach 3.0 microkernel[10]. `Ptrace()` seemed to be the choice for a portable, retargetable monitor. The model is simple and changes to support debugging parallel applications could be isolated in the OSF/1 server, leaving the microkernel alone, which helps with maintainability of the OS.

Although the method of instrumentation for debugging is conventional, this instrumentation is employed to implement high-level operations on behalf of the front-end tool. For example, to do a stack traceback using a "read memory" monitor command, a front-end tool would have to call the monitor repeatedly. The TAM provides a "stack traceback" command, so that a single call can return what normally requires many calls. This technique is used for calculating addresses, source stepping, examining message queues, and other functions in much the same way that IPD on the iPSC/860 did, but with a public interface.

6.2 Performance Instrumentation

For performance monitoring, initial implementations of the TAM will provide only profiling capability. However, an event trace mechanism will be added in the near future.

The Application Performance Monitoring Subsystem consists of three parts:

1. The TAM
2. A performance monitoring library linked with every application process

3. An event trace server that moves event trace data off the application nodes

The performance monitoring library is automatically linked with every application; no special switches are required and the library adds only 10-20K bytes to the size of a user program until performance monitoring is activated. Once activated, these libraries create event trace buffers and other structures dynamically.

To do event tracing of a parallel application, the TAM servers will instrument the application process in memory by replacing application code at the trace address with a branch to a new page. This new page will be allocated by the TAM server and attached to the application process. The new page will contain code which calls the performance library event trace code. The library will write the event trace to the trace buffer. On return, the code in the new page will execute the replaced instructions and branch back to the original application code.

At pre-determined points during performance monitoring, the performance library will send the event trace buffer to an event trace server. This server will run in the service partition. It will collect event traces from the nodes and either write the traces to a file, forward them to an interactive tool

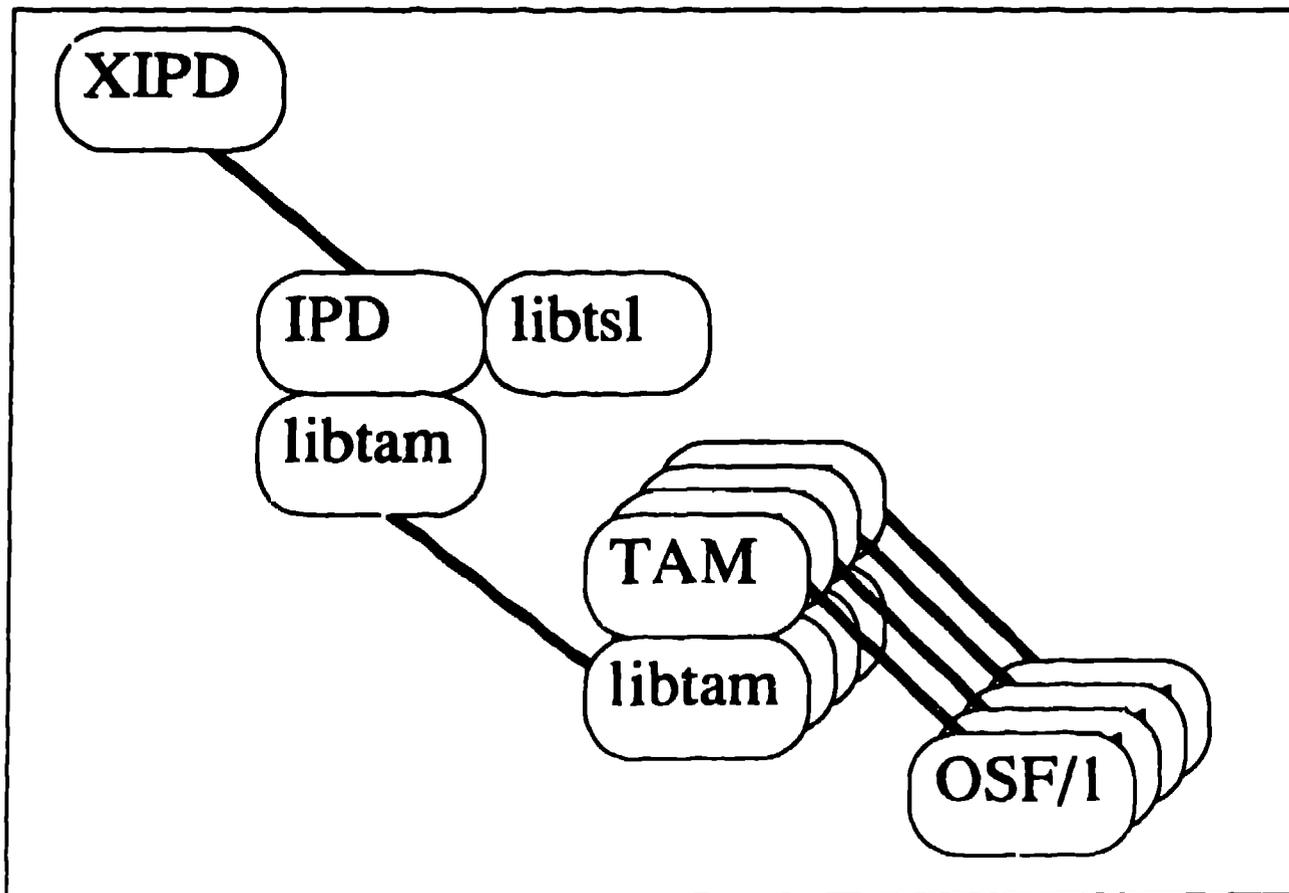


Figure 2. IPD Components

for real-time animation, or both. The trace server will be used in this way to minimize the intrusion of writing out buffer contents from the application.

Profiling an application is somewhat simpler. To profile a parallel application, the TAM servers call the performance library to dynamically activate the OSF/1 `profil()` mechanism. Both event tracing and OSF/1 profiling can be active simultaneously.

7. An Example

The Interactive Parallel Debugger (IPD) is a symbolic, source-level debugger for parallel programs written in C, FORTRAN and Assembler Language[5]. IPD offers the ability to load, start and stop parallel processes, single-step their execution, set breakpoints and watchpoints, display and change data, and examine message queues. The first implementation of IPD was for the Intel iPSC/2 and iPSC/860 computers. That implementation was developed prior to the TAM and had to rely on code within the NX operating system to handle low-level chores such as setting breakpoint traps, single-stepping an application, or modifying an application's data space.

IPD has recently been ported to the Intel Paragon system and interfaced with the TAM. The interface consists of a few high-level function calls to TAM library routines and replaces over 6,000 lines of operating system code that previously supported IPD. A separate library of routines for symbol table processing (*libst.a*) has been developed and is now being used by IPD. A graphical user interface is currently being developed. A component diagram appears in Figure 2.

The operation of IPD and its interface with the TAM can be characterized as a loop consisting of three basic actions:

1. Parse and validate a user request.
2. Forward the validated request to the TAM.
3. Receive process events back from the TAM.

Step 1 is standard to any interactive tool and is not affected by the use of the TAM. In Step 2, TAM services are requested through calls to TAM interface library functions. The library functions translate the requests into remote procedure calls and forward them on to the TAM. TAM library functions used by IPD include the following:

- `tamLoad()`

Loads an application program. The program may be either a sequential or parallel program.

- `tamExecute()` and `tamStop()`
Starts or stops a set of processes.
- `tamAddBreakpoint()` and `tamAddWatchpoint()`
Adds code breakpoints or data watchpoints to a set of processes.
- `tamReadMemory()` and `tamWriteMemory()`
Reads or writes to the data space of a set of processes.
- `tamReadMsgQ()` and `tamReadRecvQ()`
Reads the pending messages being sent or received by a set of processes.
- `tamReadRegisters()` and `tamWriteRegisters()`
Reads or writes the register file for a set of processes.
- `tamInstructionStep()` and `tamSourceStep()`
Steps a set of processes either one machine instruction or one source line.
- `tamReadTraceback()`
Reads the stack frame for a set of application processes.

Step 3 is the mechanism through which the TAM informs IPD when the state of a process being monitored changes. State changes are communicated as events, returned when IPD calls the TAM library routine, `tamRecvEvents()`. Events are generated whenever a process is created, stepped, stopped or killed, or when it is interrupted by a signal, breakpoint or watchpoint. Typically, IPD will receive any pending events and update its internal process tables between each user request.

In addition to the services described above, IPD relies on the data buffering and error handling functions included in the TAM interface library. These functions allocate space for the data and error codes returned from the parallel application and consolidate the buffers when identical data or errors are returned from different processes. IPD then uses TAM library routines to access the consolidated data or errors.

Paragon IPD is more portable and extensible than its iPSC predecessor because it no longer relies on custom operating system support. All monitoring of application processes is accomplished through the calls to the TAM interface library routines which hide the complexities of

machine dependencies, inter-processor communication and data/error reduction. Additionally, Paragon IPD incorporates new application profiling features which allow closer integration of debugging and performance monitoring functions. The integration of these functions is made possible by the use of the TAM as a common monitor.

8. Related Work

Software trends in general are moving toward portable, re-targetable, and general implementations using standard interfaces. Despite this, relatively few portable tools exist for parallel systems. Notable exceptions include Express[13], PICL/ParaGraph[7] and Pablo[14]. We believe that this is due in large part to the absence of a usable public interface to general monitoring facilities.

Integrated monitoring is not a new idea. The TOPSYS project[2][3], in particular, has implemented a sophisticated monitoring facility that is used by multiple tools. TAM technology draws heavily upon the results of this work, and will likely continue to exploit the results of this research.

Hoven[8] has implemented a Mach 2.5 interface for process monitoring that is intended to support a variety of servers running on a microkernel. Our OS extensions solve most of the same problems using Unix signals and `ptrace()`, with the exception of thread-level monitoring. Breakpoint debugging of threads is of questionable value due to its intrusiveness, but clearly `ptrace()` is not the answer for threads.

The instrumentation mechanism for event tracing is similar in concept to the work of Cargille and Miller[6] in that it modifies any existing program for event tracing, although the modification takes place in memory rather than the object file and the method of instrumentation is somewhat different.

9. Conclusions

The TAM approach provides several advantages when developing parallel software tools. Some of those advantages are as follows:

1. Reduced Tool Complexity

Parallel tools can be developed more quickly when the complexities of machine dependencies and the back-end application monitoring tasks are hidden. The TAM hides this complexity by providing a high-level interface for such tools. Also, the TAM code is

shared by multiple front-end tools, thereby reducing the total amount of code to be maintained.

2. Data Reduction

The services provided by the TAM include data buffering and reduction. These services assist the front-end tools in collecting and presenting data from multiple processes.

3. Tool Performance

The TAM network is highly tuned to minimize the inter-processor communication required for application monitoring. The state information maintained by the TAM reduces the number of communications required between the front-end tool and the TAM and the fast spanning tree broadcast algorithm optimizes communications between the TAM processes.

4. Tool Portability and Retargetability

The high-level TAM interface eliminates most machine dependencies from the front-end tools, allowing them to be retargeted to new systems more easily. Also, the TAM is a Unix-based product, written in an object-oriented fashion using C++. The use of Unix improves the portability of the TAM itself, thereby making it easier to retarget any of the front-end tools built on top of the TAM. The TAM's modular, object oriented design ensures that additional features can be added to the TAM as needed to support future tool requirements.

5. Tool Integration

Concentration of monitoring facilities that are traditionally tied to debuggers or performance monitors in a single monitor increases the power and flexibility of both types of tools. For example, a performance monitor can profile a section of a program by running to a breakpoint, turning profiling on, running to another breakpoint, and turning profiling off and flushing the results.

It is our intention that the public TAM interface be used by multiple tools from different sources. Experience has shown that academic and research institutions are continually building new tools and enhancing old ones; TAM technology is intended to enable and ease these efforts. Tool builders should be free to concentrate on managing programming paradigms, parallelism, and user interfaces, and should not be burdened with re-inventing monitoring facilities for each new tool.

10. Trademarks

1. OSF and OSF/1 are registered trademarks of the Open Software Foundation, Inc.
2. Unix is a registered trademark of Unix System Laboratories, Inc.

References

- [1] Michael Barnett, David G. Payne, and Robert van de Geijn, "Optimal Broadcasting in Mesh-Connected Architectures," *Parallel Processing Letters*, Submitted.
- [2] Thomas Bemmerl, Robert Lindhof, and Thomas Tremel, "The Distributed Monitor System of TOPSYS," In: *CONPAR 90 - VAPP IV, Lecture Notes in Computer Science, vol. 457*, Springer-Verlag, Berlin, Heidelberg, New York, September 1990.
- [3] Thomas Bemmerl, "The TOPSYS Architecture," In: *CONPAR 90 - VAPP IV, Lecture Notes in Computer Science, vol. 457*, Springer-Verlag, Berlin, Heidelberg, New York, September 1990.
- [4] D. L. Black, D. B. Golub, K. Hauth, A. Tevanian, Jr. and R. Sanzi, "The MACH Exception Handling Facility", *Proceedings, ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices Vol. 24, No. 1*, May 1988.
- [5] Don Breazeal, Karla Callaghan, and Wayne D. Smith, "IPD: A Debugger for Parallel Heterogeneous Systems," *Proceedings of the ACMIONR Workshop on Parallel and Distributed Debugging*, May, 1991.
- [6] Jon Cargille, Barton Miller, "Binary Wrapping: A technique for Instrumenting Object Code," *ACM SIGPLAN Notices, Volume 27, No. 6*, June 1992.
- [7] G.A. Geist, M.T. Heath, B.W. Peyton, and P.H. Worley, "A User's Guide to PICL, A Portable Instrumented Communication Library," Oak Ridge National Laboratory, June, 1991 ORNL/TM-11616
- [8] Rand Hoven, "Mach Interfaces to Support Guest OS Debugging", *Proceedings, USENIX Mach Symposium*, USENIX Association, November 20, 1990.
- [9] Keith Loepere, Editor, "Mach 3 Server Writer's Guide," *Open Software Foundation and Carnegie Mellon University*, March, 1992.
- [10] Keith Loepere, Editor, "Mach 3 Kernel Interfaces," *Open Software Foundation and Carnegie Mellon University*, March, 1992.
- [11] Paul Pierce, "The NX/2 Operating System," *Concurrent Supercomputing, A Technical Summary of the iPSC/2 Concurrent Supercomputer*, Intel Corporation, 1988
- [12] Open Software Foundation, *OSF/1 Programmer's Reference*, Prentice Hall, 1991.
- [13] ParaSoft Corporation, "Express User's Guide," ParaSoft Corporation, 1990.
- [14] Daniel A. Reed, Robert D. Olson, et. al., "Scalable Performance Environments for Parallel Systems," *University of Illinois*, 1991
- [15] Richard Stallman, Roland Pesch, *Using GDB: A Guide to the GNU Source-Level Debugger, Ed. 4.01, GDB version 4.4*, Free Software Foundation, January 1992

The Paragon Tools Application Monitor (TAM)

Don Breazeal

Intel Supercomputer Systems Division

donb@ssd.intel.com

Intel Supercomputer Systems Division

Outline

- **Paragon System Overview**
- **Tools Application Monitor (TAM)**
- **Paragon Tools Overview**

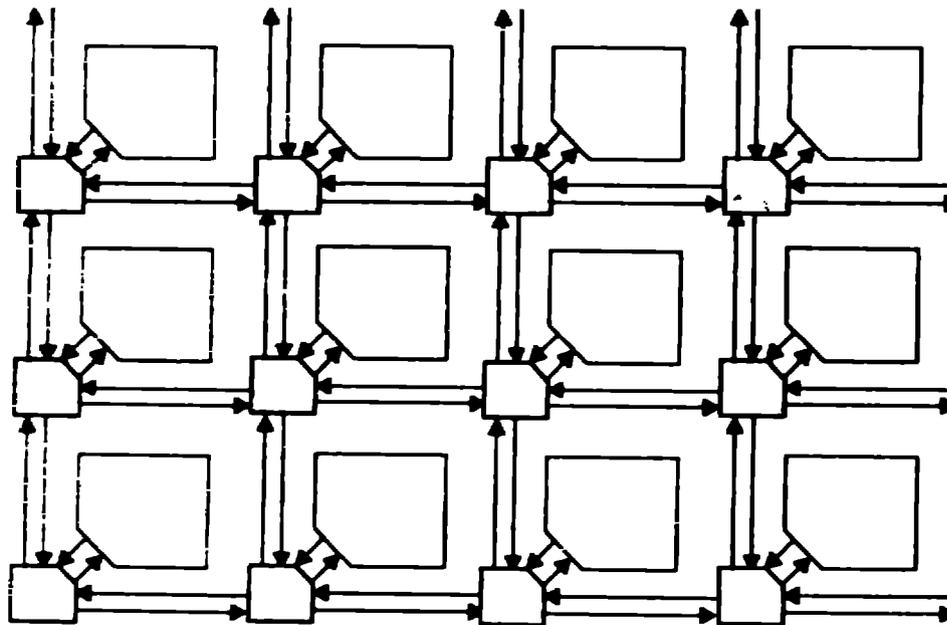
Paragon System Architecture

- **Up to 1000 nodes connected by high speed message routing H/W**
- **Each node has 2 i860/XP CPUs (75 MFlops double precision) and 16-64MB RAM**
- **NO Remote Memory Access (NORMA)**
- **Some nodes attached to IO devices**
- **RPM - global clock and H/W performance counters**

Intel Supercomputer Systems Division

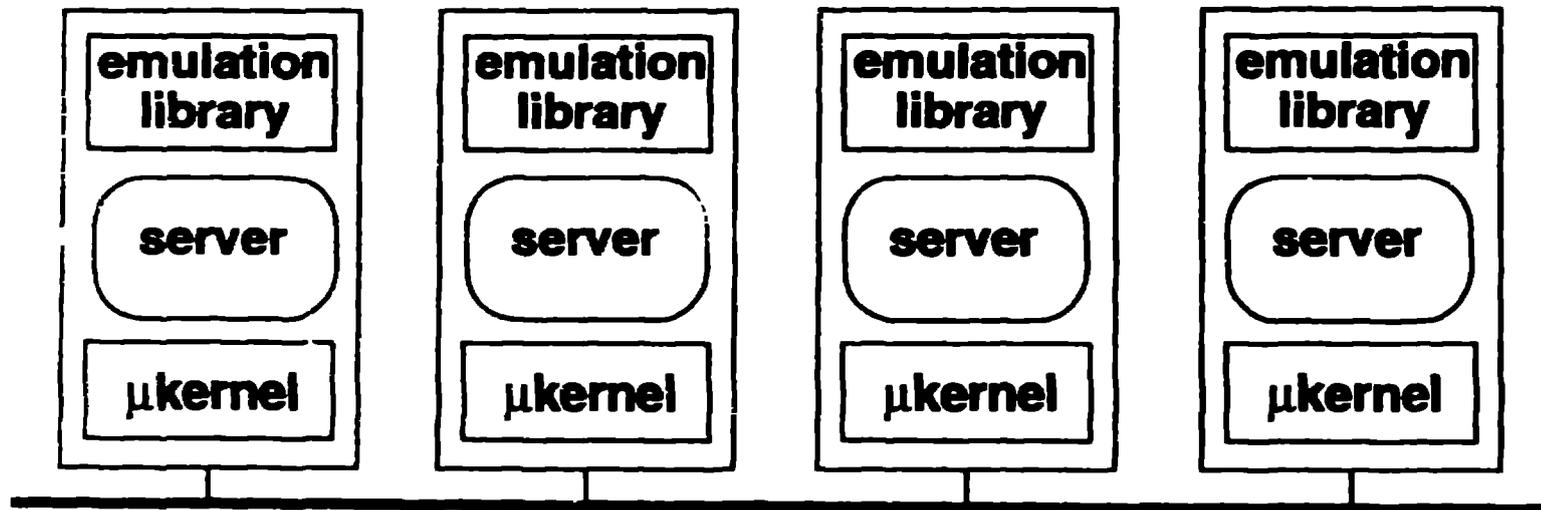
Interconnection Architecture

- **Nodes connected in 2-D mesh**
- **“Worm Hole” routing (vs. “store and forward”)**
- **200 MB/s full duplex, 40 nsec latency per hop**



Paragon OS Architecture

- **Mach 3 microkernel, emulation library, and single OSF/1 server on every node.**
- **Not all services of OSF/1 server used on each node; e.g., file services.**
- **Unused portions of server are paged out.**



Intel Supercomputer Systems Division

Distributed Unix

- **Process management fully distributed across OSF/1 servers**
 - process IDs
 - signals
 - wait()
 - ptrace()!
- **Parallel File System - files striped across I/O nodes**

Multicomputer Extensions

- **Multicomputer programming model**
 - **Typed message passing a la NX/2**
 - **Facilities for loading and controlling parallel applications**
- **Node allocation and partitioning**
- **“Service partition” load balancing and process migration provides scalable “front end”**
- **Gang scheduling in the “compute partition”**

Debugging a Parallel Program: Capturing Inter-Processor Communication in an iWarp Torus

Thomas Gross and Susan Hinrichs
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

To understand or improve the execution behavior of a program on a parallel system, it is often necessary to consider the interaction between the processors in the system. Since communication is important for all parallel programs, obtaining information about the inter-processor communication of the program is an important aspect of understanding program execution. If the user's concern is the performance of the parallel program, then the user must be able to capture the dynamic aspects of inter-processor communication. Unfortunately, timing information about inter-processor communication is often not easy to obtain. Building a special-purpose hardware performance monitor is too costly in most scenarios, and the use of VLSI to integrate communication and computation on a single component often provides only a few externally visible measurement points. A software monitor on the other hand is often too slow to allow execution of the monitored program without serious perturbation.

Although the integration of communication and computation engines on a single component provides some challenges to monitoring, it also opens the opportunity to program the processors in a parallel system so that they can monitor communication traffic. That is, during monitoring some processors execute the user program (to be monitored) while other processors execute a special monitor program (which captures information on the inter-processor communication). With adequate software tools, this information then can be analyzed to present a picture of the communication between the user processes. This paper discusses the benefits and difficulties of an implementation based on this idea for the iWarp system. We conclude that a programmable processor that integrates communication and computation is also suited to serve as a hardware monitor at a fraction of the cost of a special-purpose design.

1 Introduction

Debugging a program on a parallel system is difficult when the objective is to get a correct program. Getting a program to run fast (while maintaining its correctness) is even harder and requires a wide range of information about the execution behavior of the program. A performance debugger must provide information on how computation cycles, memory bandwidth, communication bandwidth, or other system

Supported in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330. Work furnished in connection with this research is provided under prime contract MDA972 90 C 0035 issued by DARPA/CMO to Carnegie Mellon University.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

resources are spent by the program. With this knowledge the programmer knows where to concentrate her efforts in tuning the program to remove bottlenecks.

Obtaining, analyzing, and presenting performance information for a sequential system is sometimes difficult and still an active area of research[SK90]. For parallel systems, the situation is even worse; in addition to the information that must be obtained for each processor, we must understand the interaction between the processors in the parallel system.

We have been exploring issues in performance debugging while working on the iWarp system, a private-memory, MIMD, parallel computer[BCC⁺88]. A single-node gdb-like debugger exists to debug the code on individual nodes of the iWarp, but this provides only a pigeon-hole view of the execution of a program. The single-node debugger treats inter-processor communication by a node the same way input/output is treated by a uniprocessor debugger: after a message has been sent, it is invisible and cannot be tracked until it is received by another node (when the single-node debugger for that node is able to inspect the message). To understand or improve the execution behavior of a program on a parallel system, it is often necessary to monitor this inter-processor communication. If we can observe how messages travel through the parallel system, we can combine this information with the information about the execution on each node to provide the user with a global picture of system performance. Consider the example where processor A must receive messages from both processors B and C before it can begin some computation. Suppose the message from C will arrive long after the message from B has arrived. If the compiled program on processor A has committed to receive C's message first, processor A will block, waiting to receive the message from C, and processor B will block, waiting for its message to be received. A single-node debugger is of little help in identifying such situations, since it cannot capture the reason for the delay in processor B by solely inspecting this processor's state. Furthermore, a single-node debugger may perturb the execution of the inspected process. Some pertinent communication issues to measure are frequency of messages, the number of words transmitted, and specific patterns in the messages.

There exist two approaches to obtaining such communication information: using a hardware monitor or software instrumentation. There is a well-known cost versus accuracy tradeoff between these two options. Using additional hardware to monitor performance is more accurate, because the monitoring does not steal any resources from the monitored program. However, using additional hardware takes much more effort. The monitoring hardware must be designed and built, and since it is special purpose, it cannot be used when the programmer is not interested in monitoring. For this reason, the buyers of systems are usually unwilling to pay for additional monitoring hardware (and the component designers are unwilling to sacrifice significant area or design effort to provide it). Furthermore, as advances in VLSI allow a tighter coupling of communication and computation, the monitoring of inter-process communication becomes more and more difficult. On an iWarp system, a hardware monitor that attempts to observe the communication between two adjacent nodes must understand an inter-processor bus protocol that includes resource allocation, routing, and flow-control. Once two or more processor nodes are integrated onto the same component, obtaining access to any communication between two such nodes will be next to impossible.

Software instrumentation costs less, because only the software is changed. The program or the system software can be instrumented to gather information. This option is more flexible than an approach based on a hardware monitor, but the additional software steals cycles and hardware resources from the original program, so the execution of the monitored program is perturbed. As communication overhead in parallel systems has decreased over the years enabling finer-grained communication, this software monitoring intrusion becomes less acceptable. For example, an iWarp node can send and receive four 32-bit words every 100 ns (the time it takes to perform either one single-precision floating point addition or

multiplication or two integer operations), so “bracketing” each communication operation with monitoring code may slow down a program on the iWarp by up to a factor of 40.

However, a parallel system like iWarp offers a hybrid solution. Each iWarp processor contains a computation and communication engine, which are tightly coupled, providing the computation engine with a detailed view of the communication system. At the same time, the computation engine can be programmed like a general-purpose processor, providing the opportunity to use such a processor as a monitor processor. The absence of a global memory makes all communication explicit at the hardware level, so if we monitor the communication between processors, we can obtain a complete picture of the communication in the system.

For the iWarp array, we have created a hybrid performance monitoring system that has many of the benefits of hardware monitoring while incurring costs close to those associated with software instrumentation. Section 2 gives some background about the monitored parallel system. In Section 3 we describe this idea for performance monitoring on iWarp in detail. In Section 4 we evaluate the current implementation of this performance monitoring system. Section 5 describes tools to display the inter-processor information captured by the monitor and gives an example of its use.

2 Background

In this section, we describe some key aspects of the parallel system.

2.1 Communication

We distinguish between two communication styles: memory communication and systolic communication [BCC⁺90]. In memory communication, all communication between two processors is directed through the local memory of the sender and receiver processors. The sender assembles a message in memory and then passes it to the communication system, which transfers the message to the memory of the receiver processor. Message passing is a well-known paradigm that is based on memory communication.

For systolic communication, the processor is directly connected to the communication subsystem. On the sender processor, the words of a message are passed to the communication system as they are generated without any buffering in memory. Similarly, the receiving processor consumes the words directly from the communication system without first buffering them in memory.

With memory communication, the unit of transfer from the sender processor to the receiver is a block of data; each block contains a number of words. Memory communication is easier to monitor than systolic communication. If a sender program uses memory communication, the data of a message are first assembled in memory, then the communication system is invoked to transfer the data. The entry point into the communication system is well-defined, and the communication system can record information like the starting time of the message transfer, the message size, and (upon completion) the transfer bandwidth. Any overhead associated with such bookkeeping operation is paid for only once per message. However, recording information at the start and end of a message transfer is not sufficient if the programmer needs information about the transfer of the individual words of a message, e.g., if data are transmitted in bursts.

With systolic communication, the data of a message are produced (or consumed) on the fly. That is, data are sent as they are produced by a computation unit (e.g., the floating point adder). Since the data are generated by processor on the fly, the processor cannot be used to record any information like the time the

item was sent. If we wanted to use the processor to capture such information, then the processor would have to stop generating data. So if we want to obtain timing information about systolic communication or if we need information for memory communication at a finer grain than complete messages, we cannot use the sending or receiving processor to monitor communication without serious perturbation of the program execution.

2.2 The iWarp system

The design of the iWarp component has been described earlier [BCC⁺88, BCC⁺90]; here we summarize only those aspects that are essential for the understanding of this paper.

The iWarp processor supports multiple *logical channels* between adjacent nodes, so multiple high speed connections can be set up using the same physical busses. On each node there is a finite number of queues, which buffer the data sent over the logical channels. A logical channel is assigned a queue on the source node and another queue on the destination node. These logical channels can be chained together to form a *pathway* by using the destination queue of one logical channel as the source queue of another logical channel. The communication hardware takes care of forwarding data through the intermediate nodes of chained logical channels, so the pathways provide direct, high-speed connections between nodes that are not physically adjacent. Pathways can be used to reserve resources for memory communication (message passing), or to set up connections for systolic communication.

An iWarp system is built out of iWarp processors and is arranged in an $n \times m$ torus. Each iWarp system is connected to a workstation that server as its front end, e.g., the output of a `printf` statement executing on a node appears on the file system of the front end. To date (August 1992), a number of systems have been built ranging in size from 4 to 256 nodes.

2.3 User model

The user seldom programs the iWarp system at the level of logical channels. Instead, she either uses a parallel program generator or describes the computation as a set of processes with connections between these processes. A parallel program generator like Assign [O'H91] or Apply [HWW89, BG91] takes a high-level description of a computation (that is, without any explicit statements for communication or data placement) and translates that description into a program for each node of the iWarp system with connections between nodes as appropriate. If the programmer describes the computation as a set of processes, a tool maps this set of processes onto the target iWarp torus, provided the number of processes is less than or equal the number of nodes in the target iWarp torus. After the processes are mapped onto the iWarp torus (either by the parallel program generator or the tool mentioned above), another tool sets up logical channels to implement a connection between any two nodes if the programs mapped onto these nodes are connected. The number of connections that can pass through a node is limited by the number of logical channels, which is taken into account when mapping the processes onto nodes. Figure 1 shows the mapping of a set of processes and connections onto an iWarp torus.

3 Monitoring inter-process communication

To get a handle on inter-process communication in a parallel system, it is necessary to start with the inter-processor communication. For example, if we want to observe the flow of data from Node 2 to

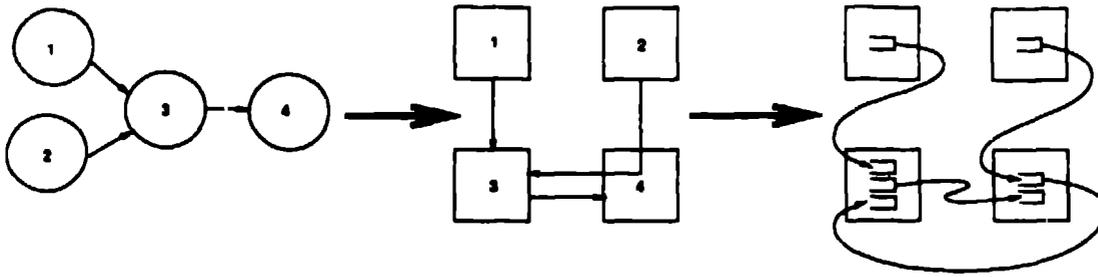


Figure 1: An example of mapping processes onto the iWarp array and mapping connections onto logical channels.

Node 3 in Figure 1, we must consider Node 4 as well, since the data travel through this node. Since the current runtime system allows only a single user process per node (and this is not likely to change in the future), information about the flow of data between processors can be directly mapped onto the inter-process communication of the user program.

If we know how the length of a logical channel queue changes over time, we have an idea of the run time communication patterns. For instance, while a queue is empty, we know either there is no communication going on over this logical channel or the receiving node is waiting for data from the sending node. While the queue is full, the sending node is sending faster than the receiving node is receiving. By watching how queues fill and empty during the course of a program's execution, the programmer can visualize the communication flow of her program and identify bottlenecks or dependences between messages. The actual number of words in a queue is less important at this level of analysis than the state (full, empty, partially full) of the queue. If the queue is full, the sender cannot proceed. If the queue is empty, the receiver cannot proceed; otherwise both can go on.

To get an accurate picture from monitoring inter-process communication we must try to maximize the accuracy of the gathered data and minimize the perturbation caused by monitoring. Section 1 explained how pure hardware or pure software monitoring solutions are not practical for the iWarp system. A pure hardware solution could not gather sufficiently detailed data about the communication, and instrumenting all communication functions could add too much perturbation for programs with fine-grained systolic communication requirements. However, we notice that not all nodes of an iWarp array are used all the time, and these unused nodes have the same capabilities as the nodes used to execute the user program, so we can use these nodes to gather information about the execution of the user program. That is, some nodes are program nodes (executing the user program), and others are monitor nodes, capturing the inter-processor communication between program nodes. Since the monitoring programs are running on a different set of nodes, they are not stealing cycles from the user program, one of the advantages of using additional hardware to monitor performance. Since the nodes are programmable, no extra hardware needs to be designed, and when monitoring is not desired, the monitoring nodes can run user programs.

This hybrid approach of using some of the nodes in the parallel system for user program execution and the rest for monitoring is integrated into the standard communication tool chain [Hin91]. Whenever the user selects the monitoring option, the tool chain spreads out the connection definitions and node mappings, so monitoring nodes can be inserted between each adjacent pair of user program nodes. The pathways between any two nodes are lengthened to pass through the monitoring nodes. See Figure 2 for two examples of this transformation. The monitoring nodes (shown dashed) are loaded with the monitoring program.

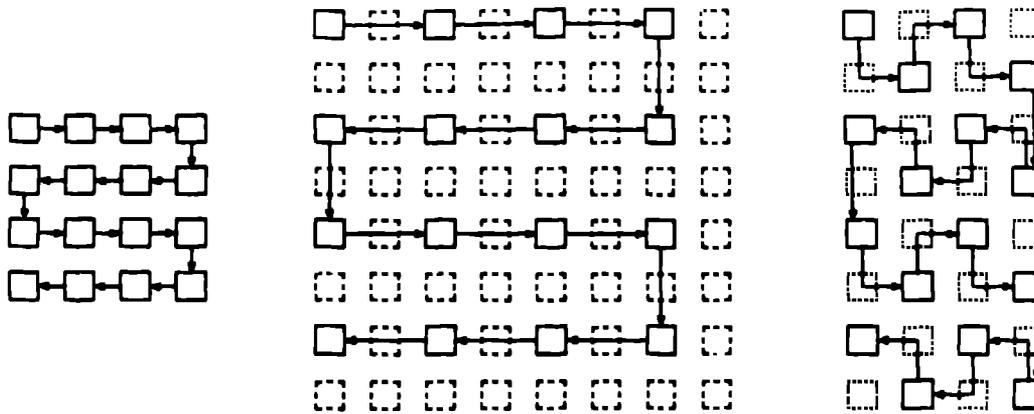


Figure 2: Network layout of original program is on the left. Two different network layouts of the monitored program are to the right. The current monitoring tool chain uses the center layout.

During the execution of the user program, the monitoring nodes gather data about the size of the logical channel queues of the pathways that pass through them. At the end of execution, this information is downloaded to a general-purpose workstation that is connected to the iWarp torus. There the programmer can use other tools to process, analyze and view the queue length data.

3.1 Gathering queue state data

To gather accurate queue length data while adding little perturbation, the monitoring program takes advantage of several features of the iWarp architecture. Each iWarp node consists of a computation agent and a communication agent that coexist on the same chip. These agents can execute asynchronously, or the computation agent can control the communication agent. The monitoring program runs the two agents asynchronously, so data passing on pathways through the monitoring node can continue without software interference. Since the computation and communication agents exist on the same chip, the computation agent can sample the state of the communication agent with little overhead. To determine the length of all network queues, the computation agent must read 4 control space registers, which takes 4 cycles.

The monitoring program also takes advantage of the hardware support for configuring multiple logical channels between nodes. The program's connections are mapped onto logical channels, and different connections use different logical channels with distinct physical queues, so the monitor program can easily differentiate between different program connections. If all the communication traffic was collected in a single queue, this division would be difficult to discern without additional protocol information and bookkeeping.

The monitoring program stores timestamps with the queue length data, so the programmer can see how network queue length changes over time. It is also interesting to compare how queue lengths change over time between different nodes. To make this comparison accurately, there must be some sense of global system time. Each iWarp processor contains two clock timers (with a resolution of 8 clocks, i.e. 400 ns on a 20 MHz system), and one of these timers is reserved for the user program. Our implementation sets the user timers of all nodes to a common global time in two steps using a synchronization package developed locally [FGOS92].

3.2 Processing queue state data

The monitoring program stores the gathered data in a buffer. At the end of execution or when the buffer fills up, the monitoring program sends this data to the host processor. If the data is transferred after the end of the monitored program's execution, the time and bandwidth spent sending the data is not critical. However, if the buffered data must be sent during execution of the user program, this transfer steals bandwidth from the monitored program, and the queue length cannot be monitored by this node while it is sending the data. This problem could be avoided by halting execution of all nodes in the iWarp torus. However, since the monitoring nodes pre-process the samples on the fly (see below), there has been no need so far to download the monitoring data before the end of the user program's execution.

Forms of data compression can be used to reduce the amount of sampled data stored on the monitoring node. We have observed that network queues are often in a steady state, so the queue length from the last sample is often the same as the queue length of the current sample. Data from these two samples can be merged into one entry. This sort of data compression should be performed some time before the data is analyzed by the programmer to merge large blocks of redundant data. Other standard data compression techniques can also be used to reduce the amount of space needed to store individual entries. These data compression techniques can be performed during the sampling loop to reduce the amount of space needed in the monitoring buffers and so increase the amount of time before the monitor data buffer is filled. However, adding this additional computation to the critical sampling loop may reduce the queue length sampling rate.

To determine the cost of the data compression, we created two versions of the time critical sampling loop in assembly code. The first version did no data compression and stored all samples in the monitoring buffer. The second version compared the current sample with the last stored sample, and only stored the current sample if it was different. On a set of example programs, the first version on average took 31 cycles per iteration. The second version on average took 30 cycles per iteration. It may seem surprising that the loop speeds are so similar, even though the first loop is simpler and performs fewer conditionals. However, often the queue length does not change between between samples, so the second loop performs fewer stores, and the comparisons are slightly faster than the stores. We decided to use the version of the loop that performed data compression in the loop, because the times of the two loops were very close and the second loop consumes far less buffer space.

4 Evaluation of the current implementation

We implemented a prototype version of the monitoring system described in the previous section. This section discusses how close we came to our goals of minimizing perturbation of the monitored program and maximizing accuracy of the gathered network data.

4.1 Monitored program perturbation

From timing several monitored programs it appears that the monitoring code does not significantly affect the execution time of the monitored program. To get a better idea of how monitoring affects the execution of the monitored program, we looked at individual factors that might skew the execution.

Spreading networks over monitoring nodes can cause two types of skew. First, each network must go over twice as many links, so the time to transmit a word doubles. The other possible source of skew is the

addition of network queues. Each monitoring node adds one queue to the network capable of buffering 8 additional words. If a sender and receiver are just barely blocking, the additional buffering may keep the sending node from blocking.

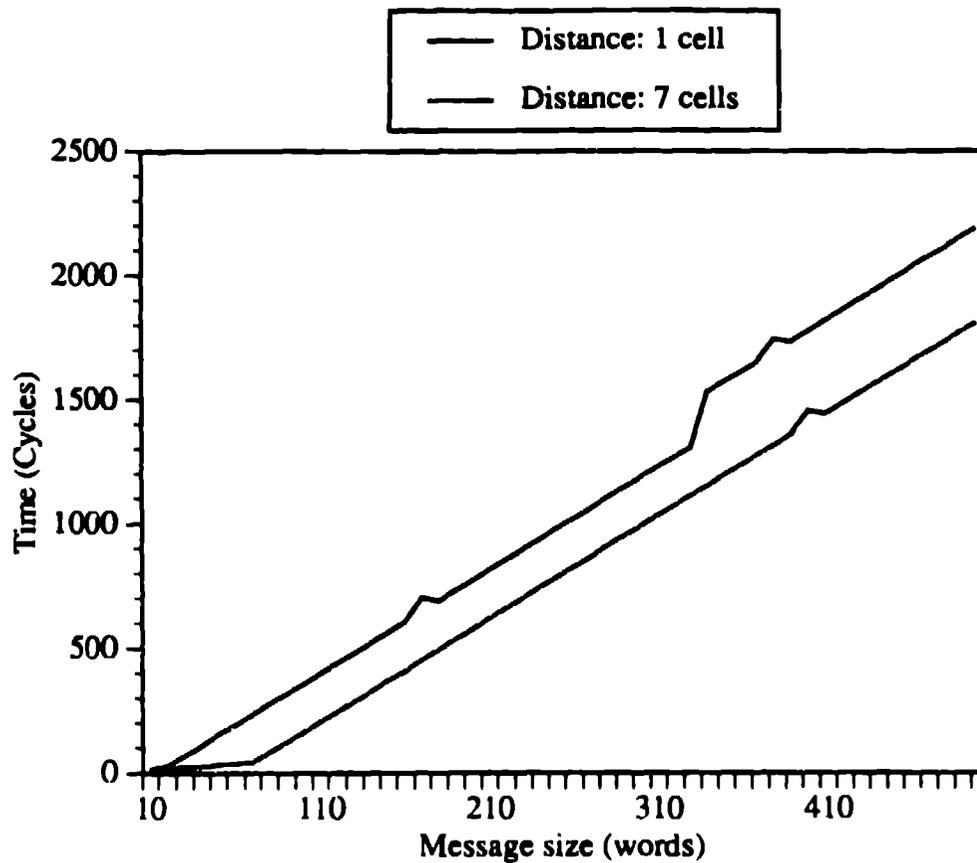


Figure 3: Graph of message size versus message sending time when the sender is faster than the receiver.

To determine the effect of longer networks, we ran and timed several versions of a simple send and receive program that varied the distance between the sending node and the receiving node. For one set of these programs, the sender sent messages faster than the receiver could receive them. For another set of programs, the sender and receiver operated at the same rate. See Figures 3 and 4 for graphs of the results. From these graphs you can see when the sender was faster than the receiver, the distance between the sending and receiving nodes made more difference. The additional buffer space on the intermediate nodes could store more data so the sender that was further away finished faster, but the slope of the graph is the same regardless of the node distance. Once the queues had initially filled, the sender sent at the same rate regardless of the distance between the sender and the receiver. When the sender and receiver were working at the same speed, the distance between the nodes made a much smaller difference. When the sender and receiver are communicating at the same speed, the messages is not really taking advantage of the buffers added by the longer network. In this case the only change dependent on the network length would be additional time it takes to travel over twice as many links.

The node programs address other nodes to specify the destination when sending data over networks. Nodes are addressed by the row and column numbers of their location in the processor array. After the

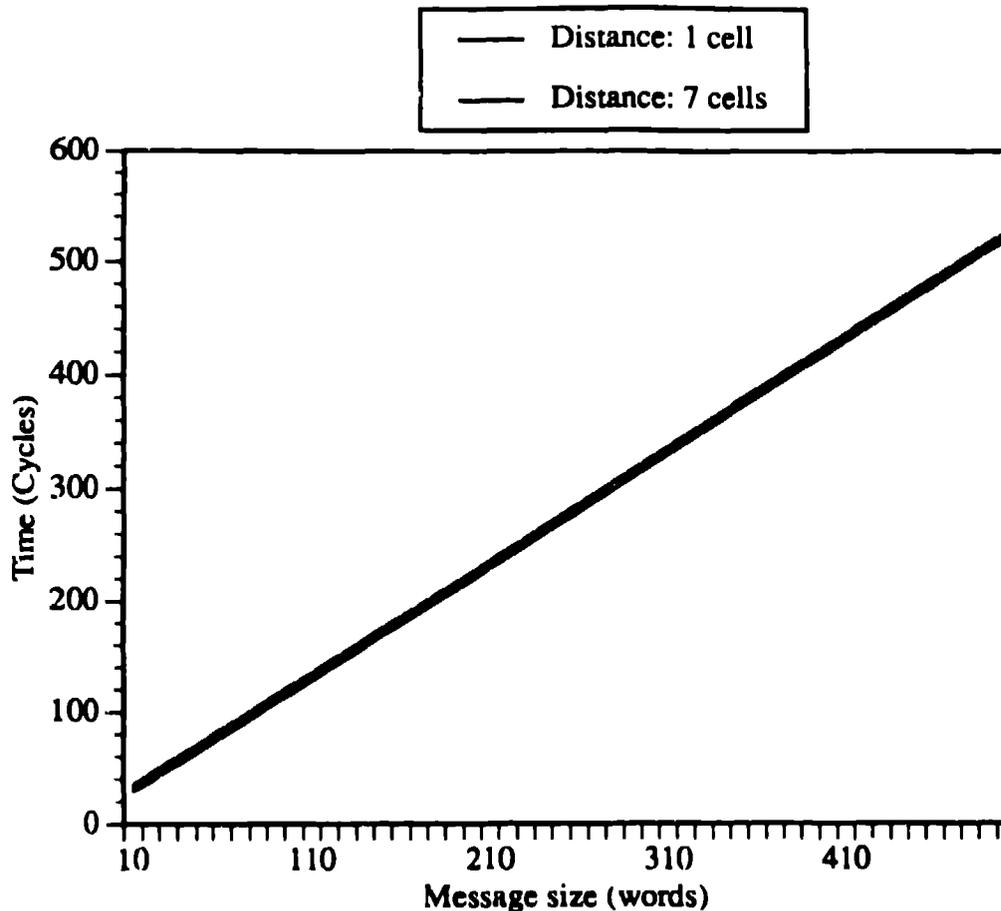


Figure 4: Roundtrip message time when the sender and receiver are operating at the same speed.

user's program has been spread out to insert monitoring nodes, the addresses of destination nodes in the node programs must be adjusted. This address transformation is performed at run time by multiplying the row and column numbers of the destination by 2 (or equivalently shifting these numbers 1 bit to the left), which adds a small overhead to the setup of a connection.

4.2 Sample data accuracy

The monitor program reads the current network queue sizes from 4 control space registers. The length of each queue is stored in four bits, and these bits are striped over the 4 control space registers. Since the queue lengths can change with each cycle, it is possible for a queue length to change while the control space registers are being sampled. The queue lengths are constrained to change at most by one with each clock tick for single word transfers (queue lengths can change by two per each clock tick for double word transfers), so the percentage of errors is lower than it would be if the queue lengths could change to any value. We have also observed that the queue lengths tend not to change continually with each clock tick, so the odds of sampling during a queue length transition are low.

Over the period the program is reading the control space registers, the length of a queue may vary by 1 for single word transfers. Consider the four number transitions that are allowed on the 0 to 8 counter,

e.g. $8 \rightarrow 7 \rightarrow 6 \rightarrow 5$ or $4 \rightarrow 4 \rightarrow 5 \rightarrow 4$. There are 203 valid four number transitions over the range of 0 to 8. The monitor program reads the queue length registers one bit at a time while the actual queue length goes through one of these four number transitions. The queue size read is *inaccurate* if it was not one of the queue length values during the 4 cycles it took to read the queue length registers.

The monitor program reads the queue lengths starting with the most significant bit, so the worst inaccuracies occur when the top bit changes in transitions such as $8 \rightarrow 7 \rightarrow * \rightarrow *$ or $7 \rightarrow 8 \rightarrow * \rightarrow *$. Downward transitions from 8 will result in numbers greater than 8. The monitor program truncates such numbers to 8. In this case the queue size is accurate, because 8 one of the queue length values during the time it took to read the queue length register. However, transitions that start from 7 to 8 are bad because the two most significant bits read will be 0 and the resulting value read will be at most 3, at least 4 off from either 8 or 7. The next worst inaccuracies occur in transition such as $* \rightarrow 4 \rightarrow 3 \rightarrow *$ or $* \rightarrow 3 \rightarrow 4 \rightarrow *$, because the the middle two bits will be inaccurately set to 1 or to 0. For transitions of the form $* \rightarrow 4 \rightarrow 3 \rightarrow *$ the resulting number read will be 6 or 7 at least two off. For transitions of the form $* \rightarrow 3 \rightarrow 4 \rightarrow *$ the resulting number will be 0 or 1 at least 2 numbers off.

Without more accurate information about the distribution of queue length changes, we cannot determine the sampling error introduced by misreading queue lengths.¹ However, as stated earlier, for an understanding of inter-processor communication, the state of the queue is more important than the actual number of words in the queue, so the current sampling technique is adequate. Designers of future system may want to chose a different design for the status registers that allows an atomic read of the queue length registers.

5 Using queue length data

After the monitored program has been executed, the programmer is left with large files of numbers. To get any useful information from this data, the programmer needs tools to effectively manipulate and display the queue size data. Much work has been done on displaying monitored data (e.g. Tapestry [Mal90], BEE [Bru90], and others). We have created two relatively simple programs to aid in the post-mortem understanding of the collected data.

The first program `xmon` is an X window program that examines the data files and displays the user's array and input logical channel queues. It steps through the execution of the monitored program and displays the queue lengths at each point in the original execution. Figure 5 shows a screen image of `xmon` in action. In that figure, `xmon` is working on a program that ran on a 4×4 array of nodes. There are two distribution networks snaking through the array, and there are networks between each pair of physically adjacent nodes. Figure 6 shows the logical arrangement of the networks used in that array program. The queues in the `xmon` display are shaded to reflect their length. The display reflects the state of the queues at the point in the execution of the original program shown by the time bar at the bottom of the screen. The time bar is labeled from 0 to the end of the original execution and is filled to the currently displayed execution time (Figure 5 shows the state of queues after the program has executed for 74,080 cycles). The user can step to the next or previous state of the queues. The user can also run in *animate* mode, where `xmon` automatically displays each consecutive state.

By looking at the changes in queue length, the programmer can visualize the flow of data in her

¹ The reason for this striped arrangement of the queue length registers was that it simplified the design of the communication agent (which contains a number of speed critical paths). In the original processor design, the queue length registers were just testing registers and so did not need to be read quickly.

program. For example, she can see periods where one set of networks is more active than another set or where a set of queues is consistently full. While `xmon` is helpful in visualizing the global data flow, it is not so good for presenting the fine details of the communication patterns. `xmon` only shows *changes* in queue length, so when running in animate mode, the time bar does not move forward smoothly. If the queue lengths did not change for a long time, the time bar will jump ahead to the time of the next queue length change. If there are short alternations of communication and computation, the short jumps in the time bar are easy to miss.

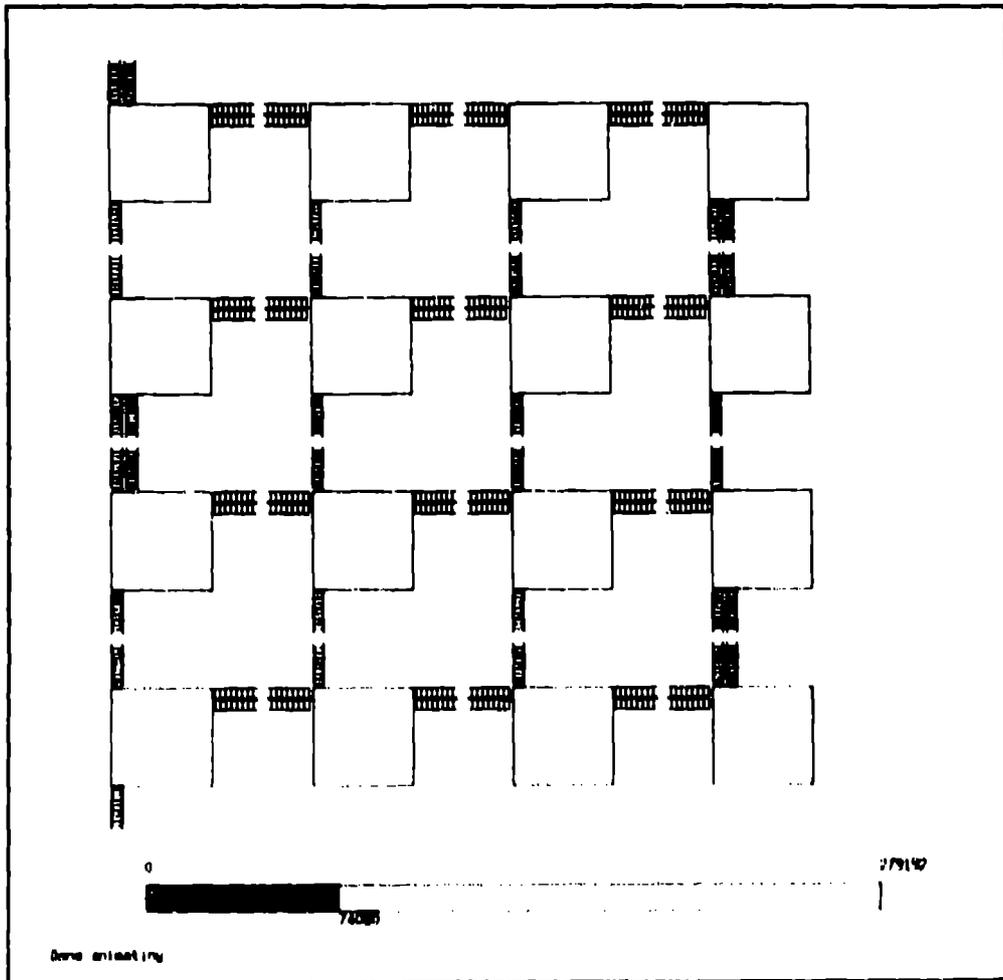


Figure 5: This is a picture of `xmon` displaying the data obtained from running the AI matrix multiply program.

By graphing queue length versus monitored execution time for each node, the programmer can enlarge interesting areas to see the fine details of the communication pattern. We created a tool called `mon-graph` that transforms the monitored data into load and data files for `gnuplot`. By adjusting the X range in the load files, the programmer can create graphs that zoom in on the interesting subsequences. Of course this creates many graphs, but by using a postscript previewer like `ghostscript`, the programmer can quickly examine many different queue length graphs.

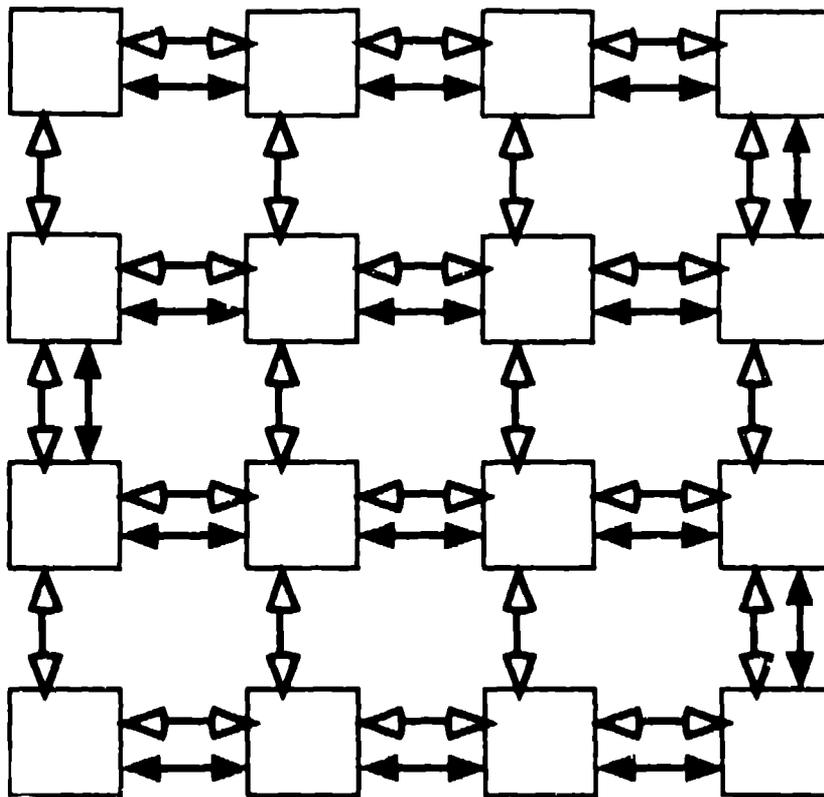


Figure 6: A picture of the logical connections used in the in the AL matrix multiply program.

5.1 Examples

Here we show an example of using `xmon` and `mon-graph` to evaluate the monitored data from two matrix multiply programs. One was an AL program which uses memory communication (see [Tse89] for details). The other program uses systolic communication and was generated by another mapping tool [Rib90]. Both programs run on a 4×4 array of iWarp nodes (using a preliminary version of the runtime system and version 2.3 of the C compiler). Figure 5 shows the queue states of the AL program displayed by `xmon`. By tuning `xmon` for each program, we can distinguish the main phases of the programs: loading, computing, and closing down.

From `xmon` the communication patterns of the two programs looked similar. In both programs the lengths of the active queues quickly changed between empty and partially full. By using `mon-graph`, we were able to zoom in on the computation phases of the two programs. Figures 7 and 8 show the resulting graphs. In the AL program queues regularly alternated between being empty and oscillating between being empty and having one entry. From knowing the block oriented nature of AL, this is expected. The program alternates between exchanging blocks of data and computing on those blocks. While computing, the program exchanges no data, so the queues stay empty. In the systolic program, communication appears to be interspersed with the computation. The first part of the graph shows the queue has length one most of the time, dropping to zero at regular intervals. The second part shows a queue length of zero most of the time, jumping to one at regular intervals. Since the systolic program sends and receives data in one word intervals as it is computing, this pattern is reasonable. In the first half, the program is consuming

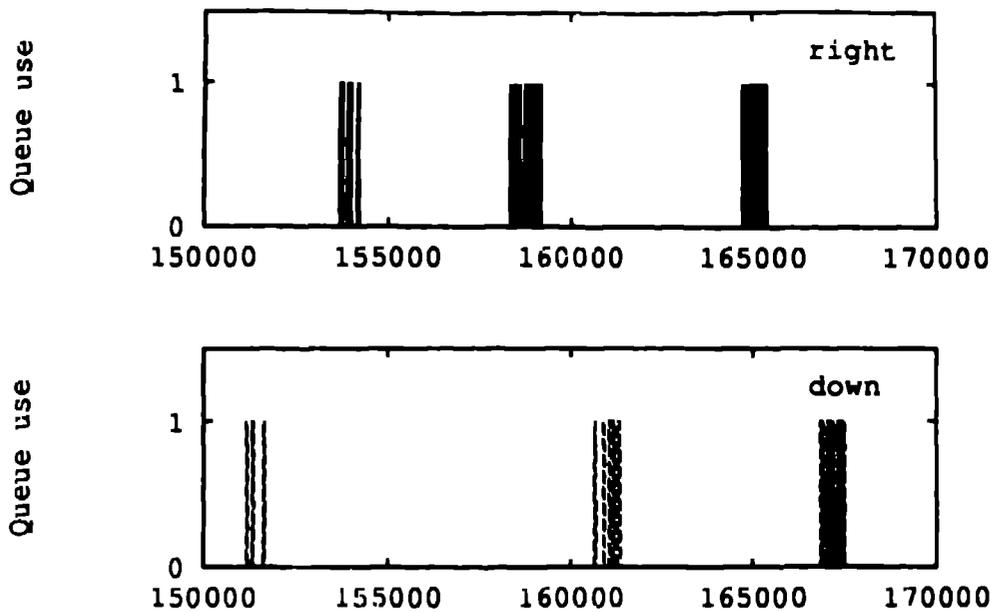


Figure 7: A graph of the queue lengths on node (1,1) over time during the execution of the AL matrix multiply program.

data slower than it is being sent, so the data stays on the queue. In the second half, the program is trying to consume data faster than it is being sent, so data is quickly read out of the input queue.

5.2 The next step

In the previous example, we gathered data on inter-processor communication and saw that there no message was blocked in the communication system. If we had discovered a blockage, the display program would have identified the node(s) that were involved in the communication, but if the nodes exchange multiple messages, the user would have to identify which message was affected. Without data from the user nodes, there is no way to relate the queue sizes and times directly with the user node activity. For some programs, the information from the monitoring nodes alone is sufficient to understand the communication patterns, but for complicated or unfamiliar programs, relating the queue sizes back to instructions in the node program would be very useful. Specifically, we would like to link queue lengths with the execution of user instructions and with the user node state (either active or spinning). One of the challenges that such a tool must overcome is that any more detailed recording of information may perturb the program execution significantly.

6 Conclusions

Our experimental monitor shows that a processor that provides a close coupling between communication system and the computation units can be programmed to serve as a performance monitor. Although this idea is simple, it is quite useful. Using idle nodes to capture information about the inter processor

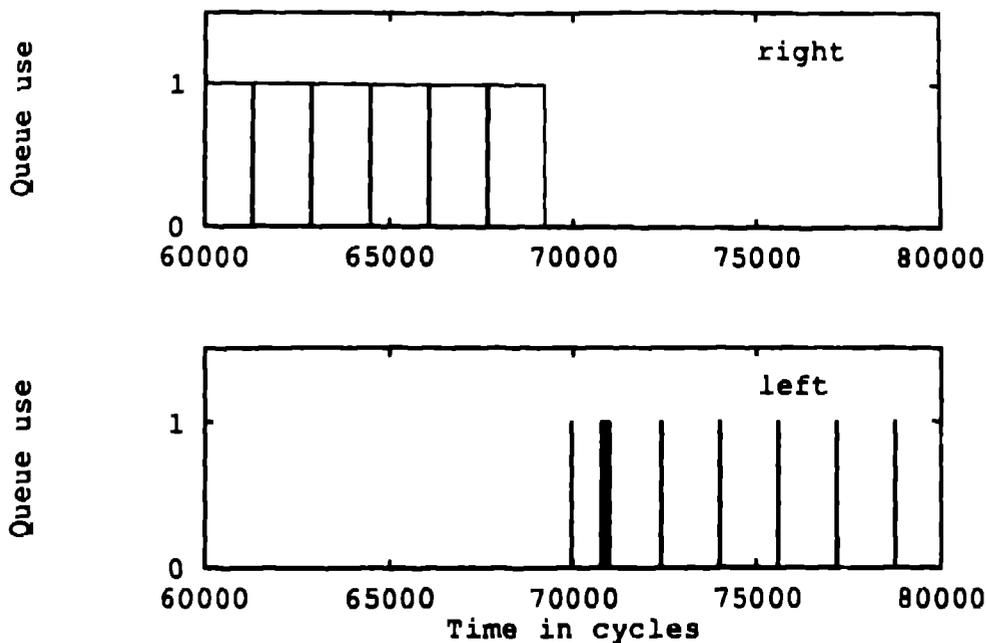


Figure 8: A graph of the queue lengths on node (1,1) over time during the execution of the systolic matrix multiply program.

communication does not require any additional hardware, and when the programmer is not interested in monitoring, all resources in the system are available to execute user programs. The information gathered by such a monitor can be used to identify communication bottlenecks, and building a visualization system is not difficult.

Future designers of integrated components that are to serve as building blocks for parallel systems may want to consider this usage of a processor during the design phase. Although we did not encounter any insurmountable problem in implementing the monitor, there are several rough edges (like the need to read the queue length registers sequentially).

The close coupling between the communication agent and the computation agent of the iWarp component is crucial for the successful operation of the monitoring program. It is not clear if other systems (where communication and computation are not integrated as closely as in iWarp) can build a similar tool as easily. However, the advantages of tightly integrated communication and computation are recognized by other parallel systems, so future systems may provide the foundation for similar hybrid monitoring.

References

- [BC⁺88] S. Bokar, R. Cohn, G. Cox, S. Gleason, T. Gross, H.T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P.S. Tseng, J. Sutton, J. Urban, and J. Webb. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of the Supercomputing Conference*, pages 330-339, 1988.

- [BCC⁺90] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, C. Peterson, J. Sussman, J. Sutton, J. Urbanski, and J. Webb. Supporting Systolic and Memory Communication in iWarp, CMU-CS-90-197. Technical report, Carnegie Mellon University, School of Computer Science, 1990. Revision of a paper that appeared in the 17th Annual Intl. Symposium on Computer Architecture, Seattle, 1990, pp. 70-81.
- [BG91] B. Baxter and B. Greer. Apply on iWarp. In *Proceedings of the 5th Distributed Memory Computer Conference*, 1991.
- [Bru90] B. Bruegge. BEE: a Basis for Distributed Event Environments: Reference Manual, CMU-CS-90-180. Technical report, Carnegie Mellon University, School of Computer Science, 1990.
- [FGOS92] A. Feldmann, T. Gross, D. O'Hallaron, and T. Stricker. Subset Barrier Synchronization on a Private Memory Parallel System. In *Proc. Symposium on Parallel Algorithms and Architectures*, San Diego, June 1992. ACM.
- [Hin91] S. Hinrichs. *Programmed Communication Service Tool Chain User's Guide*, 1991.
- [HWW89] L. G. C. Hamey, J. A. Webb, and I. C. Wu. An Architecture Independent Programming Language for Low-Level Vision. *Computer Vision, Graphics, and Image Processing*, 48:246-264, 1989.
- [Mal90] A. Maloney. *Performance Observability*. PhD thesis, University of Illinois, Urbana-Champaign, 1990.
- [O'H91] D. O'Hallaron. The Assign Parallel Program Generator. In *Proceedings of the 5th Distributed Memory Computer Conference*, 1991.
- [Rib90] H. Ribas. *Automatic Generation of Systolic Programs from Nested Loops*. PhD thesis, Carnegie Mellon University, 1990.
- [SK90] M. Simmons and R. Koskela, editors. *Performance Instrumentation and Visualization*. ACM Press, New York, 1990.
- [Tsc89] P. S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, May 1989.

**Debugging a Parallel Program:
Capturing Inter-Processor Communication in
an iWarp Torus**

**Thomas Gross
Susan Hinrichs**

**Carnegie Mellon
School of Computer Science**

October 9, 1992

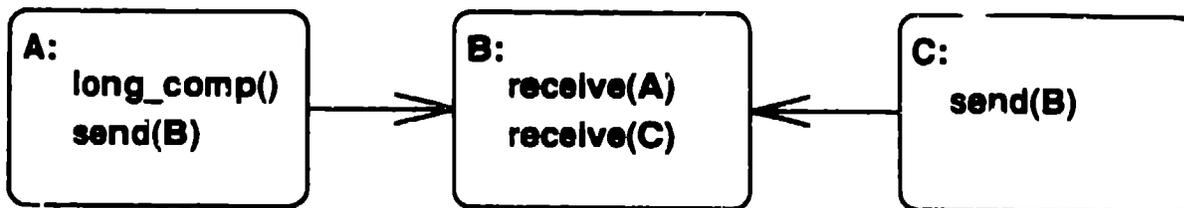
Overview of talk

- Introduction
- Target machine
- Hybrid monitoring approach
 - Implementation
 - Analysis tools
 - Evaluation
- Summary

Introduction

Performance debugging hard for single processor and even harder for multiprocessor

Need to gather execution information for complete system view

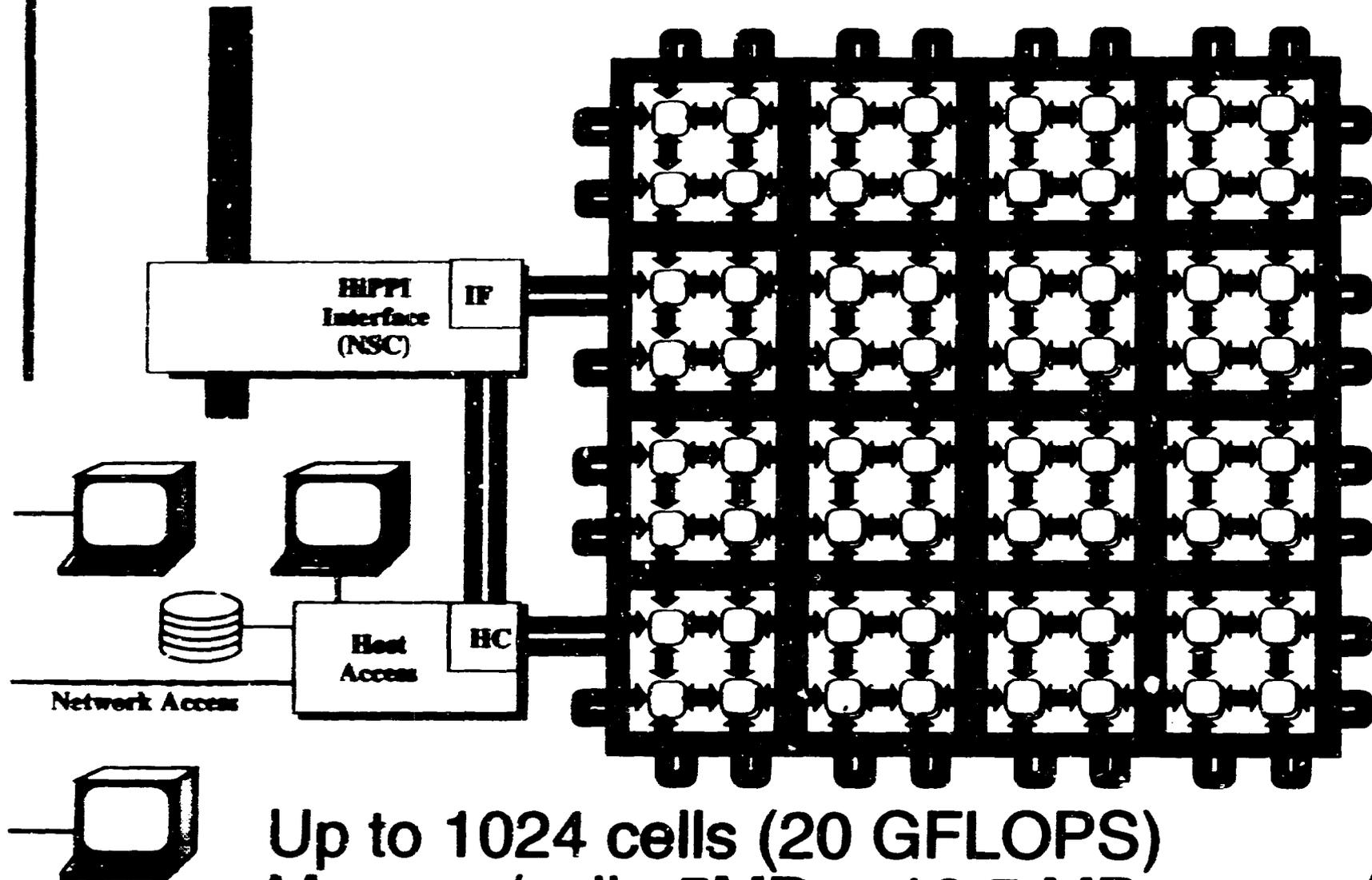


Message timing important to understand performance

Target machine

iWarp, an array of private memory cells connected in a 2 D torus by 40 Mbyte/sec busses

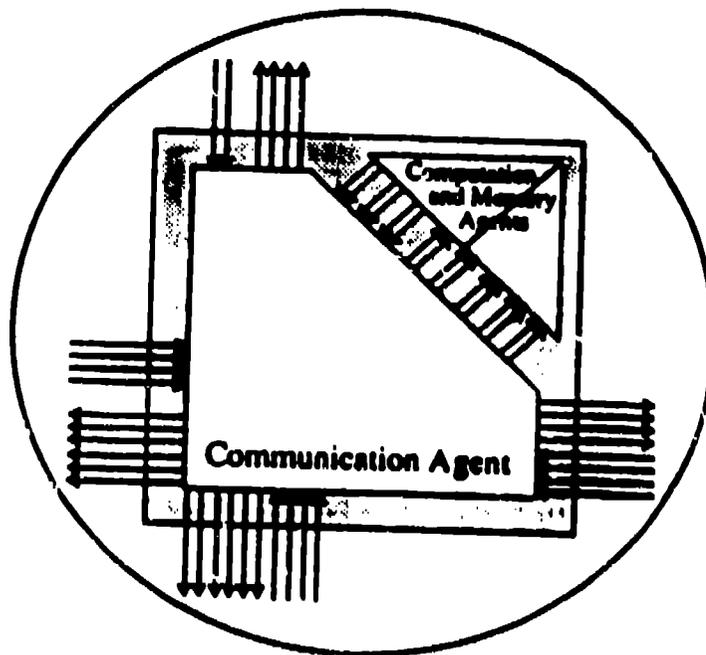
64 cell system: 1.2 GFLOPS (s.p.)



Up to 1024 cells (20 GFLOPS)
Memory/cell: .5MB -- 16.5 MB

iWarp cell

Each cell is a tightly integrated pair of communication and computation agents. The agents can operate synchronously or asynchronously.



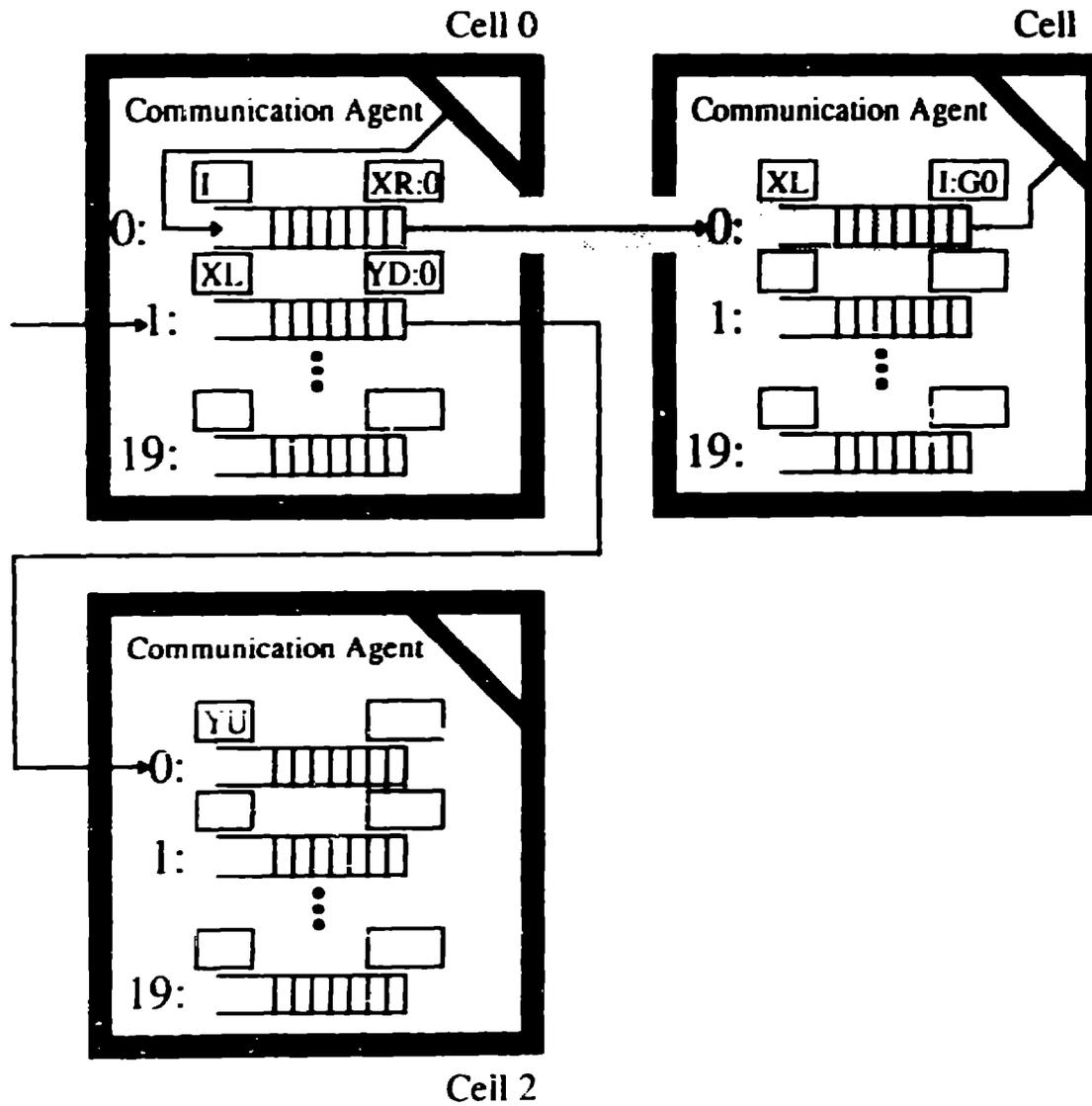
Logical channels

Hardware support for a finite number of *logical channels*. Logical channels enable multiple high speed connections over the same physical channel.

Each connection uses logical channel buffers at the source and destination cells. Logical channels can be chained together to form *pathways*, direct connections between distant “neighbors”.



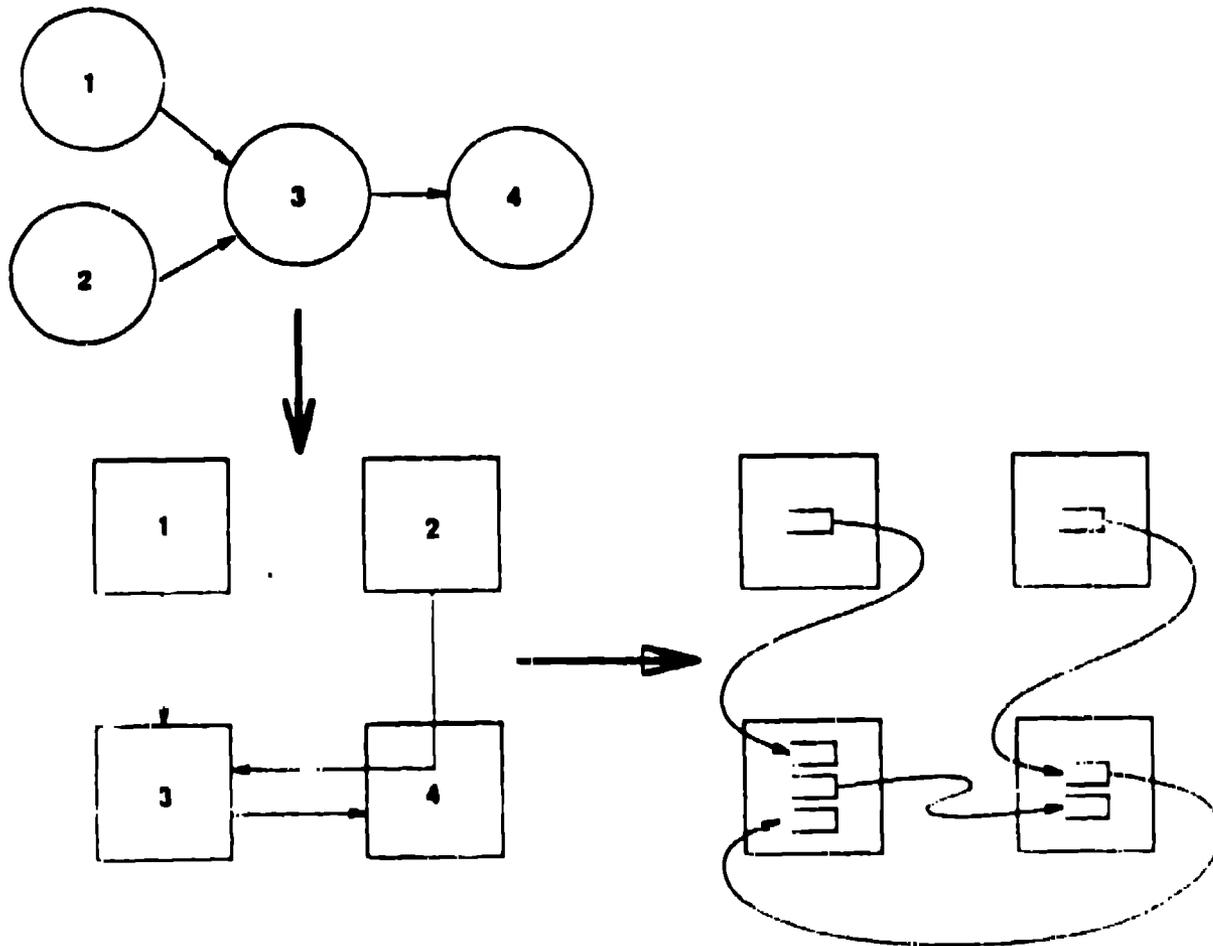
A Logical Channel Connects Two Neighbor Cells



Program communication

Cell programs communicate by sending messages over pathways.

Programming tools map processes to processors and communication networks to pathways.

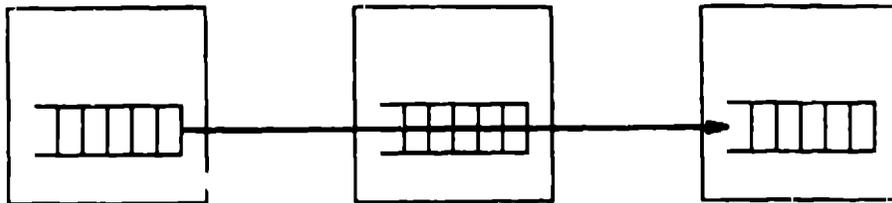


Monitoring implementation extremes

- **Monitoring hardware**
 - Very accurate
 - Steals no resources from monitored program
 - Expensive: requires new, special purpose hardware
- **Profiling software**
 - Inexpensive: requires no new hardware
 - Less accurate

Hybrid monitoring approach

Communication connections through intermediate cells have logical channels assigned on those cells.



Intermediate cell can read status registers to determine length of logical channel.

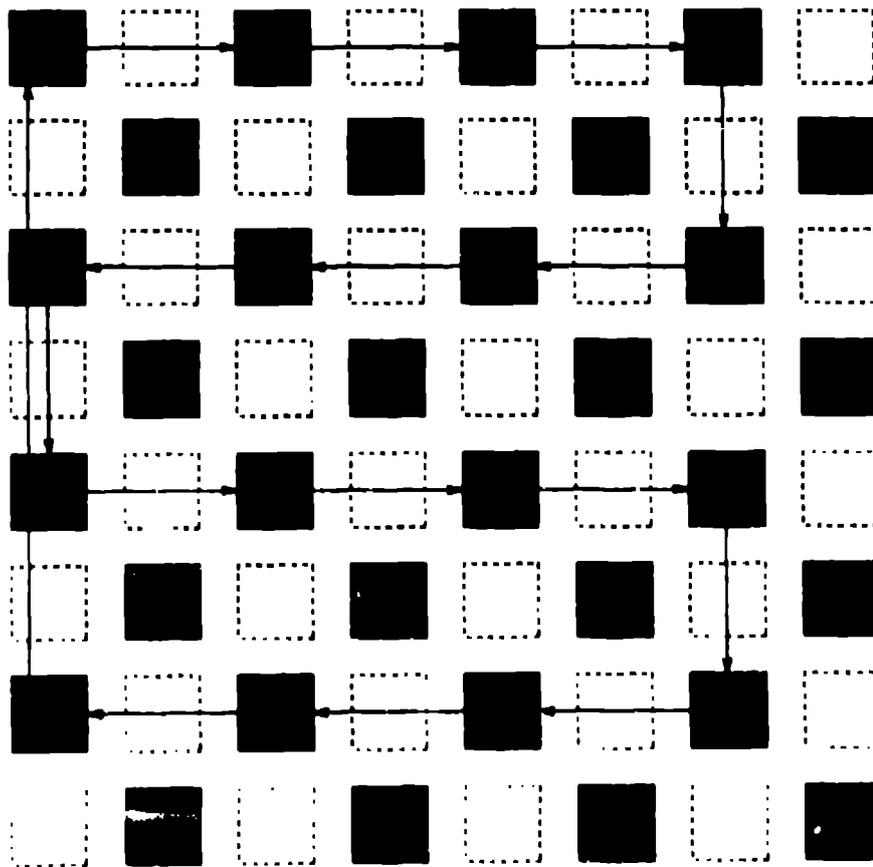
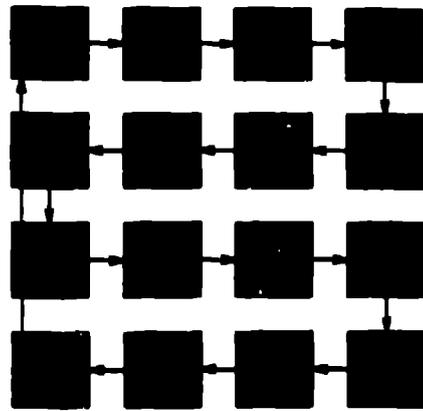
Amount of data in the logical channel reflects the communication pattern.

- empty
- not empty / not full
- full

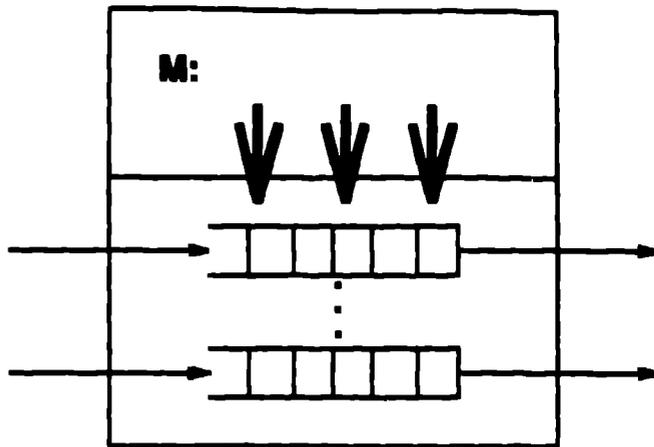
Implementation outline

- **Compile user program with monitoring flag**
- **Gather data on separate cells during execution**
- **Dump data to host system after execution**
- **Post-mortem analysis of data on host system**

Communication networks



Monitoring cell



```
while (executing)
    log_channel_lengths()
dump_log()
```

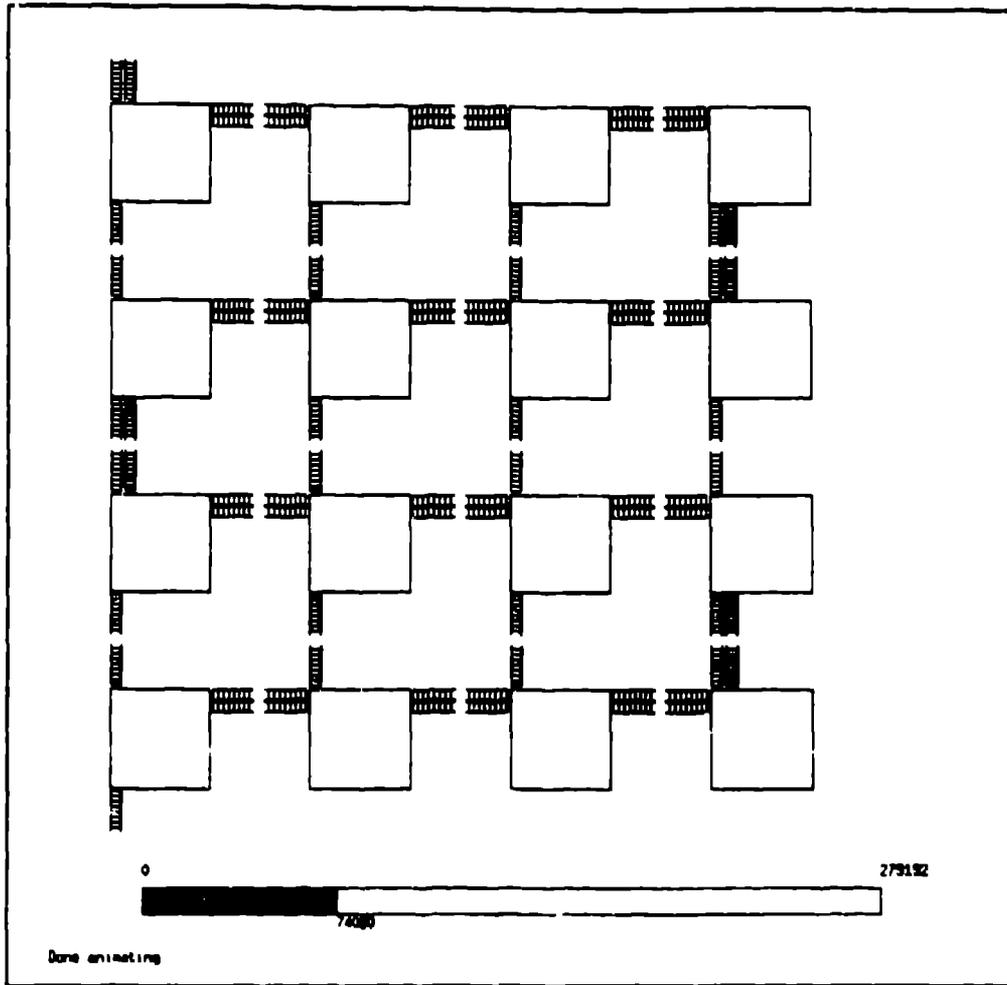
Analyzing profile data

Two simple programs to post-mortem analyze the monitored data.

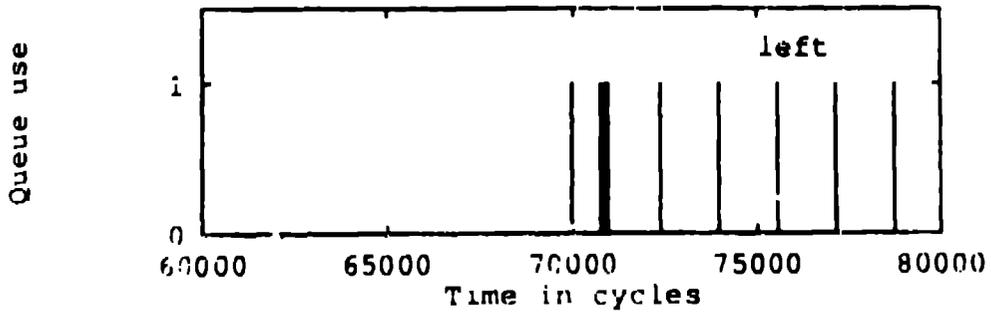
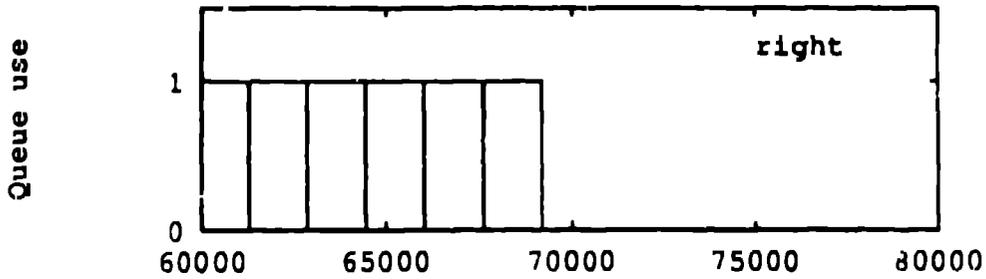
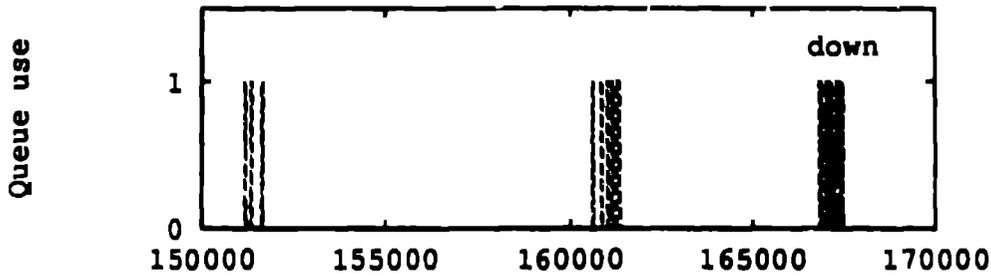
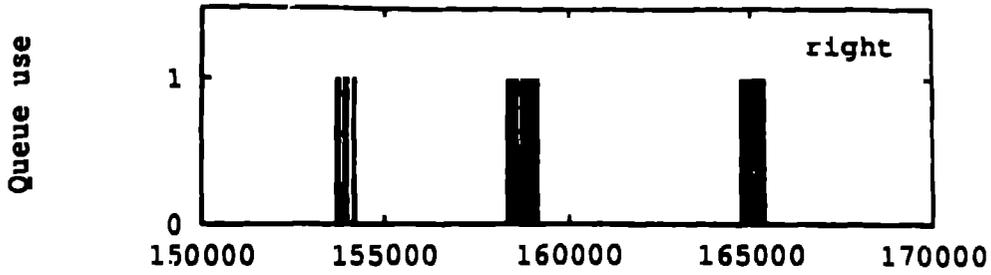
xmon - replays the logical channel lengths

mon-graph - statically graphs queue lengths on a single cell

Xmon



Mon-graph



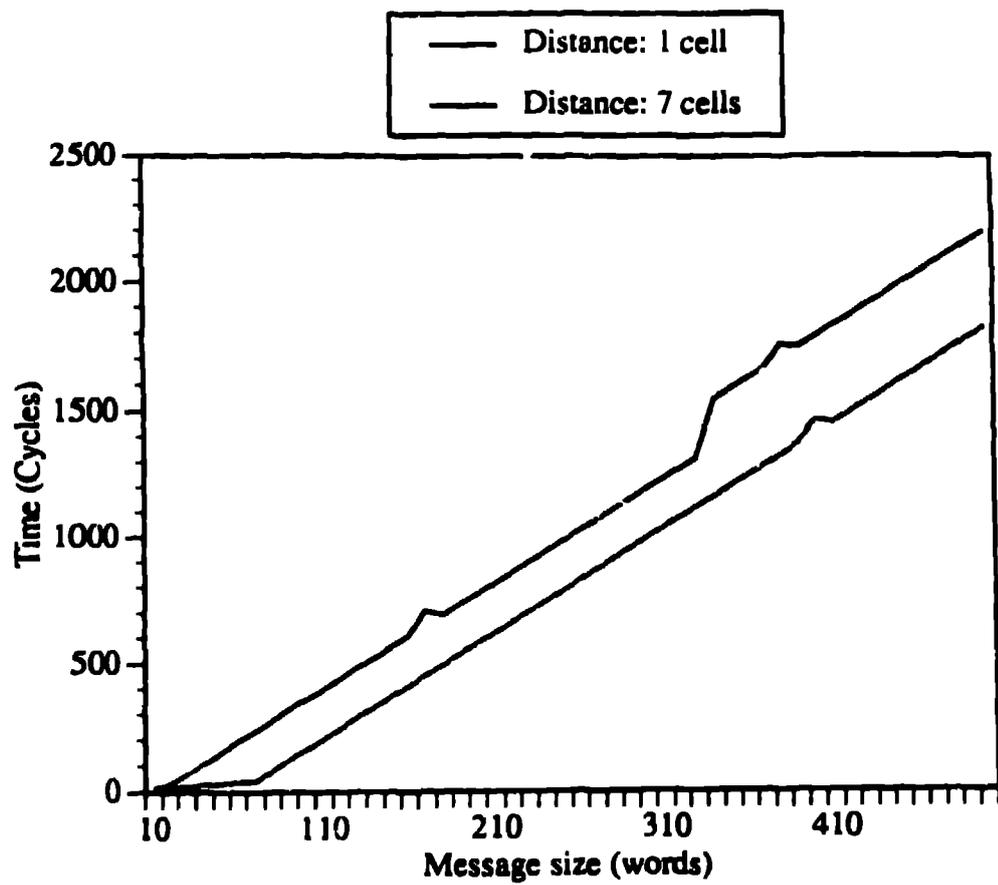
Hybrid monitoring benefits

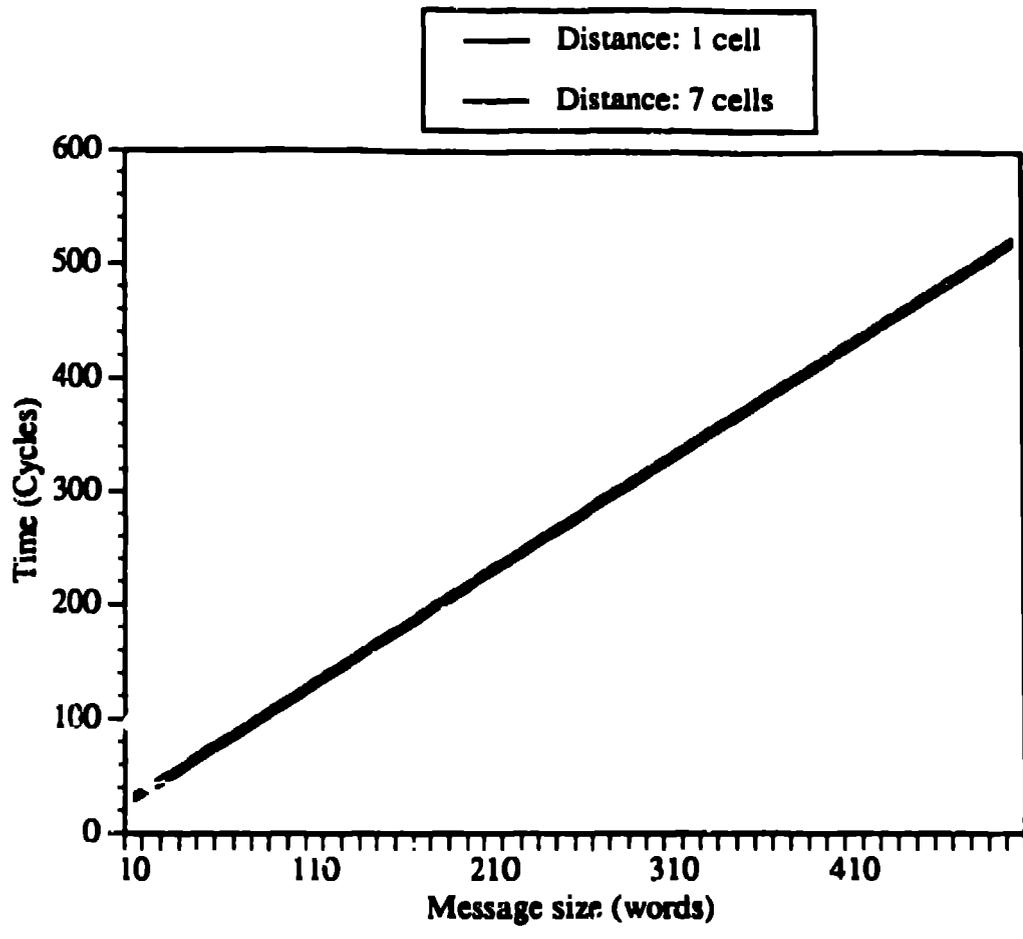
- **No specialized hardware**
- **Monitoring programs use separate resources from user program**
- **The user makes no changes to the code**

Evaluation

Possible sources of skew

- Doubling communication routes
- Changing destination cell address
- Accurately reading logical channel length registers





Summary

Tightly integrated communication and computation enables effective communication monitoring

Hybrid approach provides reasonable cost / accuracy tradeoff

Another usage model to consider in the design of such multiprocessor systems.

The Application of Code Instrumentation Technology in the Los Alamos Debugger

Fast Conditional Software Watchpoints, Integrated Performance and Coverage Analysis

**Jeffrey S. Brown, Richard Klammann
Los Alamos National Laboratory
jxyb@lanl.gov, rmk@lanl.gov**

Introduction

This paper will discuss applications of code instrumentation technology in the Los Alamos Debugger (LDB). By code instrumentation, we mean debugger modifications of the traced process to monitor execution in order to acquire information about the process while it is running. The objective is to implement debugger functionality in a way that enhances usability while remaining non-intrusive. This paper will discuss the application of code instrumentation in the implementation of fast conditional software watchpoints, and integrated performance and coverage analysis. We conclude by suggesting future application areas.

Watchpoints

Watchpoint is a debugger feature that causes a traced process to stop when a data element (or range) is modified possibly subject to a user-specified condition on the modified data. In the absence of hardware support (which is the case on the Cray YMP) the traditional implementation is to stop the traced process frequently to check the condition in the debugger while the process remains stopped. While this approach will certainly work, impact on performance of the traced process is such that the user runs out of patience long before the watchpoint condition is satisfied. With hardware support, the traced process stops only when the data element (or range) is written to. This is much more efficient than the frequent stopping of the traced process, but doesn't help when the user specifies a conditional watchpoint. The traced data element (or range) can potentially be written to millions of times before the condition of interest is satisfied again causing a severe impact on performance of the traced process.

Fast conditional software watchpoints are implemented in LDB by modifying the traced process to check the watch condition on-the-fly. This implementation allows the user to specify a complex condition on the data element being watched while minimizing the impact on performance of the traced process. Detection of the watch condition requires that the user run the code twice. The first run checks the condition at every subroutine entry and exit (*watch all*). Once the offending subroutine is located, the second run checks for the condition at every line/label (*watch in*). The granularity of the second pass is a function of compiler optimization level. The two-pass approach is necessary due to the time and code space required to instrument the entire program at the line/label level. Entry/exit level code instrumentation is fast and requires little code space in the traced process to implement.

User Interface

Watch All

Here's an example of the sort of problem watchpoints are good at solving:

```
COMMON X(2,50), A(100), Y(2,50), B(100)
...
READ(2) A
100 CONTINUE
...
CALL SSWAP(100,X,2,Y,2)
...
C = A(1)
```

Suppose you're debugging this code and discover that, at the point of the assignment to C, A(1) has been trashed. You rerun the code from the beginning, stopping after the READ, and verify that the value of A(1) is correct at that time. So somewhere between the read and the assignment, A(1) is being corrupted.

Note that SSWAP must be the source of the data corruption. SSWAP is a Cray LIBSCI routine that swaps the contents of two vectors. In this case it appears that the programmer wished to swap the first row of X with the first row of Y, however, the first argument to SSWAP should be the number of elements to swap, not the size of the array, so we've inadvertently swapped the odd elements of A with the odd elements of B, as well swapping portions of X and Y.

In this context the problem clearly needs no tools to be solved. In practice these lines of Fortran are usually in separate routines, and the ellipsis are replaced by thousands of lines of code. This is not a trivial bug to detect. Most programmers will scatter print statements randomly about their code and home in on the bug after a day or two.

Here's how the experienced LDB user tackles this problem: (We'll use "LDB>" as the LDB prompt; in real life the prompt would be the name of the currently active subroutine in the code being debugged.)

```
LDB> run to $100
LDB> watch all for change(A(1))
LDB> run
```

First we run to a point just after the read of A, where we know the contents of the array are correct. Next we tell LDB to instrument our code by patching in instructions at each subroutine entry and exit to detect a change in A(1). Lastly we continue execution of the code and wait for a diagnostic. In this case LDB will tell us that a change in A(1) was detected on the exit from SSWAP(). QED.

The all clause is a new variant to LDB's watch command; it replaces watch from, which

attempted to patch all breakpointable lines and labels in a potentially huge call tree. The new watch all is fast, but can only resolve to the subroutine level. In this case that was all we needed; checking the parameters to SSWAP revealed the problem. In other cases we may need to rerun the problem with a watchpoint set in the routine indicated by watch all

Watch In

Here's a harder bug. Say you've just integrated code from your development team into a rather large application, and now a part of the code that nobody touched is broken. You put the application under LDB, run it, and get the following:

```
user process received signal 7 (Error exit)
user process stopped at program counter: 366pb = REDUCE() + 4pb
```

Error exits are caused by executing zero instructions, which we verify by listing the CAL surrounding the error:

```
LDB> di reg(pc)-1 for 7

00000365pb: 025 2 02          B02      A2
00000365pc: 025 6 01          B01      A6
00000365pd: 024 0 01          A0       B01
00000366pa: 000 000          ERR
00000366pb: 000 000          ERR
00000366pc: 000 000          ERR
00000366pd: 000 000          ERR
```

This looks like a classic case of code being overwritten by data. To solve this, we'll set a watchpoint to stop as soon as we detect that this code has been clobbered:

```
LDB> watch all for mem(366b) .eq. 0
LDB> rerun
```

From this we get the following diagnostic:

```
watchpoint condition met in subroutine: BURN
(detected at entry to XLATE)
```

In this case BURN is a very large subroutine and XLATE is called somewhere in the middle of it -- the large granularity of the watch all command still leaves us with a difficult problem. Fortunately watch can hone the problem down further, given that we know the routine and approximate location. Let's try the following:

```
LDB> watch in BURN from 755L to 834L
LDB> rerun
```

The above watch command requires some explanation. The `in BURN` clause tells LDB to instrument routine `BURN` only, and to instrument all breakpointable locations -- lines and labels. This process can be slow if the routine is large, which `BURN` is, so we've specified a code range to instrument with the `from` and `to` clauses.

The `to` clause is simply the line number of the call to `XLATE`, in this case line 834. We know from the watch all result that we needn't instrument past that point. A missing `to` clause would cause LDB to instrument through the end of `BURN`.

The `from` clause was obtained by looking for the last subroutine call within `BURN` prior to the call to `XLATE`. Why the previous subroutine call? A negative inference available to us from having run watch in is that the test condition (code zeroed) was not true for a routine called prior to executing `XLATE`. Since the last previous routine was called at line 755, that's where we'll start the search. A missing `from` clause would cause LDB to instrument from the beginning of `BURN`.

Perhaps the most important point to notice in the above watch command is the specified condition, or rather the lack of it. If no condition is specified, LDB will reuse the previous condition. (And if there is no previous condition, LDB will generate a trivially true condition.)

The result of the second watch narrows the problem down to an assignment into a pointered array. Although the array references are all in bounds, the pointer itself is seen to have the octal value 366b; evidently the pointer has been corrupted. Since the pointer points into code space, any assignment to the pointee will overwrite code.

Implementation

Conditional watchpoints are implemented in LDB by patching the traced process to check for the watch condition on-the-fly while the process continues running. This is accomplished in two stages, code generation and process instrumentation.

Code Generation

LDB contains its own mini-compiler, capable of generating code for most Fortran conditional expressions. The compiler supports the standard relational operators, and uses a short-circuit technique to evaluate boolean combinations of relational clauses. Arithmetic expressions are limited to simple infix operators, bit manipulation intrinsics, an indirection function, and a register access intrinsic. Integer and single precision real are the only data types supported. Limited support for C-language syntax is available, including C-style array references, bit operators, and pointer dereferencing, however non-word data types such as characters and structures are not supported.

The compiler is simple, sitting on top of the LALR (YACC generated) parser and allocating address and scalar registers in dual stacks. Optimization is done via repeated passes through a peep-hole optimizer, and by recognizing indirect addressing situations during code generation. Machine code laced with relocation information is generated at the time the watch command is

processed; executable text is relocated and patched into the target executable as part of run command processing. No use is made of CRI compilers, assemblers, or loaders.

Process Instrumentation

On-the-fly checking of the watch condition involves re-routing program execution to check the condition at regular points in the program. In our initial implementation, we attempted to instrument at the line/label level for a subroutine or subtree as specified by the user. The subtree could be as large as the entire program if the user called for a subtree with the main program as the root. While this technique found the watch condition in one pass and worked fine for small programs, the time required to do the instrumentation and code space required in the traced process for large programs rendered this approach infeasible. We settled upon an implementation that quickly instrumented the entire program at the subroutine level while requiring little code space in the traced process. Once the offending subroutine was located, a second pass could effectively instrument at the line/label level in that subroutine only, and home-in on the offending code at a granularity dependent upon the optimization level used in compiling the code being watched. Los Alamos uses the CRI CFT77 compiler on UNICOS configured to always generate symbol tables and generate code at an intermediate optimization level which causes symbol table lines/labels to be generated at optimization block boundaries.

The first step in processing the LDB run command is to instrument the traced process with all active breakpoints and watchpoints. The mechanics of watchpoint instrumentation depend upon whether the user is doing a *watch all* or a *watch in*.

For either watch option, the first instrumentation step is to patch in and relocate the pseudo machine code generated by the parser in the code generation phase. The traced process is then instrumented to re-route execution flow to execute the conditional code.

Watch all causes LDB to instrument all subroutine entries and exits in the traced process that contain "standard" entry sequences. Thus, even system library routines that were not compiled with symbols, but contain a "standard" entry sequence, are instrumented. The entry sequence is modified to cause a return jump to the conditional code. If the condition is satisfied, a zero instruction in the conditional code is executed causing a SIGERR signal to be generated. The traced process signal mask is set up to trap the SIGERR, the traced process becomes stopped, and control returns to LDB. The traced process program counter is then modified to the point where execution flow was re-routed so that the user sees execution stopped at the appropriate place in the process. Watchpoint (and breakpoint) instrumentation is cleared prior to returning control to the user.

Exit checking for the watch condition is accomplished by modifying the return address stored in register B00 (Cray convention) via the conditional code entry execution is routed through. Upon return to the "standard" entry sequence, the modified register B00 is stored in the traceback area associated with the subroutine to be restored at subroutine exit (normal entry sequence mechanics). The last instruction in a standard CRI exit sequence is a jump to the address stored in register B00, which was modified to cause a jump to the conditional code. Thus, no modification of exit sequence code is required. A stack of original register B00 contents is maintained in a scratch area in the traced process. If the condition is not satisfied during the exit

check, register B00 is restored to its original contents (at subroutine entry), the stack of saved return addresses is popped, and execution continues at the restored return address.

Instrumentation of the entire program using this technique can be accomplished quickly because only one pass is required through the loader-generated list of entry points and no symbol table access is required. The instrumentation scratch code space required is small because a standard template of conditional code works for all "standard" entry (and exit) sequences.

Watch in causes LDB to instrument all lines and labels in a specific subroutine (possibly within a user specified range) to route execution flow through the conditional code associated with the watchpoint. The machine instruction located at a line/label is overlaid with a return jump to a code template for the line/label. The code template executes the overlaid instruction then jumps to the conditional code. If the condition is satisfied the process stops as described above, otherwise a jump to the contents of register B00 occurs and execution resumes at the instruction following the one that was overlaid (register B00 was set in the return jump instruction). This technique requires about one word of code space per line/label instrumented for the code template, plus the code space required for the conditional code.

For subroutines containing thousands of lines and labels it is possible that the *cdbr\$end* code space scratch block will overflow (we allocate 512 words of code space in every UNICOS executable to be used by the debugger for instrumentation). If this occurs, LDB will prompt the user for the name of a code block or data block (must reside < 4MW) where instrumentation resumes. Good candidates for additional code scratch space on UNICOS are file conversion routines (IBM2CRAY, etc.) that are called only if a user assigns an attribute to a file that causes automatic data conversion during I/O. In practice these routines are never called and thus the code space is available. The choice is the users. The debugger does not assume the presence of particular code blocks sense they may go away with a future release of the operating system. The user may respond to the prompt with a <CR> which aborts the watchpoint instrumentation. The user-specified block is restored to its original contents when the watch condition is disabled (*rel watch*). You can see why this approach is not viable for an entire large program containing hundreds of thousands of lines and labels. The code space and cpu time required to do the instrumentation is prohibitive. But within the scope of one subroutine this approach is viable.

Watchpoint Summary

LDB's watchpoint facility provides a unique and efficient method for isolating data corruption bugs. The *watch all* command quickly instruments all entry and exit sequences; it is used to isolate the routine that is causing the corruption. A second run using the *watch in* command may be needed to isolate the problem code further. The *from* and *to* clauses are used to limit the expensive instrumentation done by *watch in*.

Integrated Performance Analysis

Performance analysis provides the user with statistical information about where CPU time was spent during a run of a program. With this information, the user can determine program bottlenecks and can often greatly enhance overall performance via small changes to specific areas in the program. A program that fails to deliver right answers in a timely way is little better than a program that produces wrong answers. Therefore, the analysis and tuning of program performance can be considered a form of debugging. Integration of performance analysis with the debugger combines statistical profiling with process control. A user can achieve fine-grained performance data, for example one pass through a do-loop, which is not possible via stand-alone performance analysis.

Cray Research performance analysis tools require that the user load with a special library (LIBPROF) to enable profiling. A subsequent run produces a profile data file that must be run through the CRI PROF and PROFVIEW utilities to produce a report. The Los Alamos Debugger integrates statistical profiling such that the user does not need to load a special library and invocation of the PROF and PROFVIEW utilities is automatic. Thus, performance analysis can be done on the production version of a code.

User Interface

Profiling is controlled by typing *profile* at an LDB prompt, followed by *on*, *off*, *clear*, *dump*, or *report*. Profiling data is accumulated for each run after profiling has been turned *on*. Sampling is disabled by turning profiling *off*. Data from previous runs may be erased with the *clear* option. Profiling statistics are generated by *report*, which puts the user into a PROFVIEW session to view the results. Further analysis can be done by generating the raw file appropriate for PROF input via *dump*.

The following example illustrates LDB's features. Say that we're building a multi-user database system in which the performance of the system as a whole is important, but in particular the performance of the UPDATE function is critical -- we don't want lock the entire system for very long each time we need to update our tables.

Profiling this code as a whole won't be very helpful since we require not aggregate times, but times relative to a particular activity. Knowing that routine ABC consumed 35% of our compute cycles does not tell us how important ABC is to our time-critical UPDATE function; perhaps UPDATE doesn't rely on ABC at all.

Here's how an LDB user would profile a particular logical segment: (For clarity we'll show the LDB prompt as [LDB>].)

```
LDB> run to UPDATE
LDB> profile on
LDB> run to $999 # last statement in UPDATE
LDB> profile off
LDB> profile report
```

Thus we've managed to profile only the routine UPDATE and its children, with timing statistics for the children accrued only when invoked within the UPDATE call-tree.

In a real database system the time to do a single UPDATE may vary widely depending on scope and system load -- timing a single instance may be misleading. We can profile all invocations of UPDATE together by doing the following:

```
LDB> bkp UPDATE
LDB> link UPDATE to "profile on; run"
LDB> bkp $$$$UPDATE
LDB> link $$$$UPDATE to "profile off; run"
LDB> run
```

LDB's *link* command is used to associate a string of LDB commands with a given breakpoint, which is automatically executed whenever the breakpoint is hit. The above sequence will turn profiling on (off) at each entry to (exit from) UPDATE, and then continue the run. At the completion of the run (LDB stops at *\$exit*, just before termination) we'll issue a *profile report* to view the accumulated data.

PROF and PROFVIEW

Both PROF and PROFVIEW are CRI developed tools for presenting the results of a profiling run. We won't document them here except to give a couple of simple examples.

LDB's *profile report* command is equivalent to the following sequence:

```
LDB> profile dump
LDB> sh "prof -x (your_executable) > rawfile"
LDB> sh "profview rawfile"
LDB> sh "rm rawfile"
```

The *profile dump* command creates a profile file containing the sampling data collected so far. LDB's *sh* command runs quoted UNIX commands under a shell; here we first use the PROF utility to post-process the results, and then PROFVIEW to view them. If you're in an X-Window System environment, PROFVIEW will create its own window, otherwise a line-mode menu interface is employed.

PROF can also be used to format a report directly. After creating the profile data file and exiting LDB, the following will generate a summary of activity for those areas of the code that consumed more than 1% of the total profiled execution:

```
prof -st -H 1 (your_executable) > report
```

The scope and impact of profiling can be controlled via debugger variables. The effect of these variables will be clearer if we understand a little about how profiling is implemented on UNICOS.

A process that wishes to be profiled does so by first logically partitioning its instruction segment into a set of "buckets", where the number of buckets depends both on the size of the program instruction segment to be monitored, and the size of each bucket. By default the entire code segment is monitored, and the bucket size is 4 words. This may be altered under LDB by setting \$PROF_SADDR (start address), \$PROF_EADDR (end address), and/or \$PROF_WPB (words per bucket).

The profiling process then allocates memory from the heap to hold counters for the bucket set, passing a pointer to this area, along with a sampling interval, to the UNICOS kernel via a system call. Profiling is now enabled. The default sampling interval of 512 microseconds can be altered by setting the LDB variable \$PROF_RATE (a poor choice of terms because this is really a sampling interval, not a rate - we are following Cray conventions here).

For any process that has requested profiling, the UNICOS kernel periodically interrupts that process to examine its program counter (pc). The kernel then maps the pc to a logical bucket, and increments the corresponding bucket counter in the target process. At the end of execution, the profiled process dumps its bucket counters to the file named by the LDB variable \$PROF_DATA, (default file name is *prof.data*).

The LDB variables mentioned above, minus the leading dollar sign, correspond to the environment variables that control CRI's profiling library, LIBPROF.

As always, there are trade offs to be considered when changing the defaults. Setting a smaller bucket size to achieve finer granularity will require a larger heap for the profiled process. The heap requirements can be reduced by setting start and end addresses to monitor only a fraction of the application's code. Because profiling is an expensive procedure requiring repeated operating system intervention, the variable \$PROF_RATE should be set to a high number (lowering the frequency!) when profiling any long-running job. The only negative effect of setting this parameter large is to degrade the statistical quality of the report, an effect mitigated by the size of the job.

Implementation

The implementation of statistical profiling in LDB is done in three phases: initialization, enabling (disabling), and reporting.

Initialization

During the code instrumentation phase processing of the first *run* command following the first *profile on* command, LDB initializes profiling in the traced process. Initialization prepares the traced process to collect statistical profiling data during subsequent runs. Memory is allocated in the heap of the traced process to hold the profiling data. This is done by setting up a call to *malloc* in the traced process while the process is stopped (standard CRI argument list conventions), setting the traced process program counter to the entry point for *malloc*, and setting register B00 to the code location of a zero instruction. The traced process is then released to run. The process runs through *malloc* allocating a block of memory on the heap as specified in the input arguments. The process returns to the code location stored in register B00 on entry, which

was modified to contain the address of an instruction containing a zero instruction, which stops the traced process and returns control to LDB. LDB then saves the contents of register S1 (Cray function return convention) in the traced process which contains the address of the *malloced* block. This address is later used as an argument to the *profile* system call during the enabling phase. Profiling is initialized once. Debugger variables that control bucket size and code range must be set prior to initialization as they affect the size of the heap allocated in the traced process.

Enabling (disabling)

Profiling is enabled during the code instrumentation phase of *run* command processing following profile initialization. Profiling may be enabled and disabled many time during a debugging session, but initialization occurs once. Enabling profiling involves patching in code in the traced process to execute the *profile* system call. While the process is stopped, LDB sets up an argument list following standard Cray system call conventions, and patches in the following instructions:

```
SM16      0
SM14      0
RX
SM16      1
ERR
```

The *SM* instructions have to do with Cray multitasking. The *EX* instruction is an exchange to the UNICOS kernel to execute the *profile* system call. LDB set up the system call argument list specifying sampling rate, location of profile data buffer (returned from *malloc*), and start and end of code area being profiled. These arguments are controlled by the user via debugger variables and must be set prior to enabling profiling. Once the above code is patched in, LDB sets the program counter of the traced process to the patch code and releases the process to run. The process stops at the zero instruction (*ERR*) and control returns to LDB. Statistical profiling is now enabled. Profiling is disabled by the same mechanism but with a very large sampling rate(interval).

Reporting

Profiling statistical data is reported to the user by LDB upon entering the *profile report* command. This data is cumulative and represents runs after initialization when profiling was enabled (on). Profile data may be cleared (zeroed) by the user via the *profile clear* command. Reporting is implemented in LDB by generating a profile data file suitable for input to the PROF utility. LDB reads data from the profile data buffer in the heap of the traced process as input to the generation of a PROF input file. LDB then issues a shell command to run the PROF utility on the profile data file created. PROF associates profile data with symbol table information and generates an input file for the PROFVIEW utility. LDB then issues a shell command to execute PROFVIEW on the PROF output. PROFVIEW is an interactive utility that displays profiling data to the user in various formats utilizing a command line or X-window interface. The profile data file is retained after the debugging session is terminated for further analysis by the user. The name of the profile data file is controlled by the user via a debugger variable. Thus, several data files may be generated and saved at different points in the program.

Integrated Performance Analysis Summary

Sampling data for statistical profiling purposes can be collected and controlled via the LDB *profile* command. Sampling characteristics can be altered by setting debugger variables to control granularity, range, and frequency. Profiling reports are generated with the standard UNICOS tools PROF and PROFVIEW.

Integrated Coverage Analysis

Coverage data reports whether or not code was executed at least once. Coverage information can be useful in analyzing the effectiveness of a program test suite, tracking where new tests need to be developed, and determining "dead" code. Typical coverage tools are source level translators that instrument source code with library calls to gather coverage data at block boundaries. These tools are language dependant and require that an additional version of the code be maintained in order to analyze coverage. We are integrating coverage analysis into LDB much like performance analysis, although with a different internal mechanism. Integration into the debugger combines coverage analysis with process control allowing the user to control scope and granularity similar to integrated profiling. Coverage analysis via the debugger can be done against the production version of the code (must be compiled with symbols) and supports a combination of source languages (currently fortran, lrltran, C, and cal). The integration of coverage analysis into LDB is currently in development.

User Interface

The syntax of the LDB *cover* command will mirror the *profile* command. Five options will be supported: *on*, *off*, *clear*, *dump*, and *report*. Coverage analysis will be turned *on* and *off* by the user as required during the debugging session. A coverage analysis data file will be generated via the *dump* and *report* options. Coverage data will be viewed via the *report* option. Debugger variables will control the range of code space in the traced process to be covered and the name of the coverage data file. LDB will invoke a locally written utility to report coverage data to the user.

Implementation

Integrated coverage analysis will be implemented using existing LDB temporary breakpoint technology. During the initialization phase (first *run* command following first *cover on* command), LDB will populate the internal breakpoint table (a linked list) with LDB-generated temporary breakpoints at all lines and labels within the range of code space being covered as specified by the user via debugger variables. The temporary breakpoints are then applied to the traced process as normal. When a coverage temporary breakpoint is hit a routine will be called to update coverage data and the temporary breakpoint will be released. Thus, a code block will be trapped for coverage only once, which is all that is required. The effect on performance should not be great because the traced process is only stopped once at each code block boundary. Coverage will be disabled by setting coverage temporary breakpoints to a disabled state in the breakpoint table. Coverage data will be cleared merely by zeroing the cover data accumulated in LDB. A coverage data file will be generated while processing the *dump* and *report* options, and will be retained after the debugging session terminates.

Integrated Coverage Analysis Summary

Data for analyzing program coverage can be collected and controlled via the LDB *cover* command. Debugger variables control code range and data file name. Coverage reports are generated with a locally written interactive utility to view coverage data.

Other Applications

Other possible applications of code instrumentation technology include incremental compilation, automatic data race detection, and integrated visualization/animation.

Incremental compilation would allow the user to compile new routines at debug time to be patched in to the traced process. Thus, calculations and analysis not anticipated at compile time would be supported and would run at full speed.

Data race detection will become increasingly important as the high performance computing community moves to distributed and MIMD parallel computing environments that introduce non-determinism in program execution. An integrated debugger tool to detect data races on-the-fly would be very useful.

Integrated visualization is a feature of many debuggers now, including LDB. Debuggers allow visual snapshots of data while the process is stopped. Implementations typically pipe user data to a graphics process or call internal hard-wired graphics routines. Piping provides flexibility at the expense of performance (LDB utilizes piping). Internal hard-wired graphics provides better performance (ie can handle more user data) at the expense of flexibility. I suggest for future development debugger functionality that allows visualization (animation) of user data while the code is running via non-intrusive code instrumentation.

Summary

Code instrumentation is a powerful debugger tool capable of integrating diverse functionality under debugger control. This paper describes three applications and suggests others for future consideration.

**The Application of Code Instrumentation Technology
in the Los Alamos Debugger**

**Fast Conditional Software Watchpoints
Integrated Performance and Coverage Analysis**

**Supercomputer Debugging Workshop
October, 1992**

**Jeffrey S. Brown
Los Alamos National Laboratory**

Fast Conditional Software Watchpoints

Code generation

- mini compiler**
- fortran, limited C**
- peep-hole optimizer**
- three stage compiler:**
 - machine independent "stack" code**
 - relocatable pseudo machine code**
 - absolute machine code**

Process Instrumentation

- on-the-fly detection**
- two pass approach**
- watch all mechanism**
- watch in mechanism**

TMC collaboration

- compiled events**

```
r:/usr/tmp/jxyb/ldb% ldb -n test/test77yez.x
```

```
ldb version 1.3
```

```
built: 09/21/92 at 12:29:51
```

```
attached to absolute file: test/test77yez.x
```

```
entering debug mode ...
```

```
processing commands in .ldbinit file ...
```

```
  $srcdir = 00000000000000000000
```

```
  $srcdir = 0721453467200000000000
```

```
TEST> watch all for k .gt. 1234
```

```
TEST> list
```

```
  watchpoint active (watch all for k .gt. 1234)
```

```
  no user-specified breakpoints
```

```
TEST> run
```

```
instrumenting code for watchpointing: done
```

```
watchpoint condition met in subroutine: SUB (detected at exit)
```

```
user process stopped at program counter: 445pb = 48L @ TEST() - 14pa
```

```
TEST> watch in sub
```

```
TEST> list
```

```
  watchpoint active (watch in sub for k .gt. 1234)
```

```
  no user-specified breakpoints
```

```
TEST> rerun
```

```
instrumenting code for watchpointing: done
```

```
watchpoint condition met
```

```
user process stopped at program counter: 475pa = 5L @ SUB()
```

```
SUB> k
```

```
  00000243020b: k = 1235
```

```
SUB> list source
```

```
          subroutine sub(ch,k)
          character*(*) ch
          c      print *, "in the userpcrt routine"
          SUB()  k = k + 1
=>W 5L         return
              end
```

```
SUB>
```

r:/usr/tmp/jxyb/ldb% ldb test/test77yez.x

ldb version 1.3

built: 09/21/92 at 12:29:51

attached to absolute file: /usr/tmp/ld44028.copy (copy of test/test77yez.x)

entering debug mode ...

processing commands in .ldbinit file ...

\$srcdir = 00000000000000000000

\$srcdir = 0721453467200000000000

TEST> watch all for kk .gt. 4321

TEST> list

watchpoint active (watch all for kk .gt. 4321)

no user-specified breakpoints

TEST> run

instrumenting code for watchpointing: done

watchpoint condition met in subroutine: TEST (detected at entry to SUB)

user process stopped at program counter: 471pc = SUB() + 2pc

SUB> watch in test

SUB> list

watchpoint active (watch in test for kk .gt. 4321)

no user-specified breakpoints

SUB> rerun

instrumenting code for watchpointing: done

watchpoint condition met

user process stopped at program counter: 343pc = \$10A @ TEST()

TEST> kk

00000243021b: kk = 4322

TEST> list source

```

                                k = 0
                                kk = 0
                                p = loc(b)
W $5          5          continue
                                kk = kk + 1
=>W $10A      do 10 i=1,maxi
W $8A          do 8 l=maxi,1,-1
                                b(i) = 100-l
                                c(i) = i + b(i) - 1
                                a(i) = b(i) + c(i) + i + 1
W $8B          8          continue
W $10
W $10B        10         continue
                                pa(1) = 23.0
```

TEST>

TEST> cdbx\$cmd+3\70

00052426pa:	031 2 0 0	A2	-1	
00052426pb:	110 0 00 01263400001	00000205316,0	A0	
00052427pa:	110 1 00 01263600001	00000205317,0	A1	
00052427pd:	110 2 00 01264000001	00000205320,0	A2	
00052430pc:	100 1 00 01272200001	A1	00000205351,0	
00052431pb:	030 1 1 0	A1	A1+1	
00052431pc:	111 3 00 000	,A1	A3	
00052432pb:	110 1 00 01272200001	00000205351,0	A1	
00052433pa:	024 1 00	A1	B00	
00052433pb:	025 1 76	B76	A1	
00052433pc:	020 1 00 12455000001	A1	00000252264	
00052434pb:	025 1 00	B00	A1	
00052434pc:	025 2 77	B77	A2	
00052434pd:	051 1 0 1	S1	S1	
00052435pa:	022 0 00	A0	00	
00052435pb:	120 1 00 10604000001	S1	00000243020,0	K @ TEST()
00052436pa:	040 2 00 00464400000	S2	00000002322	
00052436pd:	061 0 2 1	S0	S2-S1	
00052437pa:	016 000252177	JSP	00052437pd	cdbx\$cmd() + 14pd
00052437pc:	031 0 0 0	A0	-1	
00052437pd:	044 4 4 4	S4	S4&S4	
00052440pa:	024 1 76	A1	B76	
00052440pb:	071 0 1 1	S0	+A1	
00052440pc:	017 000252301	JSM	00052460pb	cdbx\$cmd() + 35pb
00052441pa:	024 1 77	A1	B77	
00052441pb:	071 0 1 1	S0	+A1	
00052441pc:	017 000252236	JSM	00052447pc	cdbx\$cmd() + 24pc
00052442pa:	100 1 00 01263600001	A1	00000205317,0	
00052442pd:	100 2 00 01264000001	A2	00000205320,0	
00052443pc:	010 000252227	JAZ	00052445pd	cdbx\$cmd() + 22pd
00052444pa:	100 0 00 01263400001	A0	00000205316,0	
00052444pd:	025 0 77	B77	A0	
00052445pa:	030 0 0 2	A0	A2	
00052445pb:	035 1 77	0,A0	B77,A1	
00052445pc:	000 000	ERR		
00052445pd:	100 0 00 01263400001	A0	00000205316,0	
00052446pc:	025 0 77	B77	A0	
00052446pd:	030 0 0 2	A0	A2	
00052447pa:	035 1 77	J,A0	B77,A1	
00052447pb:	005 0 76	J	B76	
00052447pc:	100 1 00 01263600001	A1	00000205317,0	
00052450pb:	100 2 00 01264000001	A2	00000205320,0	
00052451pa:	010 000252255	JAZ	00052453pb	cdbx\$cmd() + 11pb
00052451pc:	100 0 00 01263400001	A0	00000205316,0	
00052452pb:	025 0 77	B77	A0	

00052452pc:	024 0 66	A0	B66	
00052452pd:	024 2 66	A2	B66	
00052453pa:	000 000	ERR		
00052453pb:	100 0 00 01263400001	A0	00000205316,0	
00052454pa:	025 0 77	B77	A0	
00052454pb:	024 0 66	A0	B66	
00052454pc:	024 2 66	A2	B66	
00052454pd:	005 0 76	J	B76	
00052455pa:	031 1 0 0	A1	-1	
00052455pb:	025 1 76	B76	A1	
00052455pc:	130 0 00 01262600001		00000205313,0 S0	
00052455pb:	130 1 00 01263000001		00000205314,0 S1	
00052457pa:	130 2 00 01263200001		00000205315,0 S2	
00052457pd:	006 000252164	J	00052435pa	cdbx\$end() + 12pa
00052460pb:	120 0 00 01262600001	S0	00000205313,0	
00052461pa:	120 1 00 01263000001	S1	00000205314,0	
00052461pd:	120 2 00 01263200001	S2	00000205315,0	
00052462pc:	100 1 00 01272200001	A1	00000205351,0	
00052463pb:	101 2 00 000	A2	,A1	
00052464pa:	025 2 00	B00	A2	
00052464pb:	031 1 1 0	A1	A1-1	
00052464pc:	110 1 00 01272200001		00000205351,0 A1	
00052465pb:	010 000252330	JAZ	00052466pa	cdbx\$end() + 43pa
00052465pd:	000 000	ERR		
00052466pa:	005 0 00	J	B00	

r:/usr/tmp/jxyb/ldb% ldb 1.3a/bin/cray-ymp/ldb1.3a

ldb version 1.3
built: 09/21/92 at 12:29:51

attached to absolute file: /usr/tmp/ld44287.copy (copy of
1.3a/bin/cray-ymp/ldb1.3a)

entering debug mode ...
processing commands in .ldbinit file ...

\$srcdir = 00000000000000000000
\$srcdir = 0721453467200000000000

main> bkp ldclrbkp
main> run with "-a test/test77yez.x"

ldb version 1.3a
built: 10/02/92 at 17:25:39

attached to absolute file: test/test77yez.x
entering debug mode ...

processing commands in .ldbinit file ...

\$srcdir = 00000000000000000000
\$srcdir = 0721453467200000000000

TEST> SUB\10

	SUB()	=	*
00000467pa:	020 0 00 37724000000	A0	00000177520
00000467pd:	025 0 77	B77	A0
00000470pa:	020 2 00 37722600000	A2	00000177513
00000470pd:	030 0 0 2	A0	A2
00000471pa:	022 1 04	A1	04
00000471pb:	035 1 77	0,A0	B77,A1
00000471pc:	025 2 02	B02	A2
00000471pd:	025 6 01	B01	A6
00000472pa:	024 7 01	A7	B01
00000472pb:	107 1 00 00000400000	A1	00000000002,A7

TEST> watch all for k .gt. 55

TEST> run

instrumenting code for watchpointing: done
user process stopped at program counter: 6221pc = 21L @ ldclrbkp()
ldclrbkp> sh

r:/usr/tmp/jxyb/ldb% inquiry

Machine R Mon Oct 5 20:11:44 1992

User=jxyb uid=[1726]:

PID	SIZE	SECONDS_USED_USER+SYSTEM	TTY:PROCESS_STATUS
[42416]	0.07MW	CPU= 0.4470+ 0.8136s	L7:csH SLEEPING
[43867]	0.07MW	CPU= 0.4128+ 0.6145s	p008:csH SLEEPING&
[44287]	0.36MW	CPU= 0.5353+ 0.6435s	p008:ldb SLEEPING&
[44288]	0.05MW	CPU= 0.0401+ 0.0467s	p008:klaM SLEEPING&
[44336]	0.35MW	CPU= 0.4739+ 0.3608s	p008:ld44287. STOPPED&
[44337]	0.05MW	CPU= 0.0399+ 0.0280s	p008:klaM SLEEPING&
[44354]	0.12MW	CPU= 0.0867+ 0.0124s	p008:test77ye STOPPED&
[44359]	0.03MW	CPU= 0.0040+ 0.0053s	p008:sh SLEEPING&
[44360]	0.06MW	CPU= 0.1410+ 0.2153s	p008:csH SLEEPING&
[44363]	0.32MW	CPU= 0.0022+ 0.0148s	p008:inquiry RUNNING on CPU 1

r:/usr/tmp/jxyb/ldb% ldb -p 44354 test/test77yez.x

ldb version 1.3

built: 09/21/92 at 12:29:51

attached to running process: /proc/44354

entering debug mode ...

processing commands in .ldbinit file ...

\$srcdir = 0000000000000000000000

\$srcdir = 0721453467200000000000

TEST> SUB\10

	SUB()	=	*
00000467pa:	020 0 00 37724000000	A0	00000177520
00000467pd:	020 2 00 37722600000	A2	00000177513
00000470pc:	022 1 04	A1	04
00000470pd:	024 3 00	A3	B00
00000471pa:	007 000252131	R	00052426pb cdbx\$cnd() + 3pb
00000471pc:	025 2 02	B02	A2
00000471pd:	025 6 01	B01	A6
00000472pa:	024 7 01	A7	B01
00000472pb:	107 1 00 00000400000	A1	00000000002,A7
00000473pa:	121 7 00 000	S7	,A1

TEST>

```
r:/usr/tmp/jxyb/ldb% !!
ldb 1.3a/bin/cray-ymp/ldb1.3a
```

```
ldb version 1.3
built: 09/21/92 at 12:29:51
```

```
attached to absolute file: /usr/tmp/ld44538.copy (copy of 1.3a/bin/cray-ymp/ldb1.3a)
entering debug mode ...
```

```
processing commands in .ldbinit file ...
```

```
$srcdir = 0000000000000000000000
```

```
$srcdir = 0721453467200000000000
```

```
main> bkp ldclrbkp
```

```
main> run with "-n test/test77yes.x"
```

```
ldb version 1.3a
built: 10/02/92 at 17:25:39
```

```
attached to absolute file: test/test77yes.x
entering debug mode ...
```

```
processing commands in .ldbinit file ...
```

```
$srcdir = 0000000000000000000000
```

```
$srcdir = 0721453467200000000000
```

```
TEST> $5\5
```

	\$5 @ TEST()	*
00000334pc: 042 3 7 7	S3	1
00000334pd: 060 6 7 3	S6	S7+S3
00000335pa: 040 7 00 00031000000	S7	00000000144
00000335pd: 041 5 00 00030600000	S5	1777777777777777777634
00000336pc: 130 6 00 10604200001	00000243021,0 S6	KK @ TEST()

```
TEST> watch for kk .gt. 54
```

```
TEST> run
```

```
instrumenting code for watchpointing: done
```

```
user process stopped at program counter: 6221pc = 21L @ ldclrbkp()
```

```
ldclrbkp> sh
```

r:/usr/tmp/jxyb/ldb% inquiry

Machine R Mon Oct 5 20:18:25 1992

User=jxyb uid=[1726]:

PID	SIZE	CPU=	SECONDS_USED	USER+SYSTEM	TTY:PROCESS_STATUS
[42416]	0.07MW	CPU=	0.4470+	0.8136s	L7:csH SLEEPING
[43867]	0.07MW	CPU=	0.4207+	0.6410s	p008:csH SLEEPING&
[44503]	0.12MW	CPU=	2.4194+	24.6967s	p008:test77ye IN MEMORY&
[44538]	0.36MW	CPU=	0.5361+	0.5048s	p008:ldb SLEEPING&
[44539]	0.05MW	CPU=	0.0402+	0.0816s	p008:klam SLEEPING&
[44551]	0.36MW	CPU=	0.3866+	0.2740s	p008:ld44538. STOPPED&
[44552]	0.05MW	CPU=	0.0399+	0.0762s	p008:klam SLEEPING&
[44558]	0.12MW	CPU=	0.0941+	0.0132s	p008:test77ye STOPPED&
[44559]	0.03MW	CPU=	0.0040+	0.0074s	p008:sh SLEEPING&
[44560]	0.06MW	CPU=	0.1406+	0.2128s	p008:csH SLEEPING&
[44562]	0.32MW	CPU=	0.0022+	0.0149s	p008:inquiry RUNNING on CPU 1

r:/usr/tmp/jxyb/ldb% ldb -p 44558 test/test77/yez.x

ldb version 1.3

built: 09/21/92 at 12:29:51

attached to running process: /proc/44558

entering debug mode ...

processing commands in .ldbinit file ...

\$srcdir = 00000000000000000000

\$srcdir = 0721453467200000000000

TEST> #5\5

	\$5 @ TEST()	=	*
00000334pc:	007 000252204	R	00052441pa cdbx\$end() + 16pa
00000335pa:	040 7 00 00031000000	S7	00000000144
00000335pd:	041 5 00 00030600000	S5	177777777777777777634
00000336pc:	130 6 00 10604200001	00000243021,0	S6 KK @ TEST()
00000337pb:	130 3 00 10604400001	00000243022,0	S3 KK @ TEST() + 1b

TEST> 00052441pa\5

00052441pa:	042 3 7 7	S3	1
00052441pb:	060 6 7 1	S6	S7+S3
00052441pc:	006 000252130	J	00052426pa cdbx\$end() + 4pa
00052442pa:	024 7 03	A7	NO?
00052442pb:	030 0 7 0	A0	A7+1

TEST> 00052426pa\30

00052426pa:	130 0 00 01262600001	00000205313,0	S0	
00052426pd:	130 1 00 01263000001	00000205314,0	S1	
00052427pc:	130 2 00 01263200001	00000205315,0	S2	
00052430pb:	110 0 00 01263400001	00000205316,0	A0	
00052431pa:	022 0 00	A0	00	
00052431pb:	120 1 00 10604200001	S1	00000243021,0	KK @ TEST()
00052432pa:	040 2 00 00015400000	S2	00000000066	
00052432pd:	061 0 2 1	S0	S2-S1	
00052433pa:	016 000252157	JSP	00052433pd	cdbx\$end() + 10pd
00052433pc:	031 0 0 0	A0	-1	
00052433pd:	044 4 4 4	S4	S4&S4	
00052434pa:	120 0 00 01262600001	S0	00000205313,0	
00052434pd:	120 1 00 01263000001	S1	00000205314,0	
00052435pc:	120 2 00 01263200001	S2	00000205315,0	
00052436pb:	010 000252177	JAZ	00052437pd	cdbx\$end() + 14pd
00052436pd:	100 0 00 01263400001	A0	00000205316,0	
00052437pc:	000 000	ERR		
00052437pd:	100 0 00 0. :63400001	A0	00000205316,0	
00052440pc:	005 0 00	J	B00	
00052440pd:	051 1 0 1	S1	S1	
00052441pa:	042 3 7 7	S3	1	
00052441pb:	060 6 7 3	S6	S7+S3	
00052441pc:	006 000252130	J	00052426pa	cdbx\$end() + 3pa
00052442pa:	024 7 03	A7	B03	
00052442pb:	030 0 7 0	A0	A7+1	
00052442pc:	006 000252130	J	00052426pa	cdbx\$end() + 3pa
00052443pa:	075 3 06	T06	S3	
00052443pb:	055 3 01	S3	S3>77	
00052443pc:	006 000252130	J	00052426pa	cdbx\$end() + 3pa
00052444pa:	020 0 00 01751600001	A0	00000207647	

TEST.

Integrated Performance Analysis

Interested in where cycles are spent

statistical profiling

Profiling against production version of code

no special libraries required at load time

Initialization

run through *malloc* in traced process

Enabling

patch in exchange to *profile* system call

run through patch code

Reporting

generate CRI standard profile data file

shell escape to PROF and PROFVIEW

Integrated Coverage Analysis

Interested in code executed at least once

**useful in determining validity of testing,
which test need to be developed,
weeding out "dead" code**

Coverage analysis against production version of code

**no source code instrumentation
multi-language support**

Initialization

temporary breakpoints at all lines/labels with covered code range

Enabling

apply temporary breakpoints as normal

Reporting

shell escape to locally written interactive utility

Other Applications

Incremental compilation

Data Race Detection

Integrated Animation

CXdb: The Road to Remote Debugging

Larry V. Streepy, Jr.
Rob Gordon, Dave Lingle

Convex Computer Corporation
3000 Waterview Parkway
Richardson, TX 75083
streepy@convex.com

August 27, 1992

Abstract

In today's development environment, where typical systems consist of multi-host networks, there is a need to run the debugger on one host and debug an application running on a different host. This functionality requires the development of several components: abstractions within the debugger to hide the "remote-ness" of the target process, a remote server to perform the actual debug operations, and a protocol for communications between the client debugger and the remote server. This paper covers the design and implementation of these three components in Convex's CXdb debugging system.

1.0 Introduction

This paper describes the implementation of remote debugging capabilities within the CXdb debugging system [StBr91] [BuCh91] [Conv91a] [Conv91b]. The approach used in CXdb is similar to other remote implementations, such as GDB [Sta89], and work done at BBN [Lawr90] [WeMi84].

Most remote debugging systems consist of three components: a local debugger client, a remote debug server, and a communication protocol between the client and server. Figure 1 shows a high level view CXdb's remote debugging environment.

The actual implementation of the CXdb remote debugging system contained several features not typically found in earlier (or traditional) systems. These features greatly decreased the development time required and enhanced the maintainability of the final product. They are:

1. The development and use of the Message Interface Generator (MIG). In order to decrease the development time and maintenance overhead involved in the protocol manipulation routines we developed an automated code generation system, the MIG. The MIG tools generates code that handle the tasks of building packets, sending packets, receiving packets, and extracting data fields from packets.
2. Software abstractions within the local debug client that hide the "remote-ness" of the target process from the majority of the CXdb code. This minimized the changes required to implement remote debugging.
3. A clean separation of tasks between the debug client and remote server. The local debug client is responsible for all symbolic understanding of the target application. The remote server is strictly a machine level debugger.

This paper will cover the following major areas:

- The motivation behind the development of CXdb's remote debugging capability.
- The Remote Debugging Protocol and the MIG tools created to work with it.
- The abstraction model used within CXdb to integrate remote processes.
- The design of the remote debugging server.

1.1 Motivation

More and more frequently, the solution to today's computing problems are no longer being handled by a single

machine. Typical computing configurations include cooperative networks with multiple, often heterogeneous, hosts. Client-server architectures are becoming the norm, and the number of distributed applications is increasing rapidly. With all of this distributed functionality, the environment needed to effectively debug these applications must also become distributed.

Additionally, many specialized compute servers require a general purpose front-end machine to provide access and control. A good example is a typical real-time system (RTS). Often, only a minimal environment is provided on the actual RTS. A more general-purpose host typically front-ends the RTS. Real-time applications are compiled and prepared on the front-end and then down-loaded to the RTS. This environment usually provides little debugging capability on the RTS. With a debugger capable of handling remote processes, then all that need be written for the RTS is the remote debug server, which requires far less application level support than a full symbolic debugger.

For support of kernel debugging, the remote system may not even have an application environment in which to run. The remote debug server can be embedded within the kernel to be debugged. The remote server, as described in this paper, is relatively simple to implement.

2.0 Architecture Overview

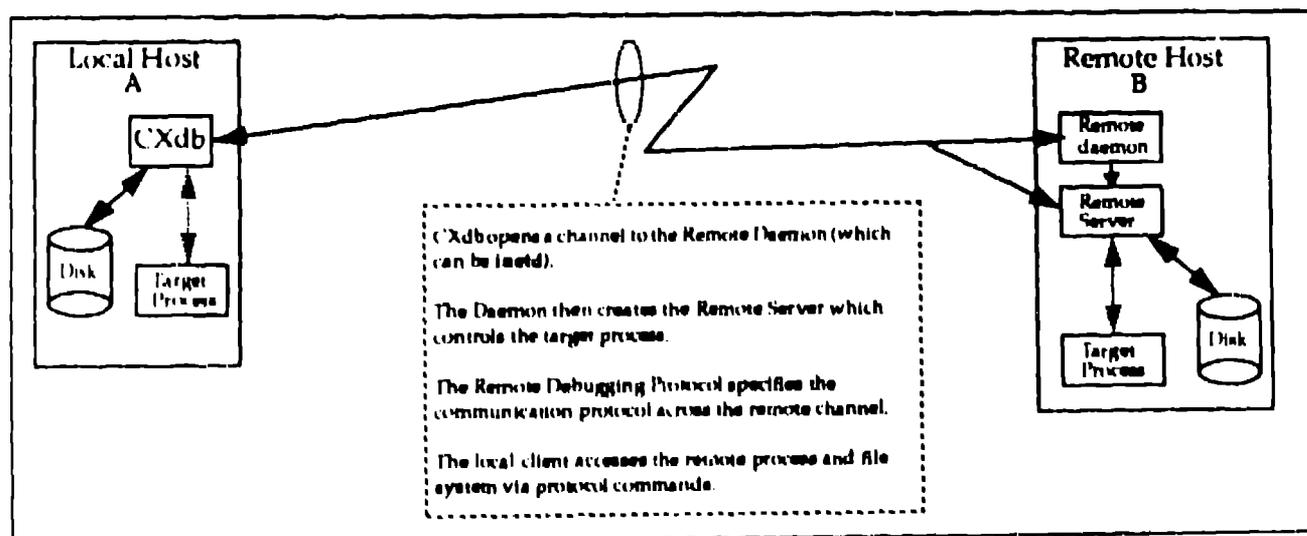
Figure 1 shows the major components within the CXdb remote debugging system. CXdb runs on host A, the "local" host, and communicates with the remote server on host B, the "remote" host. The remote server performs all actual control of the target process being debugged. The connection between the hosts is assumed to be reliable (currently TCP/IP).

The initial design of CXdb's remote debugging capabilities was meant to be sufficient to support remote debugging between two Convex hosts running ConvexOS or a local Convex host running ConvexOS and a remote host running ConvexRTS/rtk (Convex's real-time kernel).

Standard Unix Internet services are used to create the remote server. An entry is made in the */etc/services* file specifying a pre-defined port for connecting to the remote server, and an entry in */etc/inetd.conf* will cause *inetd* to start the remote server when a connection is made to that port.

In cases where *inetd* is not available, as is the case for ConvexRTS/rtk, then a Remote Daemon must be implemented which takes its place. The remote daemon simply listens on a specific port (ConvexRTS/rtk does support TCP connections) and spawns the remote server when a connection is detected.

Figure 1. Remote Debugging Environment



The Remote Protocol used by CXdb to communicate with the remote server does not place any restrictions on the actual connection other than it transmit data reliably and in order. Thus, any transport layer could be used, given sufficient support to make the link reliable. The protocol does assume that the transport layer may place restrictions on maximum packet length. How physical packet length restrictions are handled is discussed in section 3.3, "Multi-Packet Messages".

When the remote server is started, CXdb initiates a configuration dialog with the server to verify that they are compatible. If the local and remote sides are compatible, then a debugging dialog is started. When the local client is finished it terminates the connection to the remote server. See section 3.0, "The Remote Debugging Protocol", for more details.

There is a considerable amount of software machinery that must be put in place to fully process the remote protocol. Much of this code can be generated from a description of the protocol. A series of code generation tools, collectively called the Message Interface Generator (MIG), were developed to provide an automated mechanism for producing this source code. See section 4.0, "The Message Interface Generator", for more details.

The majority of the software components within CXdb have no concept of a remote process. The process interface provides a consistent interface for local and remote processes. An overview of the internal architecture that provides this uniform interface is provided in section 5.0, "The Process Interface."

The remote server is responsible for all of the actual control of the target process. It provides a protocol based interface for operating on the target. The server also provides rudimentary file services to provide CXdb with access to files on the remote host. This mechanism removes any requirement that the remote and local hosts share file systems. See section 6.0, "The Remote Server", for more details.

3.0 The Remote Debugging Protocol

The Remote Debugging Protocol (RDP) is a protocol definition that provides the mechanism for cooperation between the CXdb client and the remote debugging server.

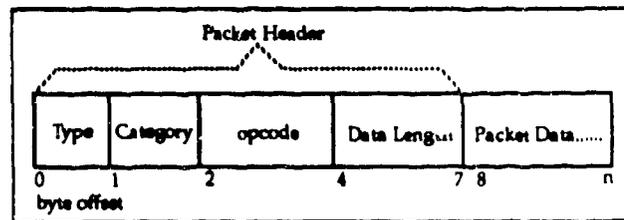
This protocol also provides a level of architecture independence. The protocol defines the operations allowed on a target process, not how they are implemented. The remote server is free to implement the operations in any form required by, or acceptable to, the remote host architecture.

The sections below describe the physical protocol packet layout and the protocol commands currently defined.

3.1 RDP Packet Layout

Each RDP packet consists of a fixed format header and a variable format body. Figure 2 shows the layout of an RDP packet. The individual packet components are described in the sections below.

Figure 2. RDP Packet Layout



3.1.1 Type

There are two types of RDP packets: Commands and Replies. Not all command packets require a reply. In an effort to simplify the development of the protocol handling, we decided to use a strict command-reply model in the protocol. The CXdb client will never send a command until the reply for the previous command has been received¹. With this paradigm in place, reply packets simply contain the command category and opcode instead of a more complicated mechanism for matching a reply to a command (such as sequence numbering).

Not all commands originate on the client side of the connection. There are commands which are sent by the remote server (the P_STATECHANGE command for an example). These commands appear to the CXdb client as *out-of-band* packets since they may be sent asynchronously with respect to client commands. Any out-of-band packets

1. This is true for all cases except the C_ABORT command which is a special command used to abort prior commands.

received while waiting for a reply are processed normally, but do not interrupt the wait for the reply.

3.1.2 Category

The Remote Debugging Protocol is broken into three major categories of operations. All of the operations available within the protocol could have been lumped into a single long list, but we felt that organizing the operations into conceptual categories simplified the task of documenting the protocol as well as making it easier to discuss.

We added the category field to the physical packet layout to increase the modularity of the code that performs the packet dispatch operations. The three defined categories are:

- CONNECT** Messages controlling the remote connection and configuration. This includes session initiation, version arbitration, configuration control (such as debug enabling), and session termination.
- FILE** Messages controlling access to files on the remote host. This includes opening, reading, writing, and closing remote files.
- PROCESS** Messages providing access to and control of the remote process image. This includes creating, attaching, and

detaching a remote process. Access to process memory, registers, attributes, and state.

3.1.3 Opcode

Each category contains a set of specific commands, or opcodes². The specific commands will be detailed in the chapters discussing each major category.

In the sections that follow, specific protocol commands will be referenced using the first letter of the category and the opcode name. For example, the CONFIG command from the CONNECT category will be referenced as C_CONFIG.

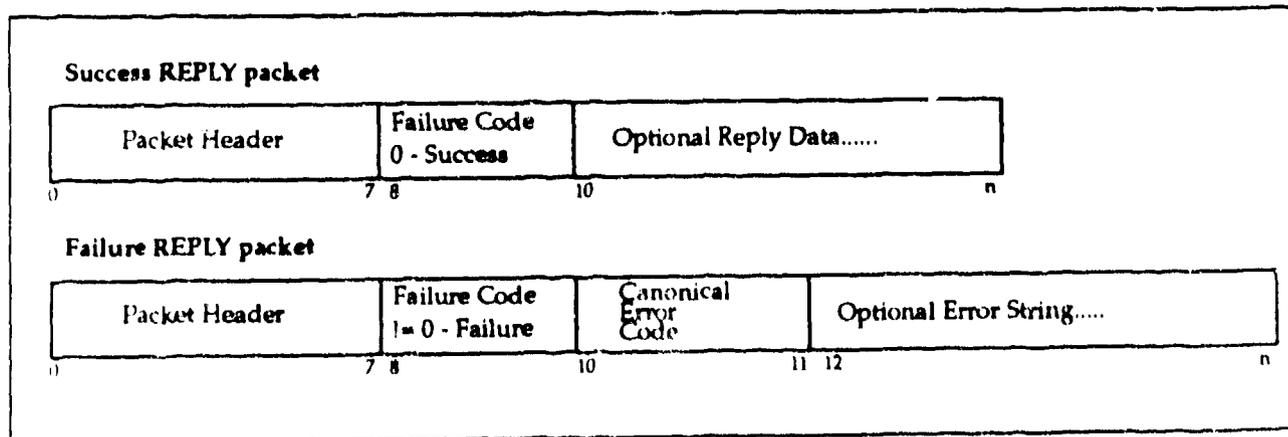
3.1.4 Data Length

The data length field contains the length in bytes of the subsequent command or reply data. It does not include the length of the packet header. A length of zero is a valid length; some packets contain no additional data. For example, the C_TERMINATE command contains no data fields.

3.1.5 Packet Data

The length and format of the remainder of the RDP packet is specific to the individual command or reply. The section on each command will detail the format of the packet data for that command. If a given command requires a reply,

Figure 3. REPLY Packet Formats



² The terms *opcode* and *command* will be used interchangeably within this document.

then the section describing that command will also detail the format of the packet data for the expected reply.

Reply packets have a standardized partial format for the packet data. See the section titled "REPLY Packet Formats" for more details.

3.2 REPLY Packet Formats

There are two forms of a reply packet: success and failure. Reply packets have a header identical to command packets. The first byte of the *packet data* is always used as a success indicator. For a success packet the remainder of the packet data is specific to each reply. For failure packets the format of the packet data is common for all command replies. Figure 3 shows the structure of both reply packets.

3.3 Multi-Packet Messages

Due to potential limits on physical packet size, the packet data for a command or a reply may not fit in a single physical packet. To handle this condition a *multi-packet message* is used to transmit the data. The packet data will be spread across multiple physical packets. A flag word in the packet data indicates when the last packet has been transmitted. It is the responsibility of the recipient to assemble all the data transmitted. If the message being received requires a reply, the reply will not be sent until all the data has been collected and processed.

A given protocol implementation will define a maximum packet size that is appropriate for the physical media being used. The protocol requires that the C_CONFIG packet will fit in a single physical packet. This is required so that the debugger client and the server can arbitrate the maximum packet size.

The following list contains the commands and replies that are currently implemented as multi-packet messages.

- F_READ reply
- F_WRITE command
- P_SETENV command
- P_CREATE command
- P_THDINQ reply
- P_RDREGSET reply
- P_WRREGSET command

- P_READ reply
- P_WRITE command
- P_STDINDATA command
- P_STDOUTDATA command
- P_STDERRDATA command

3.4 Basic Data Types

There are several basic data types that are used in describing the packet data formats for commands and replies. Table 1 shows the standard data types used within RDP packets.

Each field within a protocol packet is assigned a basic type. This type allows the MIG to properly generate code to extract and manipulate the field data in a type consistent fashion. See section 4.2.3.4, "Sender Actions", for a description of how each basic protocol type is mapped onto a specific C or C++ language type.

3.5 Packet Categories

As described above, the protocol commands are grouped into three categories. Each of the categories is briefly described in the following sections. Please refer to Appendix A for a complete listing of the commands within each category.

3.5.1 CONNECT Commands

The CONNECT commands are used to initiate a session with a remote server, arbitrate configurations, out-of-band control, and terminating a remote session. A description of the C_CONFIG and C_TERMINATE packets are given below.

C_CONFIG The C_CONFIG command is the first packet sent to the remote server after the low level connection (i.e. TCP/IP) has been established. This command informs the remote server of the local clients protocol version and architecture. The remote server verifies that it can work properly with the local client and then replies with a message that indicates its architecture, protocol version, max packet size, and an indication of compatibility. It is the remote servers task to decide if the protocol versions are compatible.

Table 1: Basic Data Types

Type	Size	Name	Description
int	1, 2, 4, or 8		Integer data. Number of bytes determines precision. Unless otherwise stated int's are unsigned.
boolean	1		Boolean value: 0 = False, 1 = True
code	1, 2, or 4		Integer (as above) whose value is a coded enumeration. Legal values will be listed for each use of this type.
string	variable	STRLEN STRING	Variable length string encoding. Format is: 2 bytes = Length of string, excluding NULL byte n bytes = NULL terminated string
buffer	variable	NBYTES BYTES	Variable length data buffer. Format is: 8 bytes = Length of data n bytes = raw data bytes

C_TERMINATE

Terminate the connection between the local client and the remote server. The remote server should release any resources it has acquired on behalf of this connection and then exit. If the remote server is controlling an executing process, then process should either be killed (if it was created) or detached (if it was attached). There is no data associated with this command.

The reply to the F_OPEN command will specify the file handle (like a file descriptor) for use in later commands referencing this file.

F_READ

Requests the remote server to read a specified amount of data from a given file. The read begins at the current file position. The current file position is updated by the number of bytes read.

The reply, which is a multi-packet message, contains the data requested.

3.5.2 FILE Commands

The FILE commands are used to access the file system on the remote host. Commands within this category provide open, seek, read, write, and close access to files on the remote host. This allows the local debugger to have access to remote executable and core files without requiring some kind of remote disk mounting. A description of the F_OPEN and F_READ packets are given below.

F_OPEN

Requests the remote server to open a specified file for further processing. The open mode (i.e read, read/write, or write) will also be specified.

3.5.3 PROCESS Commands

The PROCESS commands are used to access and control the target process on the remote system. The general types of operations are:

1. Process creation (start, kill, attach, detach)
2. Access and control of process attributes and limits
3. Process execution control (stop, single step, continue)
4. Access and control of process register sets
5. Access to process memory
6. Setting eventpoints (breakpoints, watchpoints)

A description of the P_SETEXEC, P_CREATE, and P_STATECHANGE packets are given below.

P_SETEXEC Specifies the path name of the executable to manipulate. This must be specified before a subsequent P_CREATE command can be issued. It is the responsibility of the remote server to verify that the file exists, can be accessed, and is a valid executable.

P_CREATE Create a process from the executable specified in a previous P_SETEXEC command. The arguments to supply the process are specified as part of this command. It is the responsibility of the server to perform wildcard expansion and I/O redirection based on the argument list.

P_STATECHANGE

This message is initiated by the remote server. When the server sends this message it indicates that the target process has changed state (i.e. stopped) for some reason. The local client should send P_PROCIHQ and P_THDIHQ commands to determine the new state of the process.

There are four commands which call for a little more discussion: C_ERROR, P_STATECHANGE, P_STDOUT-DATA, and P_STDERRDATA. Typically commands originate with the local debug client. However, with these four commands, they are generated by the remote server. They may be generated asynchronously to any command sent by the local client. As described previously, they are handled as *out-of-band* messages.

4.0 The Message Interface Generator

Much of the source code needed to manage the remote protocol packets is automatically generated based on a protocol definition. The Message Interface Generator (MIG) is an automated code generation system that was designed with the following goals in mind:

1. Decrease the time required to develop the protocol support modules.
2. Increase the maintainability of the protocol support modules.

3. Support development of servers in either C or C++.
4. Support our automated testing facilities already in use on CXdb.

To achieve these goals, the following features were designed into the MIG:

1. Generation of test drivers for use in automated testing.
2. Generation of sending functions that construct and send protocol packets.
3. Generation of receiving functions that break apart protocol packets.

The development and use of the MIG met all of the goals we had initially set forth. See section 4.3, "MIG Usage Experience" for more details.

The generation of these source modules by the MIG is controlled by a protocol definition and a driver specification. The MIG can generate source modules in either C or C++. The format of these control files is presented in the following sections.

4.1 The Protocol Definition

CXdb and the remote server communicate via a well defined protocol as was described previously. The protocol definition file describes this protocol in a machine processable manner. The MIG tools make use of this description for two purposes:

1. Generating an include file which contains manifest constants that describe the protocol. This file is called `proto_descrip.h`.
2. Generating source modules which automatically handle operations on the protocol packets.

The MIG generates the file `proto_descrip.h` to contain a series of #defined constants that specify the protocol organization, packet layout, and field structures. The reasons for generating this file are:

1. This allows the code generated by the MIG to reference these symbolic names and, thus, be more human readable. This will make the ramp-up time for

a new developer trying to understand the protocol operation much shorter.

2. It provides a basis for writing protocol manipulation routines by hand, if necessary.

Within each packet type, multiple packet categories can be defined, and within each category, any number of packets can be defined. For ease of implementation, the MIG tools currently use the list of categories and packets defined within the first type to be the canonical list. Subsequent type definitions may not define new categories or packets within categories.

4.1.1 Major Protocol Sections

The protocol definition is composed of several major sections:

name	The name of the protocol
version	The version number of the protocol definition
packet size	The maximum size of a protocol packet
code sets	Definition of mnemonic codes that will be used to defined packet field values
packet types	Definition of the actual packets that comprise each packet type

The protocol definition is composed of a series of keywords and parameters. Keywords are case insensitive. Complex keyword entries will be described in individual sections below. The high level structure of the protocol definition file is shown in figure 4. Keywords are shown in bold face.

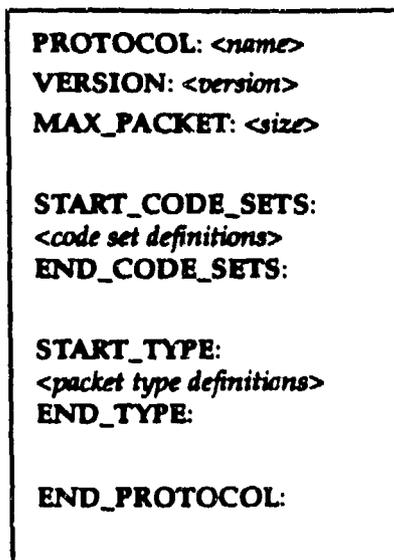
The simple definition entries, **name**, **version**, and **packet size**, are described below. The complex entries, **code set definitions** and **packet type definitions**, are covered in subsequent sections.

Protocol Name

The MIG tools generate a series of #define's that describe the packet layouts. The protocol name is used as a prefix for all these names. This name should be

3. The need to hand-code some protocol manipulation routines was planned for, but never actually needed in the CXdb system.

Figure 4. High Level Protocol Definition Structure



short and meaningful. For example, the protocol name used for CXdb's remote debugging protocol is RDP.

Protocol Version

The protocol version number is meant to be used by client/server pairs to verify that they can communicate properly. It is up to the applications to make use of this value. The MIG tools do nothing more than provide a #define name by which to reference it. The #define name is <name>_PROTOCOL_VERSION. For example, the #define name for the RDP protocol is RDP_PROTOCOL_VERSION.

Maximum Packet Size

This specifies the maximum packet size that the MIG tools will generate. When specifying the RDP protocol, we defined the concept of multi-packets. These packets contain a single logical message, but due to physical constraints (media, transport, etc.) are broken into multiple physical packets.

4.1.2 Code Set Definitions

Many of the data fields within the packets supported by the RDP can only have a specific set of values that are best

defined via a set of mnemonic names (like an enumeration in C). A specific set of mnemonic values is called a Code Set.

When the MIG tools process a code set, a corresponding group of #defined constants are created using the protocol name, the code set name, and the specific mnemonic name. An example of a code set is the open mode specifier in the F_OPEN packet. It can have one of three specific values: RONLY, WRONLY, and RDWR.

The format of a code set definition is shown below.

```
start_codes: <name> <comment>
    mnemonic 1
    mnemonic 2
    ...
end_codes:
```

The code set definition for the open mode example above is shown below.

```
start_codes: OMODE File open modes
    RONLY
    WRONLY
    RDWR
end_codes:
```

The corresponding generated output in the proto_descrip.h file is shown below.

```
/* File open modes */
#define RDP_OMODE_RDONLY 1
#define RDP_OMODE_WRONLY 2
#define RDP_OMODE_RDWR 3
```

4.1.3 Packet Type Definitions

The current RDP defines two major types of protocol packets: command and reply. In order to provide possible extensions in the future, the protocol definition language allows for an arbitrary number of packet types.

Recall that the RDP consists of two major types, command and reply, each broken into three categories, CONNECT, FILE, and PROCESS, which are in turn broken into a series of commands. The type definition syntax parallels this hierarchy; opcode definitions are nested within category definitions which are nested within type definitions.

The format of a packet type definition is shown below, followed by an example.

```
start_type: <name> <abbrev> <prefix>
start_category:
start_opcode:
end_opcode:
...
end_category:
...
end_type:
```

Example:

```
start_type: COMMAND CMD CFLD
...
end_type:
```

The parameters to the start_type entry are defined below.

- | | |
|---------------------|---|
| name | The name parameter is used to create comments within the proto_descrip.h file as well as a series of #define constants, one per packet type. From the example above, the constant for type COMMAND would be RDP_PACKET_COMMAND. |
| abbrev | As stated above, the packets defined within the first type entry form the canonical list. The abbrev parameter is used in the construction of a constant name for each packet. An example, using the FILE OPEN packet, is RDP_CMD_F_OPEN. The 'F' comes from the packet category and is described below. |
| field prefix | Defined constants are generated that describe all of the fields in the packets defined. Since the field layout of corresponding packets in different types will differ, the field prefix parameter is used in the #define name to differentiate fields that might have common names. An example is RDP_CFLD_F_OPEN_MODE_TYPE. |

The category and field definitions are described in the following sections.

4.1.4 Packet Category Definitions

Within each packet type multiple categories can be defined. The current RDP defines three categories: CONNECT, FILE, and PROCESS. The category entry is used

to define these categories. The syntax of the category entry is shown below, followed by an example.

```
start_category: <name> <prefix>
start_opcode:
end_opcode:
...
end_category:
```

Example:

```
start_category: FILE F
...
end_category:
```

The parameters of the `start_category` entry are defined below.

- name** The `name` parameter is used to create comments within the `proto_descrip.h` file as well as constants, one per packet category. From the example above, the constant for category `FILE` would be `RDP_CATEGORY_FILE`.
- prefix** The `prefix` parameter is used in the construction of constant names for each packet defined and all of the fields defined within packets. For example, `RDP_CMD_F_OPEN` and `RDP_CFLD_F_OPEN_MODE_TYPE`.

4.1.5 Packet Field Definitions

Within each category, the packets that fall within that category are defined. The definition of each packet, also called an opcode, contains the name and type of each field in the packet. The syntax of the field definition entry is shown below, followed by an example.

```
start_opcode: <name>
field: <name> <type> [<code set>]
...
[reset_offset:]
[include:
  <include text>
end_include:]
end_opcode:
```

Example:

```
start_opcode: OPEN
field: MODE _4byte OMODE
field: FILE string
end_opcode:
```

The `name` argument is used in generating named constants that identify the packet and constants that describe the fields within the packet. Using the example above, `RDP_CMD_F_OPEN` and `RDP_CFLD_F_OPEN_MODE_TYPE`.

There are three entry types within an opcode definition: `field`, `reset_offset`, and `include`. Each of these entries is described below.

4.1.6 FIELD Entry

Each `field` entry defines a single field within the current packet. The syntax of the `field` entry is shown below, followed by an example.

```
field: <name> <type> [<code set>]
```

Example:

```
field: MODE _4byte OMODE
```

The parameters of the `field` entry are defined below.

4.1.6.1 name

The name of the field. This will be used in generating the `#define` constants which describe the field. Each field is described by one or three constants depending on the field type. See the `type` parameter below for more details.

4.1.6.2 type

The type of the field. As described earlier, each field within a packet is assigned a basic type. The MIG uses this type information to generate code which can properly handle the field value. There are 10 supported field types:

<code>_1byte</code>	1 byte integral field, <code>_1byte_s</code> for signed
<code>_2byte</code>	2 byte integral field, <code>_2byte_s</code> for signed
<code>_4byte</code>	4 byte integral field, <code>_4byte_s</code> for signed
<code>_8byte</code>	8 byte integral field, <code>_8byte_s</code> for signed
<code>string</code>	4 byte integer length followed by NULL-terminated character string
<code>buffer</code>	8 byte length followed by raw binary data

The type of the field is used to determine its size and the type of variables that will be used to work with its value within the generated code. The MIG tools generate a series of #define constants to describe the field contents. If the type is one of the integral values, then three constants are generated: offset, length, and type. If the type is string or buffer, then only an offset constant is created since the size is variable.

An example of the constants created, taken from the MODE field of the OPEN packet, is shown below.

```
#define RDP_CFLD_F_OPEN_MODE_OFF 0
#define RDP_CFLD_F_OPEN_MODE_LEN 4
#define RDP_CFLD_F_OPEN_MODE_TYPE _4byte

/* Use RDP_OMODE_... codes */
```

The comment is automatically added by the MIG tools to make the include file more readable. Any time a field references a code set the name of the code set is added as a comment in the generated proto_descrip.h file.

4.1.6.3 code set

This optional entry is used to indicate which code set, if any, the fields value will be taken from. This information is only used by the MIG for generating test drivers that need to construct packets with valid data within them. The name specified must be a previously defined code set.

4.1.7 RESET_OFFSET Entry

As each field entry is processed its offset from the beginning of the packet is maintained by the MIG tools. The reset_offset entry resets the current offset to 0. This feature can be used for creating variant records. An example, used for handling architecture variations of the attach packet, is shown below.

```
start_opcode: ATTACH
include:
/* ConvexOS specific */
end_include:
field: PID _4byte

reset_offset:

include:
/* ConvexRTS/rtk specific */
end_include:
field: APPNAME string
end_opcode:
```

4.1.8 INCLUDE Entry

The include entry is used to add output to the generated proto_descrip.h file. The specified text is placed in the output file in sequence with the fields defined for this packet. An example, used for adding comments to the THDINQ reply packet, is shown below.

```
start_opcode: THDINQ
field: TOTLEN _8byte
field: LAST _4byte
field: NTHDS _4byte
include:
/* Remaining fields repeat per
thread. Offsets are relative to
thread entry. */
end_include:
reset_offset:
field: TID _4byte
field: STATE _4byte TSTATE
field: SIGNAL _4byte
field: SUBCODE _4byte
include:
/* TNAME field only exists for
ConvexRTS/rtk architecture */
end_include:
field: TNAME string
end_opcode:
```

4.2 Driver Specifications

The MIG tools provide a mechanism for automatically generating code which can manage the packets for the defined protocol. The MIG can produce four different kinds of source modules: generators, dumpers, senders, and receivers.

The four types of drivers are also broken down into two major categories: senders (generators and senders) and receivers (dumpers and receivers). The terminology is a bit awkward, but that's how it evolved.

A generator is a self contained program which generates one of every kind of packet defined within the protocol. A dumper is a self contained program which dumps the contents of every packet that it receives. The generator and dumper modules together form the basis for the automated testing of the remote protocol and MIG tools.

A sender is a module that provides functions that construct and send packets to the remote peer. A receiver is a mod-

ule that receives a packet and breaks it apart for further processing.

The generation of a driver is controlled by a driver specification language. The operation of the MIG tools and the syntax options of the language depends on the type of driver being created. The syntax of the driver specification language is described in the sections below.

4.2.1 Major Sections

The driver specification language is broken into two major sections: *setup* and *packet handling*. The setup section indicates which type of driver is being created and other options about the generation process. The packet handling section determines which packets will be handled by the driver and exactly how those selected packets will be handled. Each of these sections is described below.

4.2.2 Setup Section

The setup section tells the MIG specific information that will control the overall process of creating the driver module. It consists of the following entries (some of which are optional as indicated).

create This must be the first non comment entry in the driver specification. The MIG tools will skip all entries in the specification until it finds the initial create entry. This entry selects the type of driver being created.

name .pattern This entry defines the pattern which is used when constructing generated function names. Special characters allow for the inclusion of the category prefix and opcode name to be included in the generated name. The default pattern is `<driver_type>_<category>_<opcode>`. (optional)

no fail .reply Indicates that no failure reply function should be generated. This is necessary when multiple driver specifications are being used to construct one driver. (optional)

start .include The following text is copied to the generated source file. (optional)

4.2.3 Packet Handling Section

Following the setup section, the packet handling section describes how each packet is to be handled. The format of this section closely resembles the packet type definition section of the Protocol Definition language. The sections below describe how to select specific types, categories, and packets, and how to specify the handling for a specific packet.

4.2.3.1 Type and Category Section

A driver specification must select the packets it will handle. The syntax for selecting packet types and categories is similar to the syntax used in the Protocol Definition language. For example, a driver specification that selects packets within the COMMAND type and all the categories is presented below.

```
start_type: COMMAND
start_category: CONNECT
...
end_category:

start_category: FILE
...
end_category:

start_category: PROCESS
...
end_category:
end_type:
```

Any packet types or categories within a type that are not selected will not be processed by the generated driver.

4.2.3.2 Packet Selection

Within each category, the packets to be handled must be selected. Packets are selected using a syntax similar to the Protocol Definition language. A `start_opcode` entry is used to select a packet, and an `end_opcode` entry is used to terminate the specification for a selected packet.

One extension, compared to the Protocol Definition language, has been added to simplify the maintenance of standard drivers. The `start_opcode` entry will accept an opcode of " all ". This indicates that all packets not yet selected within the current category should be selected and have the default action applied to them.

Driver types generator, sender, and dumper have default actions built into the MIG. The default actions are summarized below.

- generator** Construct a function that generates a packet with valid data for each field in the packet and sends it to the remote client.
- dumper** Construct a function that dumps the contents of the packet in a readable format.
- sender** Construct a function that constructs a packet from the arguments supplied to the function and sends it to the remote client. The `name_pattern` entry, or the default name pattern, will be used to construct the name of the function that handles each packet.

Currently, no default action is defined for a driver of type receiver; a specific action must be specified for every packet to be processed.

An example of a section from a sender type driver specification that uses all default processing is shown below.

```

start_type: REPLY
  start_category: CONNECT
  start_opcode: _all_
  end_opcode:
  end_category:

  start_category: FILE
  start_opcode: _all_
  end_opcode:
  end_category:

  start_category: PROCESS
  start_opcode: _all_
  end_opcode:
  end_category:
end_type:
    
```

4.2.3.3 Action Specifications

Specific actions can be specified for any selected packet. The actions available depend on the category of driver, sender or receiver, being generated. If no actions are specified for a selected packet, then the default processing, if any, is applied to the packet. For example, the specifica-

tion fragment shown below selects the indicated packet and applies the default processing to it.

```

start_opcode: STATECHANGE
end_opcode:
    
```

The actions available for sender and receiver drivers are described in the sections below.

4.2.3.4 Sender Actions

For all packets selected in the driver specification, the MIG will generate a function to handle each packet. The exact functioning of that function, how it is called, how it processes each field, etc., are determined by the action specifications made for that packet.

Unless overridden by an action specification, each sender function produced will have one argument for each field defined for the packet. The type of the argument will be determined from the type of the field. The field types string and buffer produce two arguments: one for the length and one for the pointer to the data. The table below summarizes the field type to language type translation.

Table 2: Field Type to Language Type Translation

Field Type	Language Type
_1byte	unsigned char
_1byte_s	char
_2byte	unsigned short
_2byte_s	short
_4byte	unsigned int
_4byte_s	int
_8byte	unsigned long long
_8byte_s	long long
buffer	unsigned int const void *
string	unsigned long long const void *

The action specifications are described below

args Overrides the default argument list for the generated function. The syntax is:

```
args(arg decl[, arg decl, ...])
```

calledas Overrides the generated name of the function. This is generally used when, for some reason, you want a function name that doesn't follow the defined name pattern. The '%' code available in the name_pattern entry described above may be used in the specified name. The syntax is:

```
calledas <name>
```

generate This is the default action for a driver of type generator. It is an error to use this specification on any other type of driver. This action causes the function generated to construct the packet from a set of valid values based on each selected field's type. The values placed in the packet can be overridden with the **select** and **special** actions described below.

select This action is used to select specific fields for processing. This action may be specified multiple times to select multiple fields. If any **select** action is specified, then only those fields listed in **select** and **special** actions will be processed. There are two different syntaxes supported: one for senders and one for generators. The sender syntax is:

```
select(<field name>)
```

The generator syntax is:

```
select(<field name>,  
      <value1>[, <value2>])
```

For scalar fields the **value1** specification is used as the value to be placed in the field. If the field selected is of type string or buffer, then **value1** specifies the length and **value2** specifies the data for the field.

The text specified in the values is directly copied into the generated source code so #define constants or expressions may be

used. An example of two scalar fields and a string field in a generator specification are shown below.

```
select(VERSION, 10)  
select(HWARCH, RDP_HWARCH_C3)  
select(USER, 5, "streepy")
```

Note that only the fields selected will be placed in the argument list for the generated function.

special This action is identical in syntax to the **select** action described above. However, use of this action does not exclude fields that are not specifically named. For example, if a packet has 10 fields and one is listed in a **special** action, then that field will have the special values associated with it and the other nine fields will be given default values.

4.2.3.5 Receiver Actions

For all selected packets in a receiver specification, the MIG will generate a function to handle that packet. The MIG will also generate a function which will break apart the header of the packet and dispatch the packet to the proper generated handling function.

By default, the generated packet handling function will extract all of the fields within the packet into variables of a reasonable type. See Table 2 above for the type correlations. For dump drivers, the contents of each field will then be output. For receiver drivers, there is no default action; one of **dump**, **call**, or **rawcall** must be specified. The action specifications are described below.

call This action will generate a call to the specified function. Only those fields that have been selected (by default, or via **select** or **special** actions) will be included as arguments to the call. The syntax is:

```
call <function name>
```

dump Dump the contents of the selected fields to the message function (as specified in the **msg** parameter to the **create** entry). The name of the field and its value will be output.

handle_fail Specifies that code should be generated to check the failure code in the packet and the name of the function to call if a failure is detected. This is only valid on REPLY type packets. The function named in the action will be passed to the `__dispatchReplyFail` function (which is generated by the MIG unless the `no_fail_reply` entry was used in the setup section). The syntax is:

```
handle_fail <function>
```

rawcall This action is identical to call except that the length of, and a pointer to the raw packet data are added to the end of the argument list of the function called. The syntax is:

```
rawcall <function>
```

The length argument is of type `unsigned long long` and the data pointer is of type `char *`.

select This action is used to select specific fields for processing. This action may be specified multiple times to select multiple fields. If any `select` action is specified, then only those fields listed in `select` and `special` actions will be processed. There are two different syntaxes supported:

```
select (<field name>)

select (<field name>,
      <varname>, <type>,
      [<varname2>, <type2>])
```

For scalar fields, only a single variable name and type are allowed. For string and buffer fields, the first variable name and type are for data length and the second are for the data pointer. The specification provides a mechanism for overriding the default variable name and type chosen for the variables used to extract the fields from the packet. This can be needed to provide type consistency in the function call specified in the `call` action described above. An example of the two formats is shown below.

```
select (VERSION)
select (CLTYPE, cltype, int)
select (USER, ulen, int,
       uname, char *)
```

special This action is identical in syntax to the `select` action described above. However, use of this action does not exclude fields that are not specifically named. For example, if a packet has 10 fields and one is listed in a `special` action, then that field will have the special values associated with it and the other nine fields will be given default values.

4.3 MIG Usage Experience

The development of the MIG tools proved to be a very large win for the development of CXdb's remote capabilities. Table 3 provides data comparing the relative amounts of code generated versus configuration table size.

Table 3: MIG Source Code Summary

Source	Lines
MIG source code - Perl (22% comments)	3320
Protocol Definition	663
Driver specifications	991
Total	4974
Generated source code	21341

As indicated by the table, roughly 5000 lines of source and tables generated 21000 lines of source code. Aside from the 4X direct benefit, the overall maintainability of the system has been greatly enhanced. The maintenance of protocol manipulation routines by hand is tedious and error prone. With the use of the MIG, modifications to the protocol require a change to the protocol definition and regenerating the derived sources. The chance for error in the maintenance is virtually nil, assuming a correct specification (obviously, the client and server will have to be modified to handle any modified protocol values, but the maintenance of the protocol handlers has been removed). For a complete example of the protocol definition and driver specifications, see Appendix B.

5.0 The Remote Server

The remote server provides all of the actual debugging control of the target process. CXdb controls the activity of the server with the Remote Debugging Protocol. The server has absolutely no high-level understanding of the process being debugged; it is restricted to machine-level debugging.

The rationale for excluding source-level information from the server is two-fold: one, reduce the complexity of the remote server; and two, cleanly separate the tasks performed by the server and CXdb. It is important to keep the server small and easy to port/implement. This leads to quicker development on new architectures. The second reason needs a little more explaining.

Unlike DBX and other STAB⁴-based debuggers, CXdb does not maintain its symbolic information within the executable. As described in [Stre91], CXdb uses a set of auxiliary data files to maintain the debugging information. Access to these files by both CXdb and the remote server would pose a serious limitation on the debugging environment. The user would either have to maintain duplicate files on each machine, or provide some kind of remote disk mounting. Either approach can be problematic to the user. To mitigate these problems, only CXdb needs access to the data files; the server only needs access to the executable. Any information that CXdb needs from the executable is retrieved using the FILE commands of the RDP.

4. STAB (Symbol TABLE) information includes name, type, and location for variables and address ranges for source statements

Figure 5. Remote Server Architectural Overview

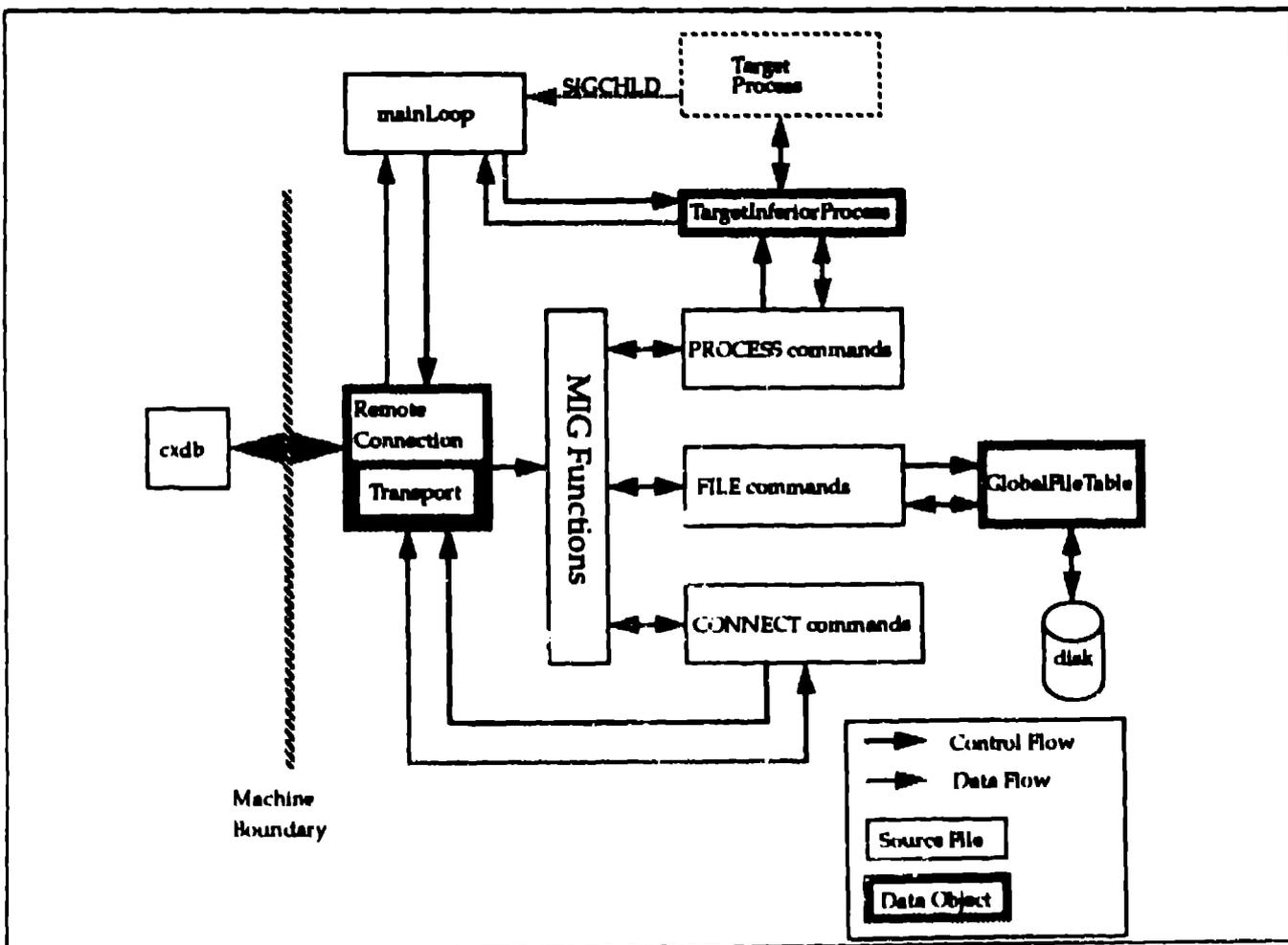


Figure 5 presents a graphical overview of the organization of the major server components. Each of the major data structures and modules are further described in the following sections.

Several major data objects are shown in the diagram, specifically: RemoteConnection, TargetInferiorProcess, and GlobalFileTable. These objects are described in section 5.1, "Major Data Objects".

The server is essentially a protocol engine. It is always waiting for either a state change within the target process or a command to be received from the remote client. This "waiting" is performed by the I/O Manager Module (IOMM). See section 5.2, "IOMM" for more details.

The server incorporates a rudimentary file server in order to implement the FILE commands required by the RDP. The operations performed by this module are described in section 5.3, "File I/O".

All of the actual control of the target process is contained within the Process Interface (PI). The ConvexOS remote sever's PI was derived directly from its equivalent in CXdb (with all symbolic understanding removed). See section 5.4, "Server PI" for more details.

5.1 Major Data Objects

As shown in Figure 5, there are several major data objects that are central to the servers operation. Each of these data objects are briefly described here.

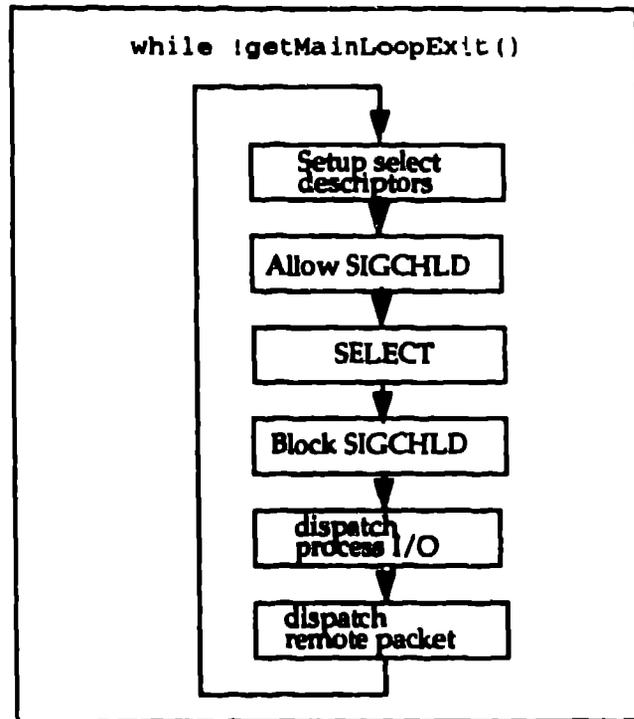
RemoteConnection

This object encompasses all aspects of the network connection to the remote client. It provides operations to initialize the connection, monitor the connection for traffic (used by `select()`), and read and dispatch packets from the remote client (using the MIC generated functions).

TargetInferiorProcess

This is the object which embodies the Process Interface. All the operations described in the "PI" section are provided by this object.

Figure 6. IOMM Processing Algorithm



GlobalFileTable

The server maintains a single table of open files for processing FILE category RDP commands. The operations described in the "File I/O" section are provided by this object.

5.2 IOMM

The IOMM module is the central event dispatch system for all of the remote server. Figure 6 shows the algorithmic control of the server's main loop. The individual steps are outlined below.

1. Set up all the descriptors that need to be monitored for activity. This includes the connection to the remote client and the stdout and stderr of the target process. Any activity on these descriptors will cause the select call to exit.
2. Allow SIGCHLD signals to be delivered. The SIGCHLD signal is normally blocked to prevent problems with reentrancy of the PI modules (especially in malloc). SIGCHLD is only allowed in this narrow region.

3. Perform a select call. The server will wait here until some activity on the file descriptors, or a SIGCHLD (indicating a target process state change) occurs.
4. Block SIGCHLD.
5. Dispatch any target process I/O that was detected on it's stdout and stderr. This data is sent back to the remote client to be displayed to the user.
6. Read and dispatch any remote packet. This will call MIG generated functions to do the actual packet parsing and dispatch.

5.3 File I/O

All file I/O performed on behalf of the remote client (due to FILE category commands) is managed by the `GlobalFileTable`. This data object maintains a set of `OpenFile` objects to handle each file. An `OpenFile` object provides methods for the following operations: open, close, seek, read, and write. The `GlobalFileTable` provides methods for allocating and deallocating `OpenFile` entries.

5.4 Server PI

The Process Interface (PI) within the remote server is modelled after the PI within CXdb. CXdb's PI is described in section 6.0, "The Process Interface". This section will only discuss the differences between the implementations.

Several PI components were removed as part of constructing the server, they are:

1. Support for core files. This is handled directly by CXdb using the file I/O operations supported by the server.
2. Support for remote images. Although possible, it didn't seem reasonable to support multi-level remote access.
3. The breakpoint table. All breakpoints are managed by CXdb using remote read/write of the process address space. This implementation may be modified in the future if the overhead of managing the breakpoints remotely becomes unacceptable.
4. All references to symbolic debugging information. No debugging information beyond the machine state is maintained by the server.

6.0 The Process Interface

The Process Interface (PI) within CXdb is the point of access and control for all process related information. This includes:

1. Maintaining the process' environment (environment data, initial working directory, command line arguments, etc.).
2. Creation of the process or attaching to an existing process.
3. Managing the breakpoints placed by the user and created by CXdb to control process execution.
4. Access to each thread's machine state: scalar, vector, and communication registers.
5. Handling of all signals received by the target process.
6. Access to each thread's stack and memory.

In order to hide the details of accessing all of this process information, several major abstractions were designed into the PI. Figure 7 shows the high-level design of the PI.

Brief descriptions of the important elements of the diagram are given below.

InferiorProcess This maintains the state of the target process across instantiations of a process image. For example, the information that is used to initially create the target process is stored here.

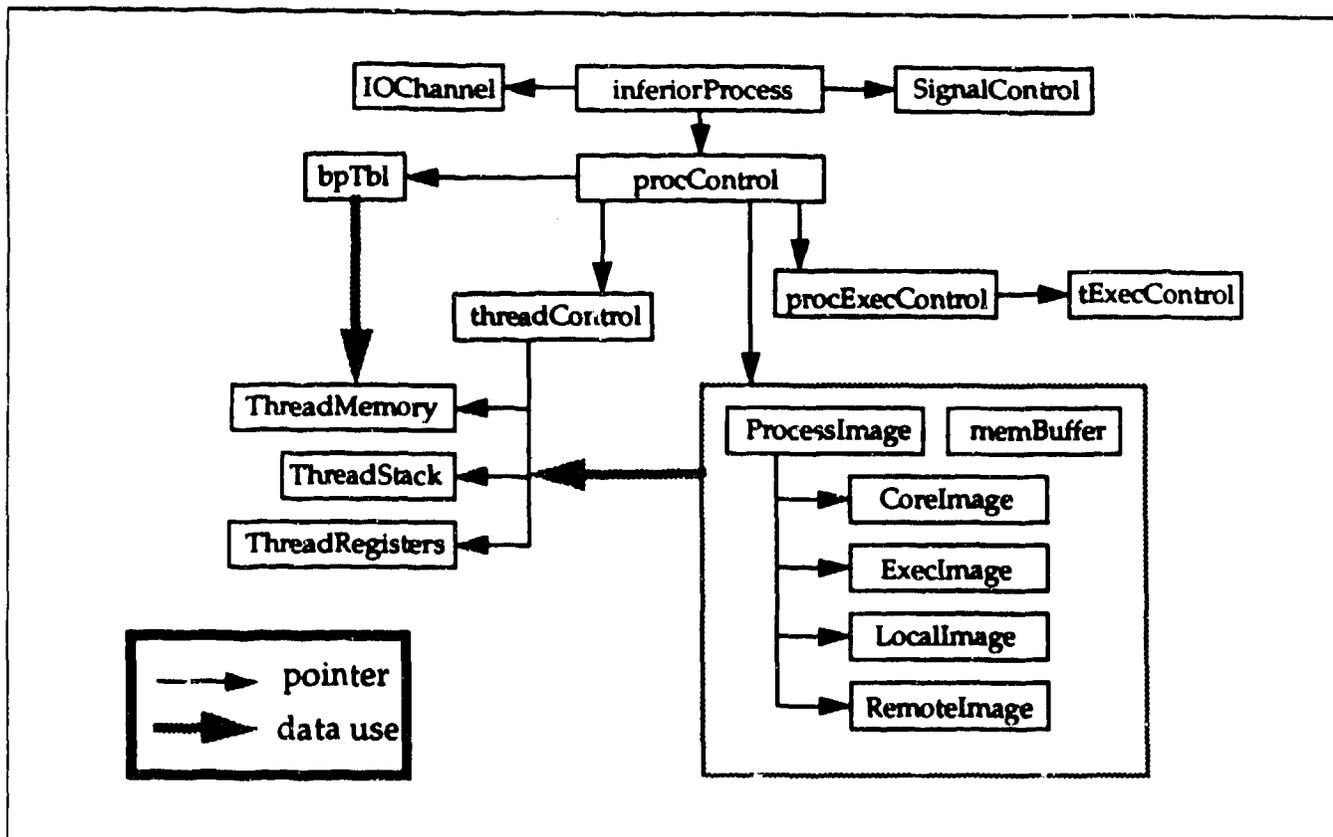
procControl Access to the process as a whole (such as stopping, starting, etc.).

threadControl Access to individual thread attributes (registers, memory, stack, etc.).

ProcessImage All manipulation of the connection to the remote server is managed within the `RemoteImage` object. The `RemoteImage` object is a derivation of the `ProcessImage` base class that defines the standard interface to a process image.

By using a standard base class for all access to images (be they executables, core files, or local or remote processes), the PI presents a common view of a process image. This minimizes the impact on the rest of CXdb when the internal

Figure 7. PI High-Level Organization



representation of a process image is modified, as was done when remote images were added.

With these abstractions in place, all the components of CXdb can make requests for control or information on the target process and the PI handles the activity regardless of the location of the process image.

7.0 Summary

The development of CXdb's remote debugging capabilities took roughly 1 person year (three developers for about 4 months) to complete. The development was done in these general phases:

1. Define the Remote Debugging Protocol.
2. Develop the MIG tools.

3. Develop the CXdb process image modifications and implement the ConvexOS remote server.

4. Develop the RTK server (this was really done concurrently with step 3).

By fully defining the remote protocol and developing the MIG tools up front, we saved valuable time during the implementation of the CXdb and remote server code. As development progressed, small oversights or misunderstandings in the RDP would be uncovered. The MIG tools made the task of updating the protocol code a snap.

Obviously, we have only begun to leverage the usefulness of remote debugging. There are several classes of debug operations that can benefit from this technology. Some of them are:

1. Kernel debugging. By embedding the remote server within the Kernel, CXdb could be used to perform kernel debugging on a remote machine. This would

- provide an improved debugging environment for kernel developers.
- 2. Remote debugging over dial-in lines. This would enable customer support personnel to debug processes running on a customer's machine using only a standard modem line. This would greatly enhance the level of support available to the customers.
- 3. Embedded systems. Any embedded system which can have a remote server built into it can benefit from this technology (in the same way as the Real-time system).

All-in-all, the development of CXdb's remote capabilities was a definite success and a step forward in both the functionality and the maintainability of the system.

8.0 Acknowledgments

I would like to acknowledge my colleagues who participated in the definition and development of CXdb's remote debugging capabilities, in particular, Gary Brooks, Rob Gordon, David Lingle, Steve Simmons, Ken Harward, Ray Cetrone, Keith Knox, and Lloyd Tharel. I would also like to thank Gary Brooks, Dave Lingle, and Jon Loeliger for their careful review of this paper.

9.0 Trademarks and Copyrights

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX computer corporation.

UNIX is a trademark of AT&T Bell Laboratories.

X Window System is a trademark of M.I.T.

A Complete Protocol Command List

This appendix contains a complete list and description of all the commands within the remote debugging protocol.

A.1 CONNECT Commands

C_CONFIG The C_CONFIG command is the first packet sent to the remote server after the low level connection (i.e. TCP/IP) has been established. This command will inform the remote server of the local client's protocol version and architecture. The remote server should verify that it can work properly with the local client and then reply with a message that indicates its architecture, protocol version, max packet size, and an indication of compatibility. It is the remote server's task to decide if the protocol versions are compatible.

C_DEBUG To aid in debugging the connection to the remote server and the operation of the server itself, servers should support a debug mode which can be enabled with this command. No reply is expected from this command.

C_ABORT Abort any operation the remote server may be performing and reset it to a known state. This can be used in response to a user interrupting the local client during a long operation on the remote (such as reading large amounts of data). There is no data associated with this command.

C_ERROR This message is initiated by the remote server. If any error other than normal error detection while handling a command packet is encountered, then a C_ERROR packet is sent to the debug client. It is the responsibility of the client to display the error message to the user.

C_TERMINATE Terminate the connection between the local client and the remote server. The remote server should release any resources it has acquired on behalf of this connection and then exit. If the remote server is controlling an executing process, then process should either be killed (if it was created) or detached (if it was attached). There is no data associated with this command.

A.2 FILE Commands

- F_OPEN** Requests the remote server to open a specified file for further processing. The open mode (i.e read, read/write, or write) will also be specified.
- The reply to the F_OPEN command will specify the file handle (like a file descriptor) for use in later commands referencing this file.
- F_CLOSE** Requests the remote server to close a file previously opened. The file to close is specified by file handle.
- F_SEEK** Requests the remote server to seek to a specified location within a given file. The seek is an absolute offset from the start of the file; relative seeks are not supported.
- F_READ** Requests the remote server to read a specified amount of data from a given file. The read begins at the current file position. The current file position is updated by the number of bytes read.
- The reply, which is a multi-packet message, contains the data requested.
- F_WRITE** Requests the remote server to write a specified amount of data to a given file. The writing begins at the current file position. The current file position is updated by the number of bytes written.
- F_SETCWD** Specifies the directory pathname that the remote server should use to interpret relative path names. The actual interpretation of the path name is server dependent.
- P_SETENV** Specifies the environment that a created target process will start with. This may not be applicable to all architectures.
- P_ATTACH** Attach to a process running on the remote host. The identifier of the process to attach to may differ between architectures. On most Unix systems it will be a process id (PID).
- P_DETACH** Detach the current process and let it continue running outside the remote servers control.
- P_CREATE** Create a process from the executable specified in a previous P_SETEXEC command. The arguments to supply the process are specified as part of this command. It is the responsibility of the server to perform wildcard expansion and I/O redirection based on the argument list.
- P_KILL** Terminate the target process. Do not send the reply until the target is killed.
- P_PROCINQ** Inquire on the target process' state. The reply includes, among other things, the number of threads, the signal and sub-code that caused it to stop or die and, CPU time consumed.
- P_THDINQ** Inquire on the state of all threads or a specific thread. A thread id of -1 indicates all threads. Information retrieved is similar to that of P_PROCINQ, but on a per-thread basis.
- P_GETCWD** Get the current working directory of the target process.
- P_SETDIR** Set the initial working directory for the target process. When the remote server creates the target process it will first *chdir* to this directory.

A.3 PROCESS Commands

- P_SETEXEC** Specifies the path name of the executable to manipulate. This must be specified before a subsequent P_CREATE command can be issued. It is the responsibility of the remote server to verify that the file exists, can be accessed, and is a valid executable.
- P_STATECHANGE** This message is initiated by the remote server. When the server sends this message it indicates that the target process has changed state (i.e. stopped) for some reason. The local client should send P_PROCINQ and P_THDINQ commands to determine the new state of the process.

P_STOP Stop the target process. The remote server should perform whatever operation is necessary to stop the execution of the target process. The reply should not be sent until the process has stopped.

P_THDSTEP Prepare the indicated thread to single step when the process is resumed. A signal may be specified to give the thread upon resumption.

P_THDCONT Prepare the indicated thread to continue when the process is resumed. A signal may be specified to give the thread upon resumption.

P_RESUME Resume the execution of the target process. The P_THDSTEP and P_THDCONT commands should have been used to specify how each thread will execute prior to using this command.

P_RDREGSET Read a specified register set from a specific thread within the target process. The binding of register set number to actual registers is architecture dependent.

P_WRREGSET Write a specified register set in a specific thread within the target process.

P_SEEK Requests the remote server to seek to a specified location within the process image. The seek is to a virtual address within the address space of the process; relative seeks are not supported.

P_READ Requests the remote server to read a specified amount of data from the process image. The read begins at the current image position. The current image position is updated by the number of bytes read. Since individual threads can have private memory the thread to read from must be specified.

P_WRITE Requests the remote server to write a specified amount of data to the process image. The writing begins at the current image position. The current image position is updated by the number of bytes written.

P_STDINDATA Sends data from the local client to be sent to the target process' stdin. This is normally data typed by the user of the local client.

P_STDOUTDATA This message is initiated by the remote server. When the server sends this message it indicates the target process has written data to stdout and it needs to be handled by the local client.

P_STDERRDATA This message is initiated by the remote server. When the server sends this message it indicates the target process has written data to stderr and it needs to be handled by the local client.

B Protocol Definition Example

A considerable amount of information was relayed in the description of the protocol definition. It is generally difficult to see how all the pieces fit together without a reasonable example. The example presented below is a trimmed version of the actual RDP used by CXdb. Figure 7 contains the protocol definition, Figure 8 contains the CXdb sender driver, figure 9 contains the CXdb receiver driver. Even though some packets have been trimmed from the definition, it is still very lengthy. Without the MIG tools, the job of constructing the protocol handling modules would have been very time-consuming, error-prone, and boring.

Figure 8. Example Protocol Definition

```
# Copyright (c) 1992 Convex Computer
# Corporation
# All rights reserved.
#
#+++++
# Protocol definition
#+++++

protocol: RDP
version: 1
max_packet: 10240
```

```
start_code_sets:
start_codes: CLTYPE Client types
  CXDB
  RTKDB
  ADTODB
  RTKRUN
end_codes:

start_codes: HWARECH Hardware arch
  C1
  C2
  C3
  MP1
end_codes:

start_codes: SWARCH Software arch
  CXOS9_1
  CXOS10_0
  RTK2_0
end_codes:

start_codes: OMODE File open modes
  RDONLY
  WRONLY
  RDWR
end_codes:

start_codes: REGSET Register sets
  SCALAR
  VECTOR
  COMM
end_codes:

start_codes: PSTATE Process state
  RUNNING
  STOPPED
  SIGNALED
  EXITED
end_codes:

start_codes: TSTATE Thread state
  RUNNING
  STOPPED
  DEAD
end_codes:
end_code_sets:

start_type: COMMAND CMD CFLD

start_category: CONNECT C
start_opcode: CONFIG
field: VERSION _4byte
field: CLTYPE _4byte CLTYPE
field: HWARECH _4byte HWARECH
field: SWARCH _4byte SWARCH
field: USER string
end_opcode:

start_opcode: PASSWORD
field: PASSWORD string
end_opcode:

start_opcode: DEBUG
field: FLAGS _4byte
field: LOG _4byte
field: LOGFILE string
end_opcode:

start_opcode: ABORT
end_opcode:

start_opcode: TERMINATE
end_opcode:

start_opcode: ERROR
field: MSG string
end_opcode:

end_category:

start_category: FILE F
start_opcode: OPEN
field: MODE _4byte OMODE
field: FILE string
end_opcode:

start_opcode: CLOSE
field: HANDLE _4byte
end_opcode:

start_opcode: SEEK
field: HANDLE _4byte
field: POS _8byte
end_opcode:

start_opcode: READ
field: HANDLE _4byte
field: NBYTES _8byte
```

end_opcode:

start_opcode: WRITE
field: HANDLE _4byte
field: TOTLEN _8byte
field: LAST _4byte
field: DATA buffer
end_opcode:

start_opcode: SETCWD
field: CWD string
end_opcode:

end_category:

start_category: PROCESS P
start_opcode: SETEXEC
field: PATH string
end_opcode:

start_opcode: SETENV
field: TOTLEN _8byte
field: LAST _4byte
field: DATA buffer
end_opcode:

start_opcode: ATTACH
include:
/* ConvexOS specific */
end_include:
field: PID _4byte
reset_offset:
include:
/* ConvexRTS/rtk specific */
end_include:
field: APPNAME string
end_opcode:
start_opcode: DETACH
end_opcode:

start_opcode: CREATE
field: TOTLEN _8byte
field: LAST _4byte
field: NARGS _4byte
field: ARGS buffer
end_opcode:

start_opcode: KILL
end_opcode:

start_opcode: PROCINQ
end_opcode:

start_opcode: THDINQ
field: ALL _4byte
field: TID _4byte
end_opcode:

start_opcode: SETDIR
field: PATH string
end_opcode:

start_opcode: STATECHANGE
field: PROCNUM _4byte
end_opcode:

start_opcode: STOP
end_opcode:

start_opcode: THDSTEP
field: TID _4byte
field: SIGNAL _4byte
end_opcode:

start_opcode: THDCONT
field: TID _4byte
field: SIGNAL _4byte
end_opcode:

start_opcode: RESUME
end_opcode:

start_opcode: RDREGSET
field: TID _4byte
field: REGSET _4byte REGSET
end_opcode:

start_opcode: WRREGSET
field: TID _4byte
field: TOTLEN _8byte
field: LAST _4byte
field: REGSET _4byte REGSET
field: DATA buffer
end_opcode:

start_opcode: SEEK
field: TID _4byte
field: VADDR _8byte

```
end_opcode:

start_opcode: READ
field: TID _4byte
field: NBYTES _8byte
end_opcode:

start_opcode: WRITE
field: TID _4byte
field: TOTLEN _8byte
field: LAST _4byte
field: DATA buffer
end_opcode:

start_opcode: STDINDATA
field: TOTLEN _8byte
field: LAST _4byte
field: DATA buffer
end_opcode:

start_opcode: STDOUTDATA
field: TOTLEN _8byte
field: LAST _4byte
field: DATA buffer
end_opcode:

start_opcode: STDERRDATA
field: TOTLEN _8byte
field: LAST _4byte
field: DATA buffer
end_opcode:

end_category:
end_type:

start_type: REPLY RPLY RFLD

start_category: CONNECT C
start_opcode: CONFIG
field: VERSION _4byte
field: COMPAT _4byte
field: PWREQ _4byte
field: HWRARCH _4byte HWRARCH
field: SWARCH _4byte SWARCH
end_opcode:

start_opcode: PASSWORD
field: VALID _4byte
end_opcode:

start_opcode: DEBUG
end_opcode:
start_opcode: ABORT
end_opcode:

start_opcode: TERMINATE
end_opcode:
end_category:

start_category: FILE F
start_opcode: OPEN
field: HANDLE _4byte
end_opcode:

start_opcode: CLOSE
end_opcode:

start_opcode: SEEK
field: POS _8byte
end_opcode:

start_opcode: READ
field: TOTLEN _8byte
field: LAST _4byte
field: DATA buffer
end_opcode:

start_opcode: WRITE
field: NBYTES _8byte
end_opcode:

start_opcode: SETCWD
end_opcode:
end_category:

start_category: PROCESS P
start_opcode: SETEXEC
end_opcode:

start_opcode: SETENV
end_opcode:

start_opcode: ATTACH
end_opcode:

start_opcode: DETACH
end_opcode:
```

```

start_opcode: CREATE
end_opcode:

start_opcode: KILL
end_opcode:

start_opcode: PROCINQ
field: STATE _4byte PSTATE
field: TCNT _4byte
field: SIGNAL _4byte
field: SUBCODE _4byte
field: ESTATUS _4byte
field: CORE _4byte
field: UCPUSEC _4byte
field: UCPUMS _4byte
field: SCPUSEC _4byte
field: SCPUMS _4byte
field: PID _4byte
end_opcode:

start_opcode: THDINQ
field: TOTLEN _8byte
field: LAST _4byte
field: NTHDS _4byte
include:
/* Remaining fields repeat per
thread. Offsets are relative to
thread entry.*/

end_include:
reset_offset:
field: TID _4byte
field: STATE _4byte TSTATE
field: SIGNAL _4byte
field: SUBCODE _4byte
field: WPID _4byte
field: UCPUSEC _4byte
field: UCPUMS _4byte
field: SCPUSEC _4byte
field: SCPUMS _4byte
include:
/* TNAME field only exists for
ConvexRTS/rtk architecture */
end_include:
field: TNAME string
end_opcode:

start_opcode: SETDIR
end_opcode:

start_opcode: STATECHANGE
end_opcode:

start_opcode: STOP
end_opcode:

start_opcode: THDSTEP
end_opcode:

start_opcode: THDCONT
end_opcode:

start_opcode: RESUME
end_opcode:

start_opcode: RDREGSET
field: TOTLEN _8byte
field: LAST _4byte
field: DATA buffer
end_opcode:

start_opcode: WRREGSET
end_opcode:

start_opcode: SEEK
field: VADDR _8byte
end_opcode:

start_opcode: READ
field: TOTLEN _8byte
field: LAST _4byte
field: DATA buffer
end_opcode:

start_opcode: WRITE
field: NBYTES _8byte
end_opcode:

start_opcode: STDINDATA
end_opcode:

start_opcode: STDOUTDATA
end_opcode:

start_opcode: STDERRDATA

```

```

end_opcode:
end_category:

end_type:
end_protocol:

```

Figure 9. Example Sender Specification

```

# Copyright (c) 1992 Convex Computer
    Corporation
# All rights reserved.
#

create: sender
name_pattern: cltsnd_@c_@t

start_type: COMMAND

start_category: CONNECT
start_opcode: _all_
end_opcode:
end_category:

start_category: FILE
start_opcode: _all_
end_opcode:
end_category:

start_category: PROCESS

Just do PID, APPNAME is for
realtime
start_opcode: ATTACH
select (PID)
end_opcode:

start_opcode: _all_
end_opcode:

end_category:
end_type:

```

Figure 10. Example Receiver Specification

```

# Copyright (c) 1992 Convex Computer
    Corporation
# All rights reserved.
#
create:
receiver=dispatchPacket,msg=printf
name_pattern: cltrcv_@c_@t

start_include:
#include <stdio.h>
#include "common/ExecObject.h"
#include "pi/RemoteImage.h"
#include "pi/SigchldQueue.h"
#include "iomm/rmtReceive.h"
end_include:

start_type: COMMAND

start_category: CONNECT

start_opcode: ERROR
call handleRemoteError
end_opcode:

end_category:

start_category: PROCESS
start_opcode: STATECHANGE
call SigchldQueue.handleStatechange
end_opcode:

start_opcode: STDOUTDATA
call handleRemoteStdout
end_opcode:

start_opcode: STDERRDATA
call handleRemoteStderr
end_opcode:

end_category:
end_type:

start_type: REPLY
start_category: FILE
start_opcode: OPEN
handle_fail ExecObject::handleOpenFail
call ExecObject::handleOpenReply
end_opcode:

start_opcode: CLOSE
handle_fail ExecObject::handleCloseFail
call ExecObject::handleCloseReply
end_opcode:

```

```

start_opcode: READ
handle_fail ExecObject::handleReadFail
call ExecObject::handleReadReply
end_opcode:

start_opcode: WRITE
handle_fail ExecObject::handleWriteFail
call ExecObject::handleWriteReply
end_opcode:

start_opcode: SEEK
handle_fail ExecObject::handleSeekFail
call ExecObject::handleSeekReply
end_opcode:

start_opcode: SETCWD
handle_fail ServerObj::handleSetCwdFail
call ServerObj::handleSetCwdReply
end_opcode:

start_opcode: SETFS
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleStandardReply
end_opcode:

end_category:

start_category: CONNECT

start_opcode: CONFIG
handle_fail ServerObj::handleConfigFail
call ServerObj::handleConfigReply
end_opcode:

start_opcode: PASSWORD
handle_fail ServerObj::handleStdFailure
call ServerObj::handlePasswordReply
end_opcode:

start_opcode: DEBUG
handle_fail ServerObj::handleStdFailure
call ServerObj::handleStandardReply
end_opcode:

start_opcode: ABORT
handle_fail ServerObj::handleAbortFail
call ServerObj::handleAbortReply
end_opcode:

end_category:

start_category: PROCESS

start_opcode: SETEXEC
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleStandardReply
end_opcode:

start_opcode: SETENV
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleStandardReply
end_opcode:

start_opcode: ATTACH
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleStandardReply
end_opcode:

start_opcode: DETACH
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleStandardReply
end_opcode:

start_opcode: CREATE
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleStandardReply
end_opcode:

start_opcode: KILL
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleStandardReply
end_opcode:

start_opcode: PROCINQ
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleProcStateReply
end_opcode:

start_opcode: THDINQ
handle_fail RemoteImage::handleStdFail
select (FAILURECODE, _fc, unsigned int)
select (TOTLEN, _totlen, unsigned long
        long)
select (LAST, _last, boolean)
select (NTHDS, _nthds, unsigned int)
rawcall
RemoteImage::handleThreadStateReply
end_opcode:

start_opcode: SETDIR
handle_fail ServerObj::handleStdFailure
call ServerObj::handleStandardReply
end_opcode:

start_opcode: STOP
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleStandardReply
end_opcode:

```

```

start_opcode: THDSTEP
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleStandardReply
end_opcode:

start_opcode: THDCONT
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleStandardReply
end_opcode:

start_opcode: RESUME
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleStandardReply
end_opcode:

start_opcode: RDREGSET
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleReadRegsReply
end_opcode:

start_opcode: WRREGSET
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleStandardReply
end_opcode:

start_opcode: SEEK
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleSeekReply
end_opcode:

start_opcode: READ
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleReadReply
end_opcode:

start_opcode: WRITE
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleWriteReply
end_opcode:

start_opcode: STDINDATA
handle_fail RemoteImage::handleStdFail
call RemoteImage::handleStandardReply
end_opcode:
end_category:

end_type:

```

This is an example of the generated code for the sender driver that handles the C_CONFIG and P_ATTACH commands.

```

void
cltsnd_C_CONFIG_COMMAND(
BaseRdpTransport & link, unsigned int
_version, unsigned int _cltype, unsigned
int _hwarch, unsigned int _swarch,
unsigned int _user_len, const void *
_user_ptr)
{
char *datap = Packet + RDP_PACKET_HDRSZ;
int type = RDP_PACKET_COMMAND;
int category = RDP_CATEGORY_CONNECT;
int opcode = RDP_CMD_C_CONFIG;
DataLen = 0;

SET_RDP_HDR_FLD(Packet, TYPE, type);
SET_RDP_HDR_FLD(Packet, CATEGORY,
category);
SET_RDP_HDR_FLD(Packet, OPCODE, opcode);

/* Now that the header is setup, do each
field */

DataLen += RDP_CFLD_C_CONFIG_VERSION_OFF;
SET_RDP_CMD_FLD(datap, C_CONFIG,
VERSION, _version);
DataLen += RDP_CFLD_C_CONFIG_VERSION_LEN;

DataLen += RDP_CFLD_C_CONFIG_CLTYPE_OFF;
SET_RDP_CMD_FLD(datap, C_CONFIG,
CLTYPE, _cltype);
DataLen += RDP_CFLD_C_CONFIG_CLTYPE_LEN;

DataLen += RDP_CFLD_C_CONFIG_HWARCH_OFF;
SET_RDP_CMD_FLD(datap, C_CONFIG,
HWARCH, _hwarch);
DataLen += RDP_CFLD_C_CONFIG_HWARCH_LEN;

DataLen += RDP_CFLD_C_CONFIG_SWARCH_OFF;
SET_RDP_CMD_FLD(datap, C_CONFIG,
SWARCH, _swarch);
DataLen += RDP_CFLD_C_CONFIG_SWARCH_LEN;

DataLen += RDP_CFLD_C_CONFIG_USER_OFF;
SET_RDP_CMD_STRING_LEN(datap,
C_CONFIG, USER, _user_len);
DataLen += RDP_AFLD_STRING_STRLEN_LEN;
GET_RDP_CMD_STRING_PTR(datap, C_CONFIG,
USER, datap);
strcpy(datap, (const char*)_user_ptr);
DataLen += _user_len + 1;

```

Figure 11. Example Generated Sender Code

References

```
SET_RDP_HDR_FLD(Packet, DATALEN,
                DataLen);

/* Now that the packet is built, send it */

link.write(Packet, RDP_PACKET_HDRSZ +
           DataLen);
return;
}

void
cltsnd_P_ATTACH_COMMAND(
    BaseRdpTransport & link,
    unsigned int _pid)
{
    char *datap = Packet + RDP_PACKET_HDRSZ;
    int type = RDP_PACKET_COMMAND;
    int category = RDP_CATEGORY_PROCESS;
    int opcode = RDP_CMD_P_ATTACH;
    DataLen = 0;

    SET_RDP_HDR_FLD(Packet, TYPE, type);
    SET_RDP_HDR_FLD(Packet, CATEGORY,
                   category);
    SET_RDP_HDR_FLD(Packet, OPCODE, opcode);

    /* Now that the header is setup, do
       each field */

    DataLen = RDP_CFLD_P_ATTACH_PID_OFF;
    SET_RDP_CMD_FLD(datap, P_ATTACH,
                   PID, _pid);
    DataLen += RDP_CFLD_P_ATTACH_PID_LEN;

    SET_RDP_HDR_FLD(Packet, DATALEN,
                   DataLen);

    /* Now that the packet is built, send it */

    link.write(Packet, RDP_PACKET_HDRSZ +
               DataLen);
    return;
}
```

References

- [BuCh91] Buyse, R. and Chiarelli, M., "A User Interface Strategy for CXdb", presented at Xhibition 91.
- [Conv91a] CONVEX CXdb Reference, 1st Edition, Convex Computer Corporation (1991).
- [Conv91b] CONVEX CXdb User's Guide, 1st Edition, Convex Computer Corporation (1991).
- [Lawr90] Lawrence, S., "Mixed Mode Debugging: A High-Level Multi-process Debugger for pSOS[™] and Unix", BBN Advanced Computers, Inc., August, 1990.
- [Stal89] GDB Manual, Third Edition, Richard M. Stallman, 1989.
- [StBr91] Streepy, L., Brooks, G., Hansen, G., Simmons, S., "CXdb: A New View on Optimizations", Proceedings of the Supercomputer Debugging Workshop '91.
- [WeMi84] Welles, C. and Milliken, W., "Loader Debugger Protocol", Arpa Internet RFC-909, BBN Communications Corp., July 1984.



CXdb: The Road to Remote Debugging

**Larry V. Streepy, Jr.
Rob Gordon, and Dave Lingle**

Convex Computer Corporation

October 5, 1992



Introduction

- ◆ **Motivation**
- ◆ **Protocol Description**
- ◆ **Message Interface Generator**
- ◆ **Server Overview**
- ◆ **CXdb Abstractions**
- ◆ **Conclusions and Future Directions**

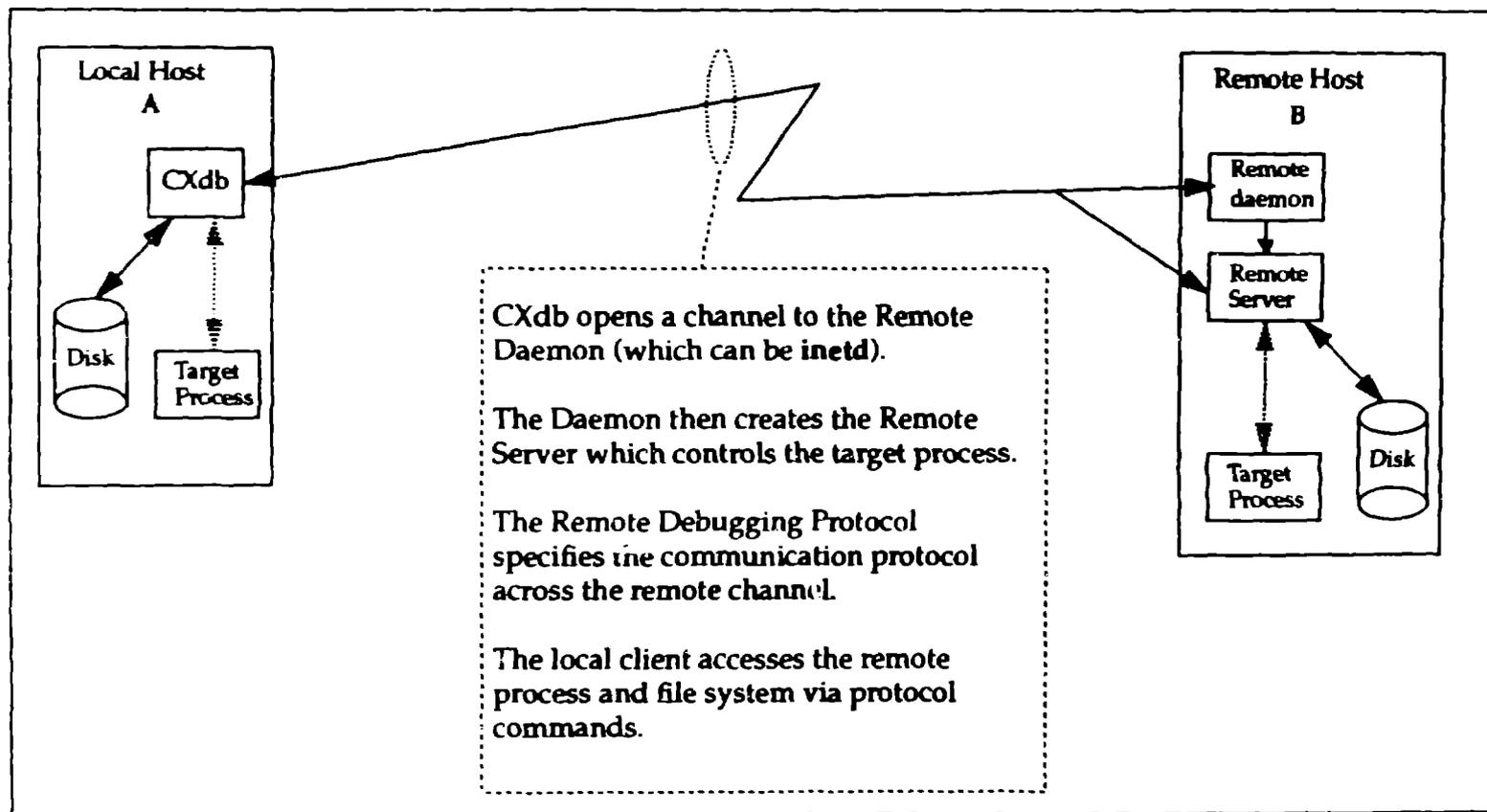


Motivation

- ◆ **Typical computing configurations include cooperative networks with multiple, often heterogeneous, hosts.**
- ◆ **Many special-purpose compute servers, real-time systems for example, require front-end machines to provide access and control.**
- ◆ **While performing kernel debugging the application environment of the target machine is not active.**



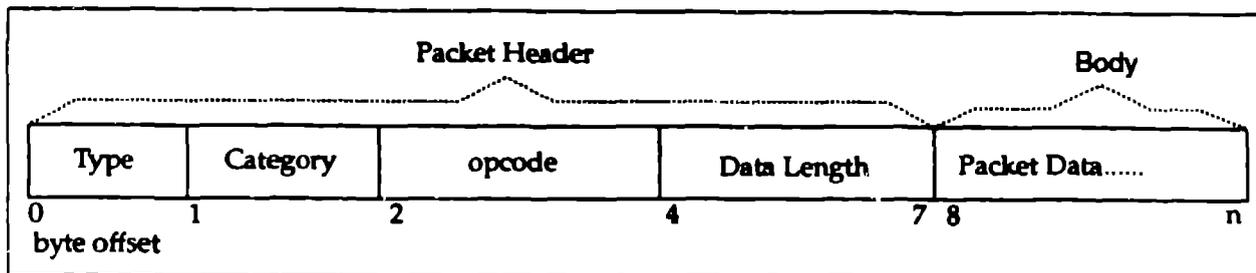
Remote Debugging Environment





Remote Protocol

RDP Packet Layout



Type There are two types of RDP packets: Command and Reply. Not all command packets require a reply. Strict command-reply model.

Category There are three major operation categories:

CONNECT Control of remote connection and configuration. Includes session initiation, version arbitration, configuration control, and session termination.

FILE Access to files on the remote host. Includes open, seek, read, write, and close.

PROCESS Access to and control of the remote process. Includes creating, attaching, and detaching a remote process; access to process memory, registers, attributes, and state.

Opcode The specific operation, or command, within a category. There are currently over 120 commands in all three categories.

Data Length The length of the body of the packet. This may be zero.

Packet Data The data associated with the command or reply, if any.



Message Interface Generator

Design Goals:

- ◆ **Decrease the time required to develop the protocol support modules.**
- ◆ **Increase the maintainability of the protocol support modules.**
- ◆ **Support development of servers in both C and C++.**
- ◆ **Support our automated testing facilities already in use on CXdb.**

Supported Features:

- ◆ **Generation of test drivers for use in automated testing.**
- ◆ **Generation of sending functions that construct and send protocol packets.**
- ◆ **Generation of receiving functions that break apart protocol packets.**
- ◆ **Generates code in both C and C++.**



Protocol Definition

- ◆ Machine-processable description of the protocol.
- ◆ Used to generate an include file which contains manifest constants that describe the protocol.
- ◆ Used to control the process of generating source modules that automatically handle operations on the protocol packets.

High Level Protocol Definition Structure

```
PROTOCOL: <name>
VERSION: <version>
MAX_PACKET: <size>

START_CODE_SETS:
<code set definitions>
END_CODE_SETS:

START_TYPE:
<packet type definitions>
END_TYPE:

END_PROTOCOL:
```



Example Definition

protocol: RDP

version: 1

max_packet: 10240

start_code_sets:

start_codes: CLTYPE Client types

CXDB

RTKDB

end_codes:

start_codes: HWARECH Hardware arch

C1

C2

C3

MP1

end_codes:

end_code_sets:

start_type: COMMAND CMD CFLD

start_category: CONNECT C

start_opcode: CONFIG

field: VERSION _4byte

field: CLTYPE _4byte CLTYPE

field: HWARECH _4byte HWARECH

field: SWARCH _4byte SWARCH

field: USER string

end_opcode:



start_opcode: DEBUG
field: FLAGS _4byte
field: LOG _4byte
field: LOGFILE string
end_opcode:

start_opcode: ABORT
end_opcode:

start_opcode: TERMINATE
end_opcode:

start_opcode: ERROR
field: MSG string
end_opcode:

end_category:

start_category: FILE F
start_opcode: OPEN
field: MODE _4byte OMODE
field: FILE string
end_opcode:

start_opcode: WRITE
field: HANDLE _4byte
field: TOTLEN _8byte
field: LAST _4byte
field: DATA buffer
end_opcode:

end_category:



start_category: PROCESS P

start_opcode: ATTACH

include:

/* ConvexOS specific */

end_include:

field: PID _4byte

reset_offset:

include:

/* ConvexRTS/rtk specific */

end_include:

field: APPNAME string

end_opcode:

start_opcode: CREATE

field: TOTLEN _8byte

field: LAST _4byte

field: NARGS _4byte

field: ARGS buffer

end_opcode:

start_opcode: STATECHANGE

field: PROCNUM _4byte

end_opcode:

end_category:

end_type:

end_protocol:



Driver Specifications

- ◆ **The MIG tools generate source code to manage the packets within the protocol.**
- ◆ **The source code generation is controlled by the protocol definition and a driver specification.**
- ◆ **Four kinds of drivers are supported: generators, dumpers, senders, and receivers.**
- ◆ **These four types are also broken into two general categories: senders (generators and senders) and receivers (dumpers and receivers).**
[isn't the evolution of names wonderful?]



Example Driver Spec

create: sender

name_pattern: cltsnd_%c_%t

start_type: COMMAND

start_category: CONNECT

start_opcode: **_all_**

end_opcode:

end_category:

start_category: FILE

start_opcode: **_all_**

end_opcode:

end_category:

start_category: PROCESS

Just do PID, APPNAME is for real-time

start_opcode: ATTACH

select(PID)

end_opcode:

start_opcode: **_all_**

end_opcode:

end_category:

end_type:



Example Receiver Spec

```
create: receiver=dispatchPacket,msg=printf  
name_pattern: cltrcv_%c_%t
```

```
start_include:
```

```
#include <stdio.h>  
#include "common/ExecObject.h"  
#include "pi/RemoteImage.h"  
#include "pi/SigchldQueue.h"  
#include "iommm/rmtReceive.h"
```

```
end_include:
```

```
start_type: COMMAND
```

```
  start_category: CONNECT
```

```
    start_opcode: ERROR
```

```
    call handleRemoteError
```

```
    end_opcode:
```

```
  end_category:
```

```
  start_category: PROCESS
```

```
    start_opcode: STATECHANGE
```

```
    call SigchldQueue.handleStatechange
```

```
    end_opcode:
```

```
    start_opcode: STDOUTDATA
```

```
    call handleRemoteStdout
```

```
    end_opcode:
```

```
  end_category:
```

```
end_type:
```



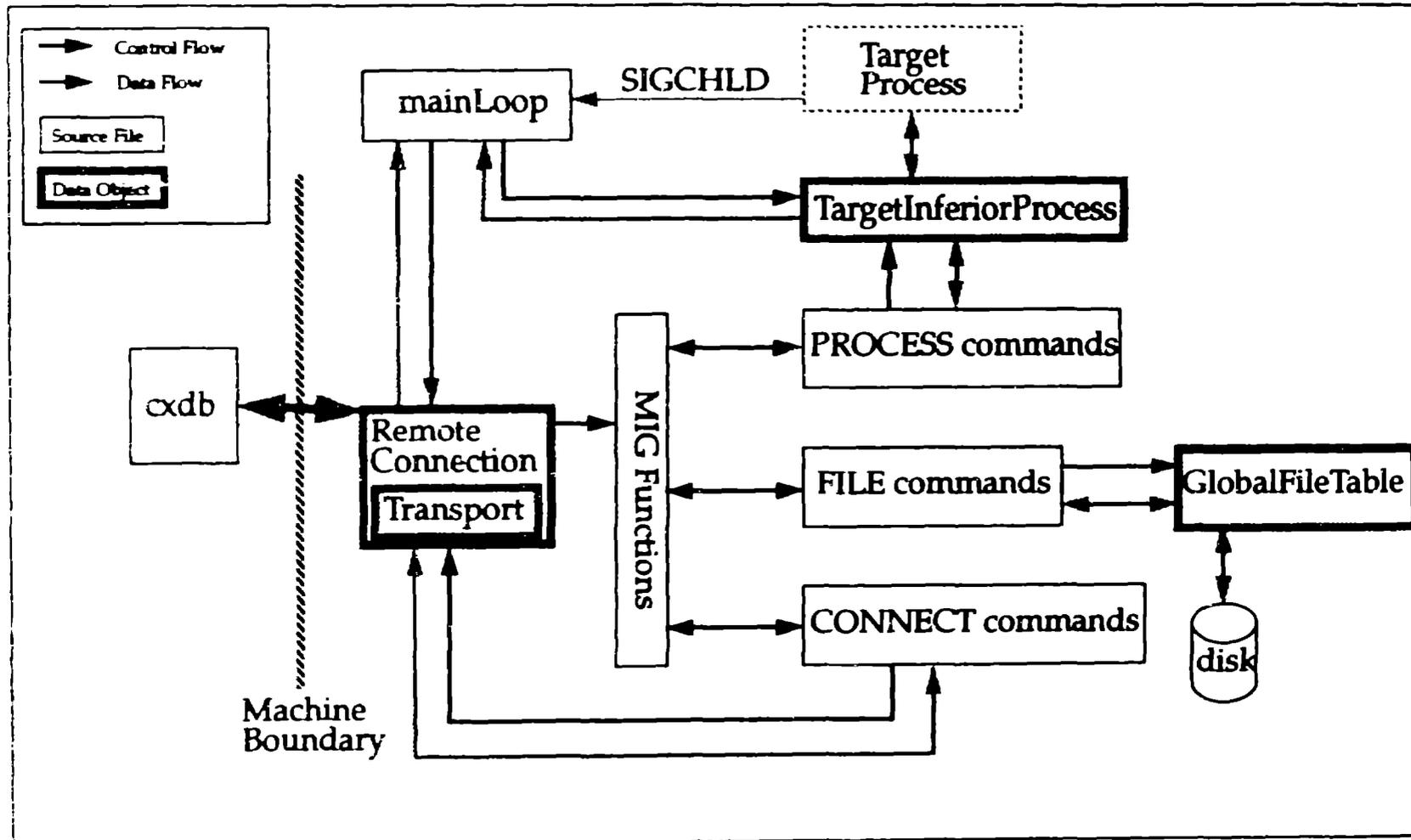
MIG Usage Experience

- ◆ All design goals were met.
- ◆ Development time was greatly reduced and maintainability was dramatically increased.
- ◆ The table below shows some source code statistics.

Source	Lines
MIG source code - Perl (22% comments)	3320
Protocol Definition	663
Driver specifications	991
Total	4974
Generated source code	21341

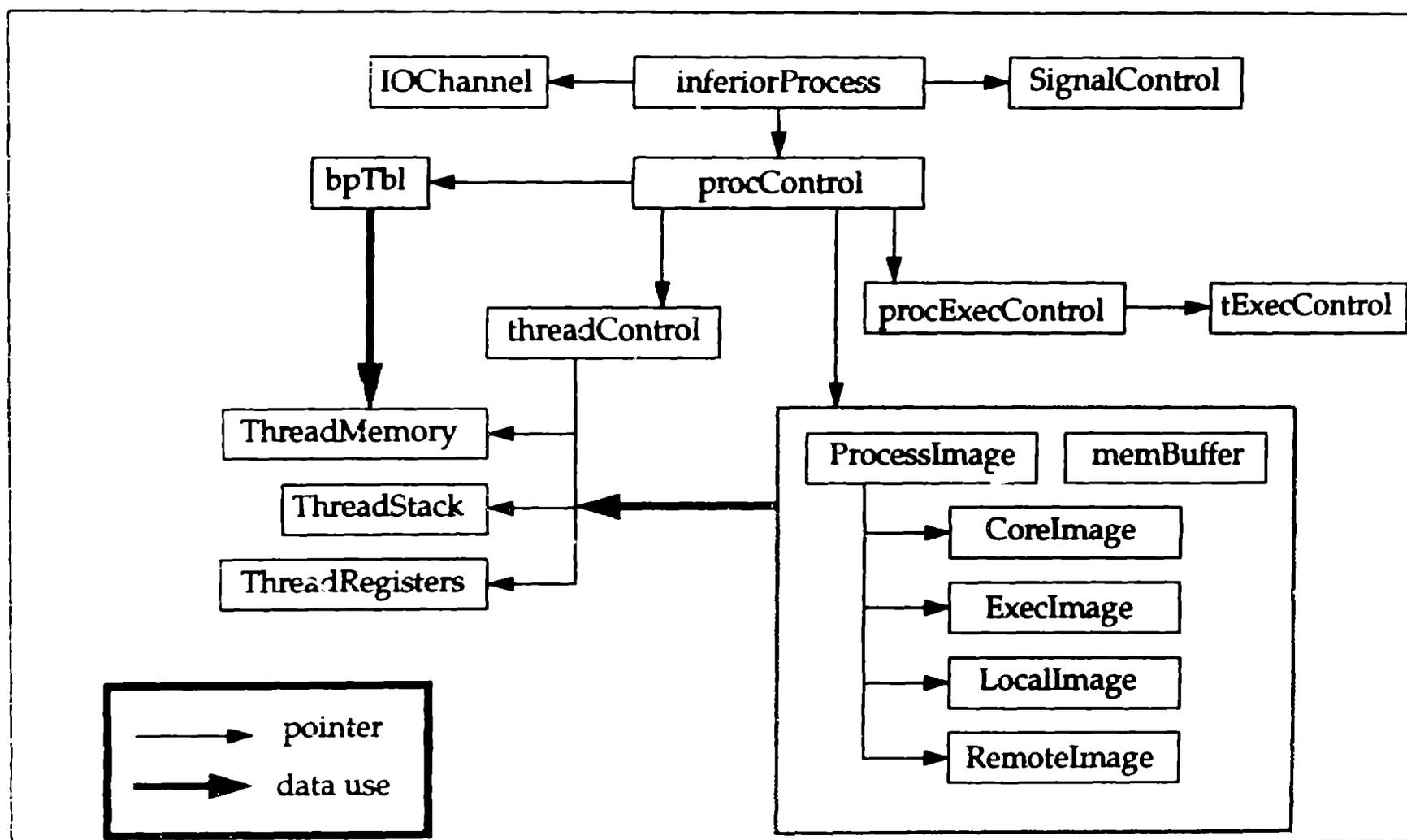


Remote Server Architecture





CXdb Process Interface Abstractions





Conclusions and Future Directions

- ◆ **The implementation of the CXdb remote debugging system used several features not typically found in earlier (or traditional) systems:**
 - ⇒ The development of the MIG decreased development time and maintenance overhead in the protocol manipulation routines.
 - ⇒ Software abstractions within the local debug client minimized the modifications required to implement remote capabilities.
 - ⇒ The clean separation of tasks between the debug client and remote server decreased the complexity and development time of the remote server. The remote server is a machine level debugger.
- ◆ **Several classes of debug operations can benefit from remote debugging technology: Kernel debugging, debugging over dial-in lines, and handling embedded systems.**

User Needs Discussion Summary

The following is an unordered list of opinions and desired debugging capabilities expressed by the group during the user needs discussion.

minimize context switches

intuitive and familiar user interface
easy to use for the first-casual user
easy access to complex features

breakpoint dependencies
break at location A if last break location was location B

support a mix of shared memory and distributed memory models

overcome the user education problem

support debugging large codes
debugger impact on code performance an issue for large codes

effectively handle the transition from fortran 77 to fortran 90
performance, complexity and portability concerns

provide tools to debug code someone else wrote
program decomposition, etc.

MIMD extension of where tree

fast dynamic print statements

fast tracing via patching and/or hardware

post mortem static analysis tools
apply fortran heuristics

debugging support for homogeneous clusters of workstations

hardware support for profiling, state at interrupt, watchpoints, tracing (buffer)

tool integration

debug optimized code
code in continuous state of development

encourage users to use a debugger
overcome perception that debuggers are hard to use

locate source statement that caused program abort

standards would facilitate debugger development
for example, user (command line and GUI) and symbol-table interface

effective support for low-level debugging, due to
must occasionally debug code with no symbol table
program state has changed from abort condition
sometimes required due to lack of debugger functionality

graphical representation of
program structure integrated with process control
data structures

breakpoint at entry if called by a particular routine

incremental compilation (patching) linked with breakpoints



SD '92

**P
r
o
c
e
e
d
i
n
g
s**

LANL

LAUR #

92-562