

Hob Nob

Anuradha Vaidyanathan

Hob nob: to have or not have

Outline

- Introduction
- Goals
- Architecture Research
- Systems Research

Introduction

- Two degrees
 - BS, MS
- Two research passions
 - Architecture, Operating Systems
- Two places with people I love
 - Bangalore, (mom,pop), Raleigh, NC (best friends)
- Two strengths
 - Insomniac, patient

Goals

- Hands on experience with research problems
- Hope to someday get a PhD and be a lazy professor
 - I love teaching!
- Biggest Inspiration: Richard Feynman

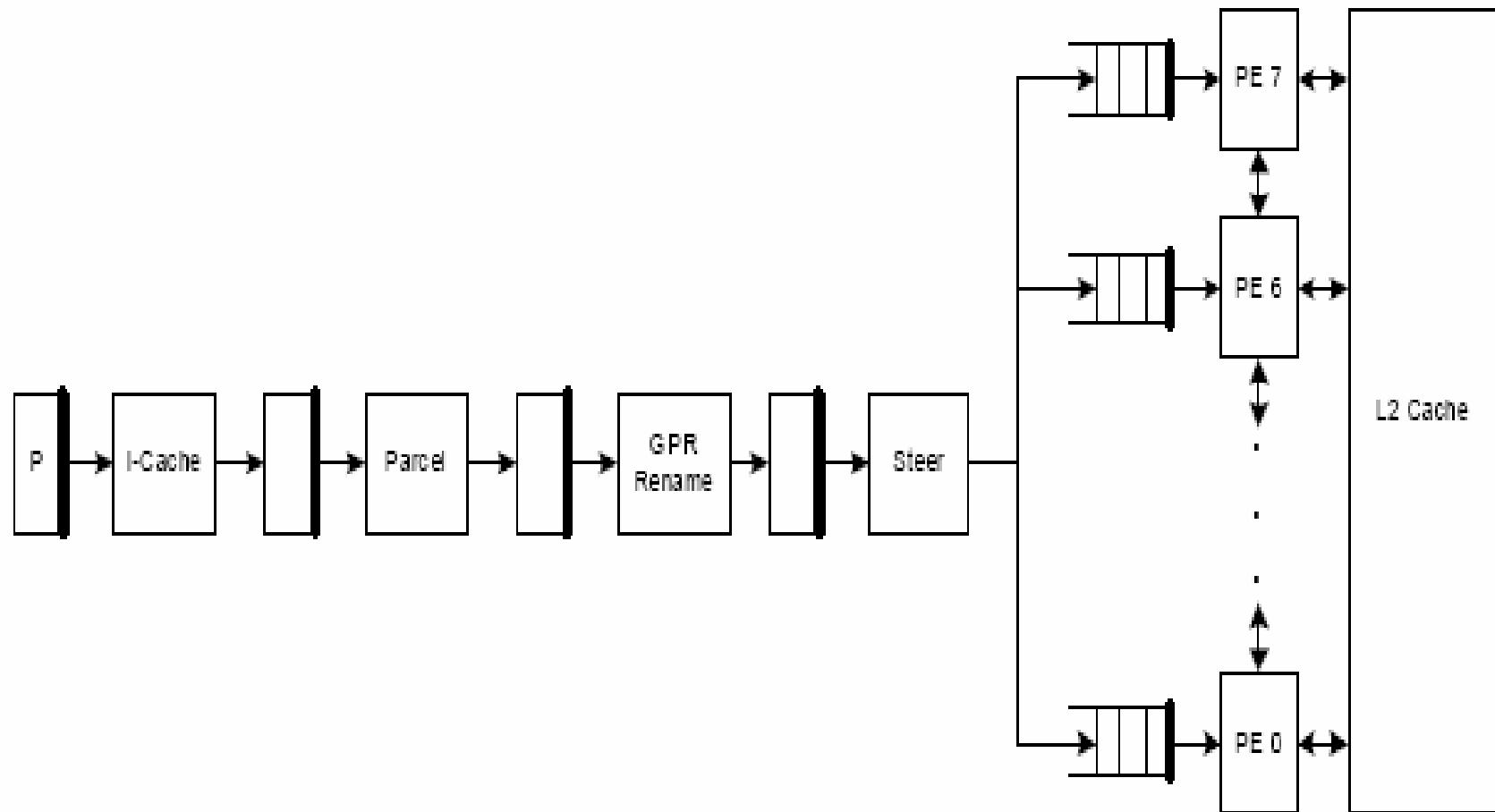
Architecture Research

- ILDP
- Poseidon
- DataScalar
- CMP Synchronization

ILDLP [Summer 00]

- Hypothesis - Simple clusters with in-order issue lead to reduced complexity
- Related Work
 - Alpha 21264
 - Palacharala and Jouppi
 - ILDP, ISCA '02
- Motivation - Fast clock with shallow pipelines
- My contribution - Measuring inter-cluster communication

ILD P Processor



Methodology

- **Simulated** - *The 'simple clusters' that were designed basically had one FIFO in them and so the issue-width for each cluster was one.*
 - *Simplescalar 2.x*
 - *SPEC Benchmarks*
- **Steering Method** - *Dependence based heuristic*
- **Measuring**
 - *Critical Delays, Not at Head-of-FIFO, Histogram of full FIFOs, Empty FIFOs, Issue Percentages, FIFO full*
 - **Inter-cluster Communication (in this talk)**

Steering Algorithm

```
rd <-- rs,rt,rj
```

Instruction Format

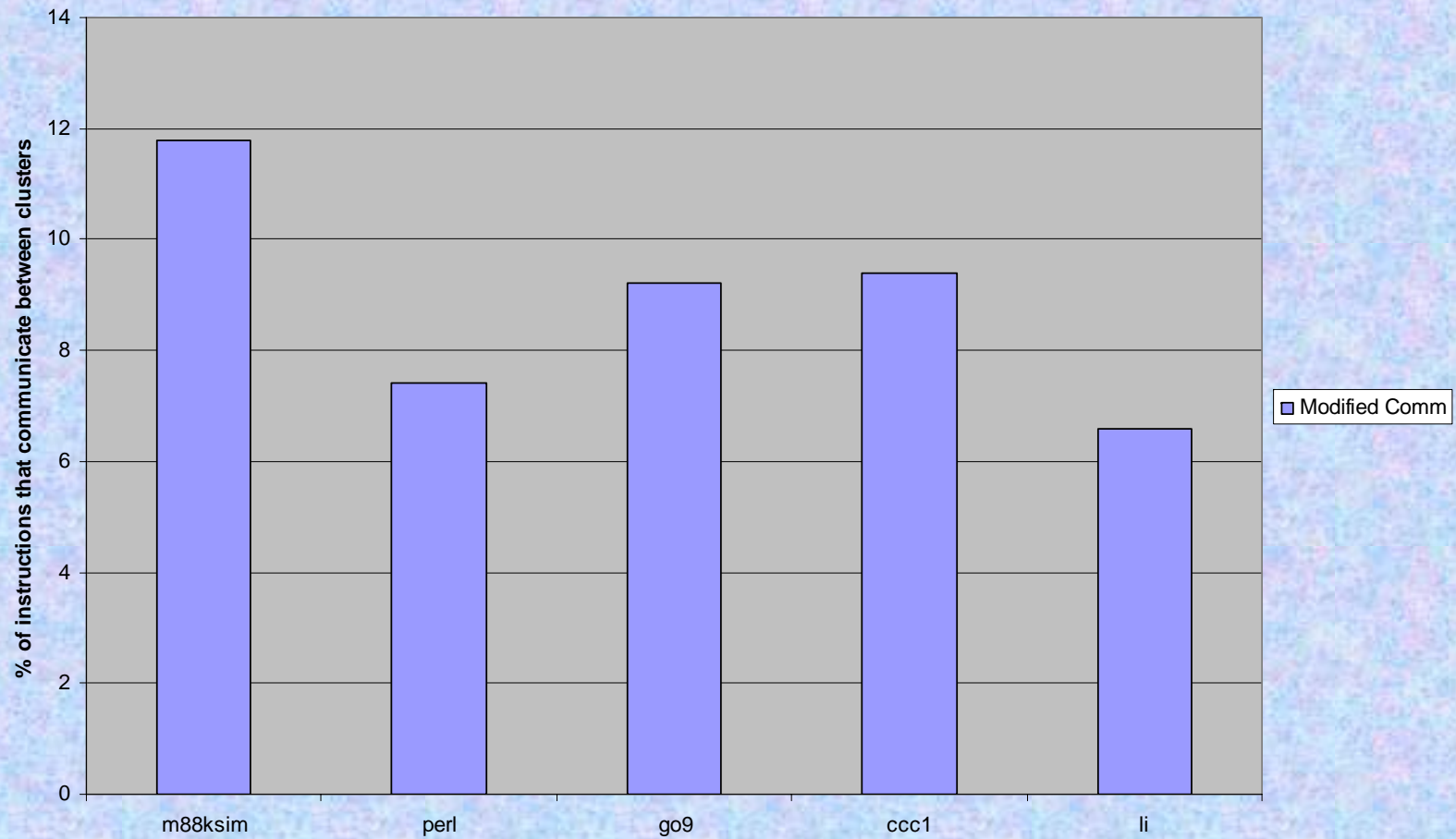
- Identify the creators of rs,rt and rj
- If there are no creators, steer the instruction to the shortest FIFO.
- If there is a creator for one of the three registers:
 - Check to see if the creating instruction has already issued.
 - If it has, check to see if the FIFO it issued from is full.
 - If that FIFO is not full, steer this instruction to that FIFO.
 - If that FIFO is full, steer this instruction to the shortest FIFO.

Steering Algorithm

- If the creating instruction has not issued, check to see if the FIFO it was steered to is full.
 - If that FIFO is not full, steer this instruction to that FIFO.
 - If that FIFO is full, find the relative position of the creating instruction within the FIFO and steer this instruction to the FIFO which is as long as the position in which the creator is sitting.
 - If the creator is in the last slot of a FIFO, steer the instruction to the Longest FIFO available.
- If there is a creator for more than one of the three registers:
 - Apply the steps applied to the case when we had one creator.
 - If the creating instruction(s) have issued, try to steer it to the FIFO from which the creator(s) issued.
 - If the FIFO of the creator(s) are full, steer the instruction to the shortest FIFO.
 - If the creator's have not issued, find the relative positions of the creators within their respective FIFO's.
 - Depending on which creator is at a latter position, try to steer the instruction to that FIFO or to a FIFO which is as long as the position in which the creator at the latter relative position is sitting.

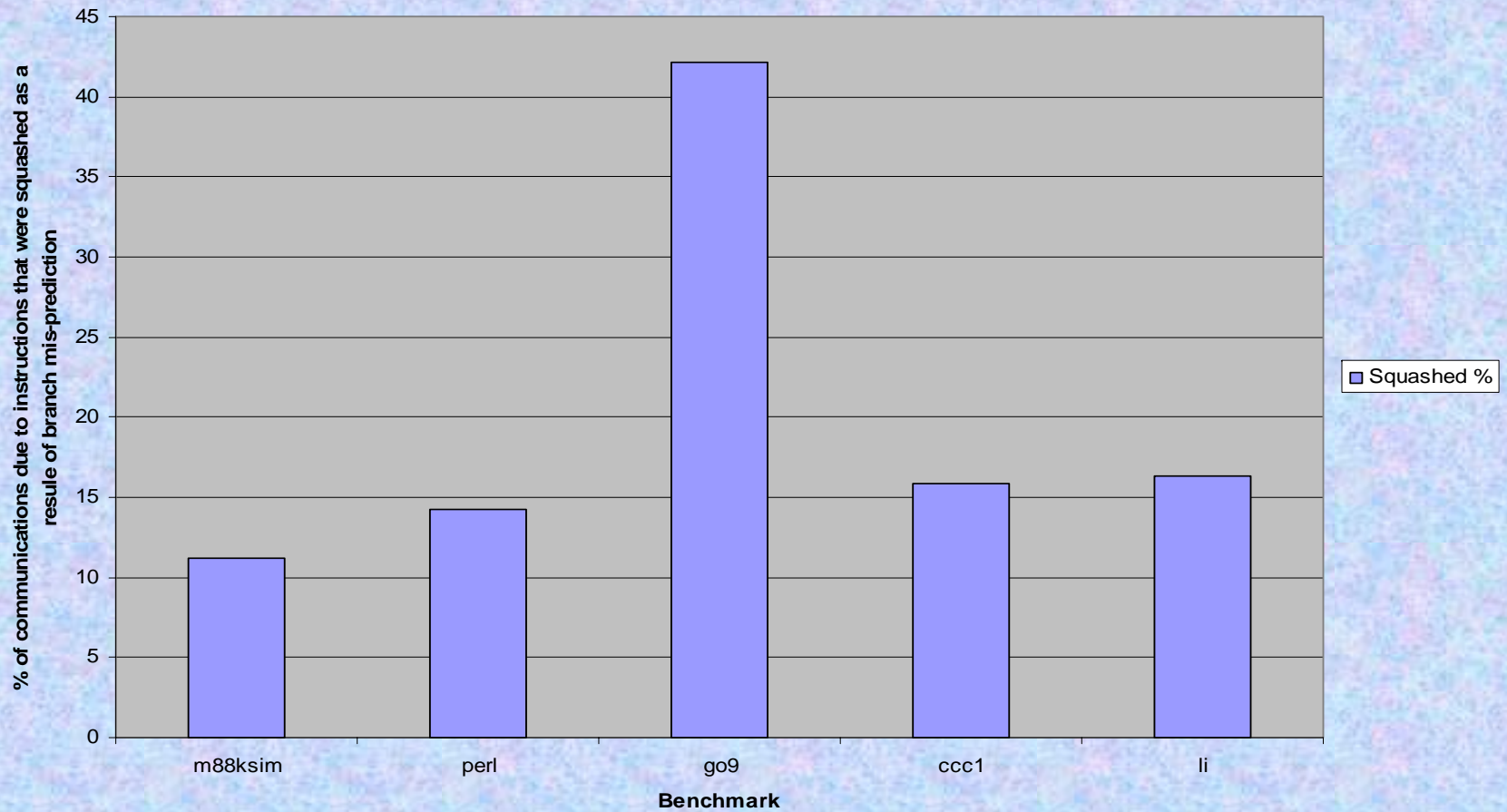
Results

Inter-Cluster Communication



Results

Percentage of bad communications



Architecture Research

- ~~ILDP~~
- Poseidon
- DataScalar
- CMP Synchronization

Poseidon [MS Thesis, 02]

- **Hypothesis** - Hardware support for security is a good use for some of the billions of transistors
- **Related Work**
 - StackGhost (software)
 - RAS (Skadron et.al)
- **Motivation** - Buffer Overflows are a conduit for > 60% of attacks
- **My contribution** - Design and implementation of *Shadow Stack*. (PLDI '02)

The Shadow Stack

- Key Observation : Call-return semantics are potentially violated during an attack
- Can we monitor the calls and returns and flag potential attacks?
- Almost all of the time..

Design

Address	Stack Pointer of JAL/JALR (r[29])
---------	-----------------------------------

- LIFO with two entries
 - Address - Next PC for calls and value of r31 for returns
 - Stack Pointer at the time of dispatch

Operations

A LIFO structure and can carry out the following operations:

- push()
- pop()
- top()
- empty()
- clear()

Placement

- Retirement stage of the pipeline
- This gives us an accurate view of the world
- No repair, recovery mechanisms a.la Return Address Stack [Skadron et. al]

Detection Mechanism

- If the call-return semantics have been violated → This could be an attack
- What about legitimate program constructs
- Setjmp()/Longjmp() also violate call return semantics
- Add some more state - like the Stack Pointer (the second entry)

Setjmp()/Longjmp()

- Setjmp() saves stack environment
- Useful for dealing with errors in low-level subroutines
- Longjmp() restores what was saved on the last call of setjmp()

Rewinddddddddddddddd

Return Address	Stack Pointer	Needs to be purged?
0x4002b8 calling dead_meat()	SP2	✓
0x4004a8 calling longjmp()	SP2	✓
caller of function()'s return address	SP0	

Recovery - Essential Exceptions

- Attack exception
- Underflow exception
- Overflow exception
- Context Switch exception ←
perhaps my nemesis

Evaluation

- Two parts
 - Functional Simulation - Does this truly detect attacks? Modified sim-fast simulator.
 - Timing Simulation - Design Issues need to be answered. SimpleScalar 3.0.

Timing Simulation

- Motivation == design details
- What design details?
 - Latching address and SP to pass down the pipeline
 - Modeling latencies incurred due to this detection check at the end
 - Modeling latencies due to rewinding in the case of `setjmp()/longjmp()`

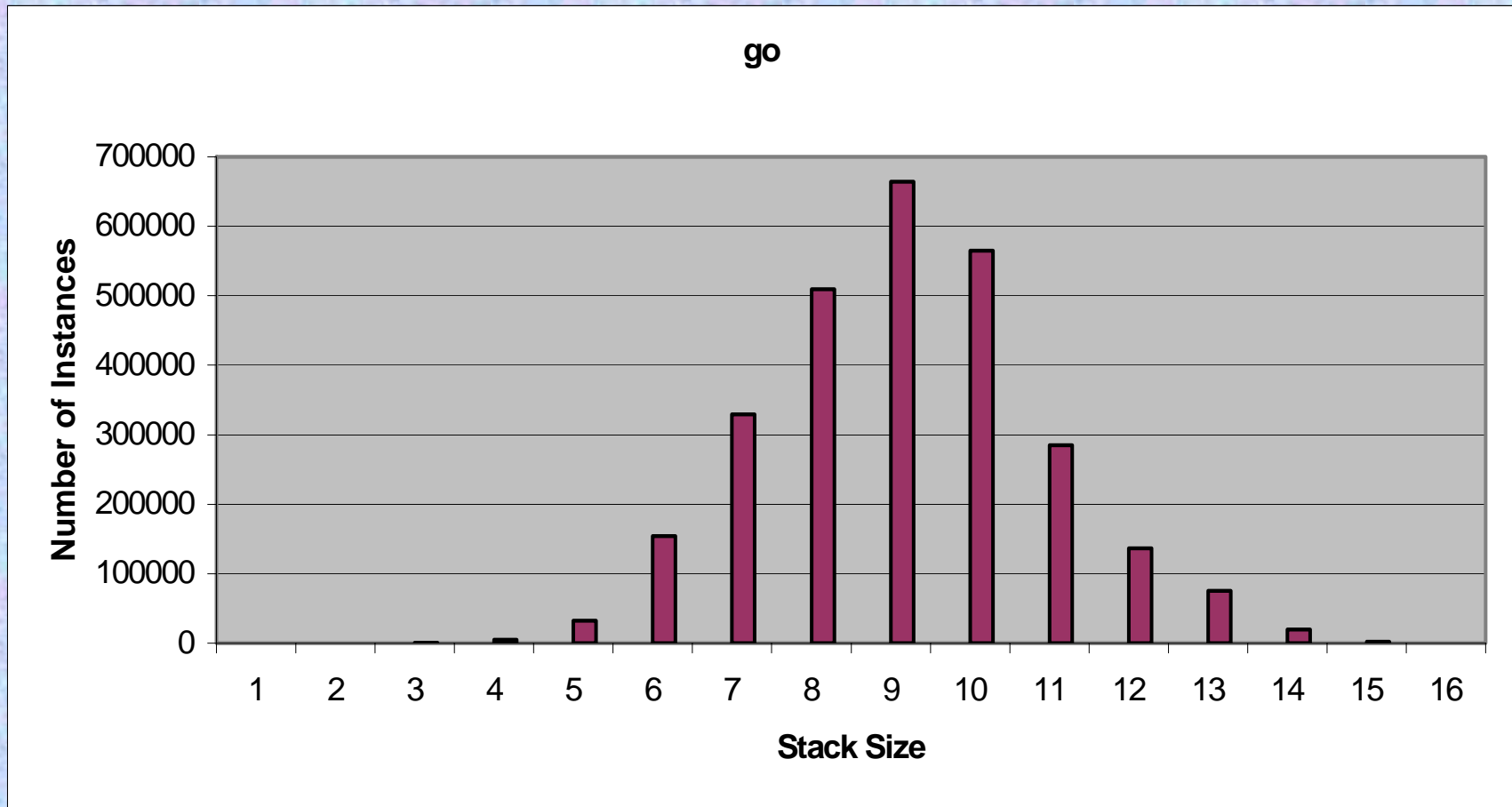
Timing Simulation

- Ran buggy programs - accurate detection
- Ran li - finished correctly
- Ran gcc, go, parser and jpeg to measure performance impact due to the shadow stack

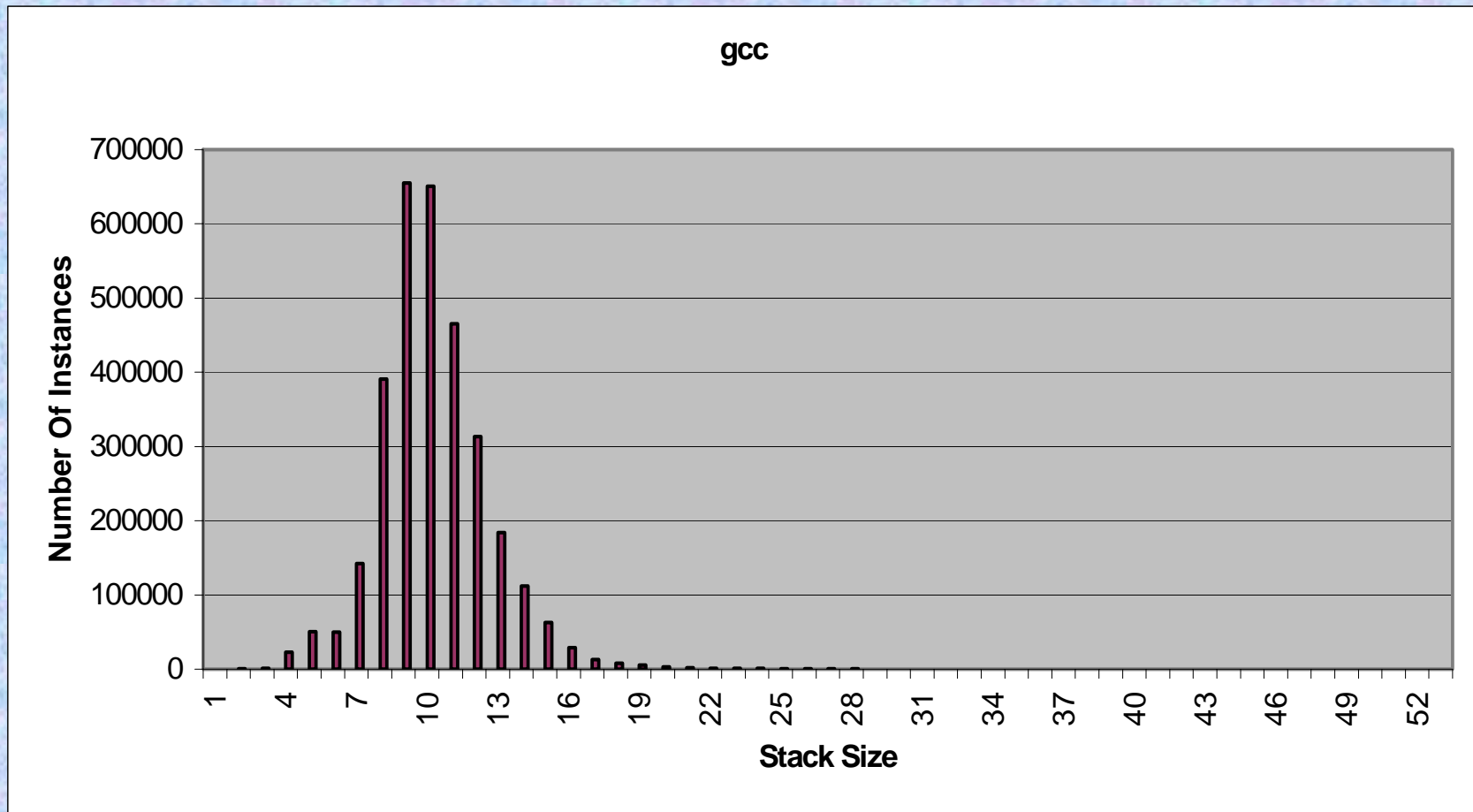
Results

Benchmark	Most Popular stack size	Maximum Stack size observed
li	37	90
gcc	8	52
go	8	14
parser	6	16
jpeg	5	62

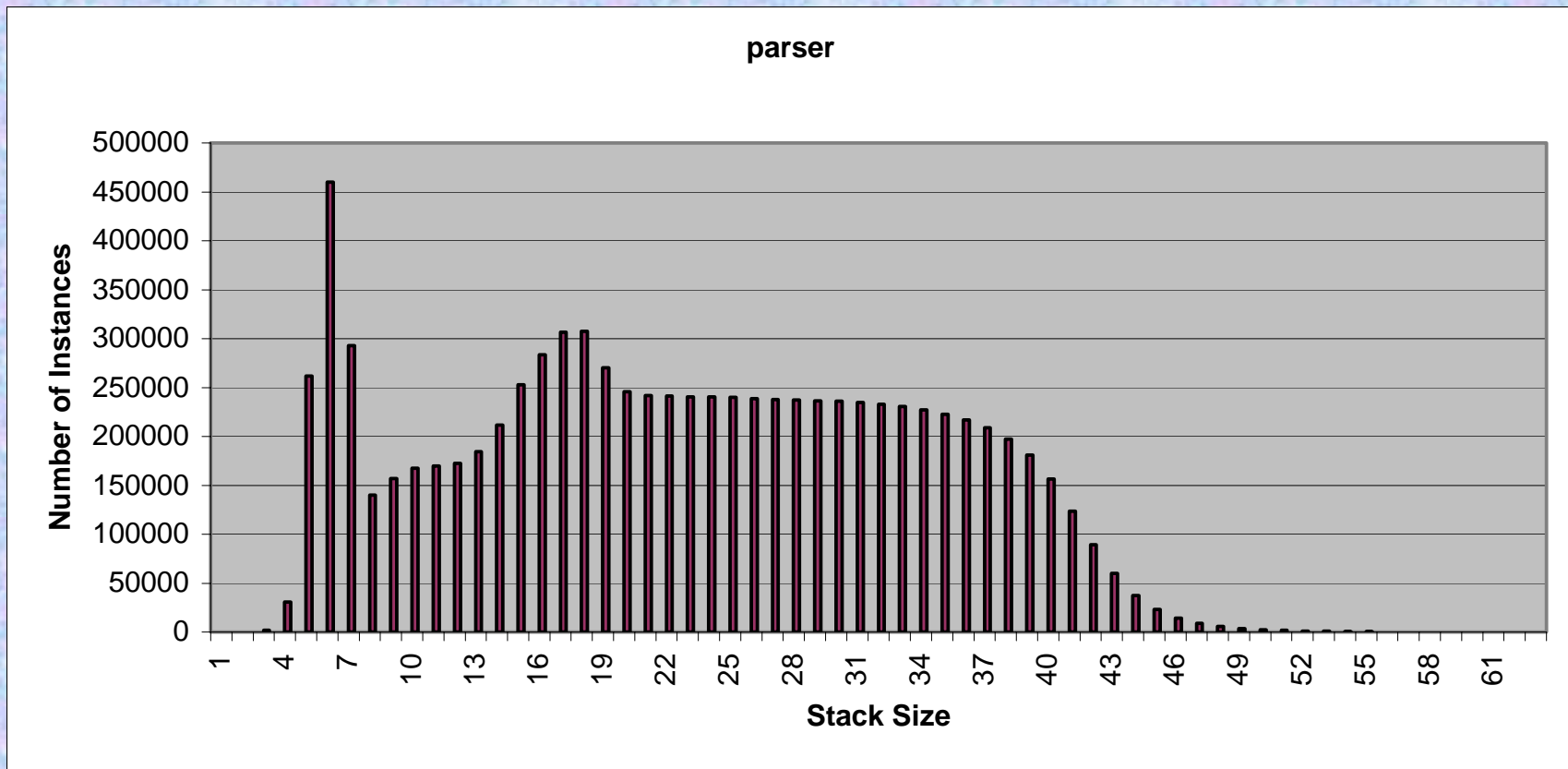
Stack Size Histograms - go



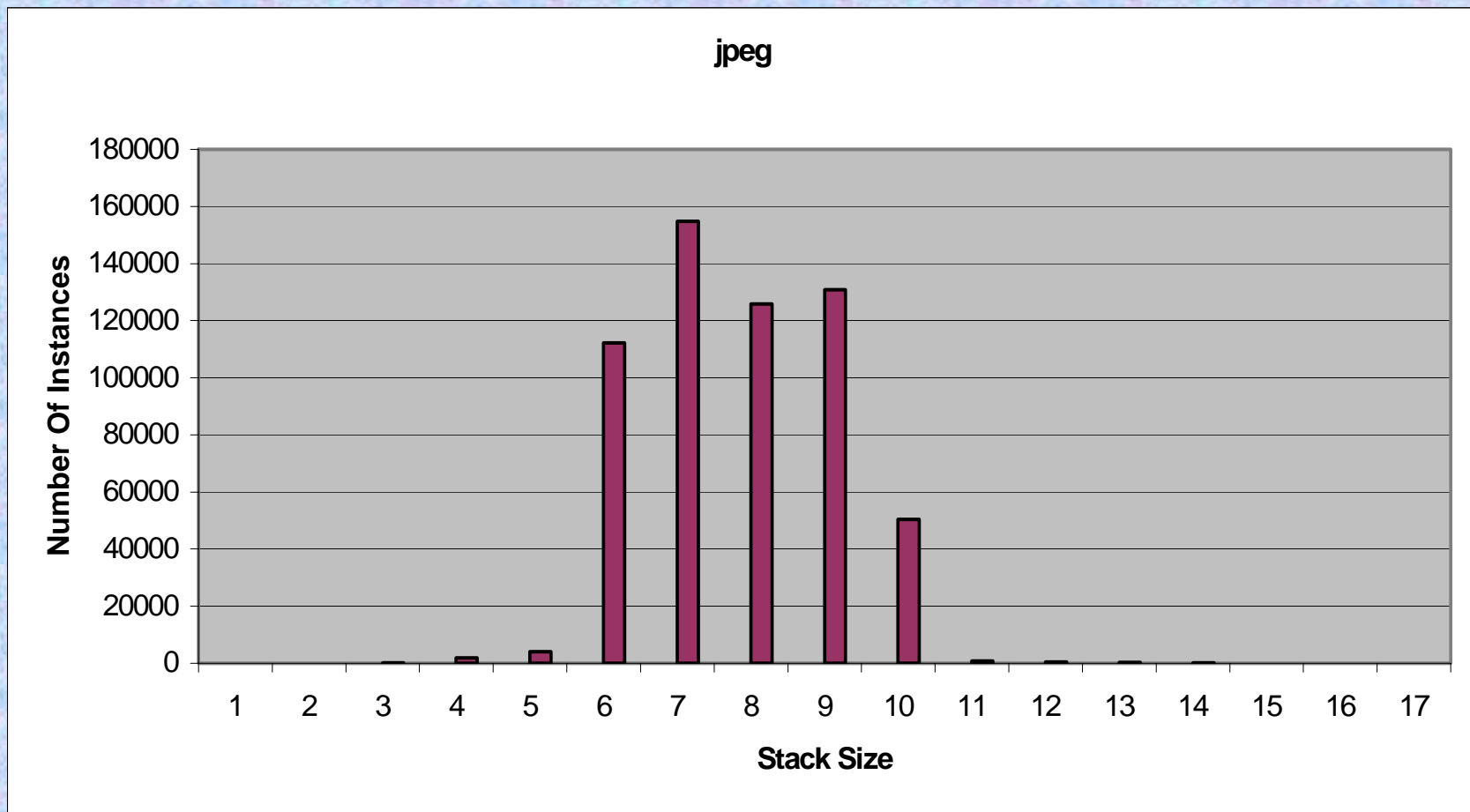
Stack Size Histograms - gcc



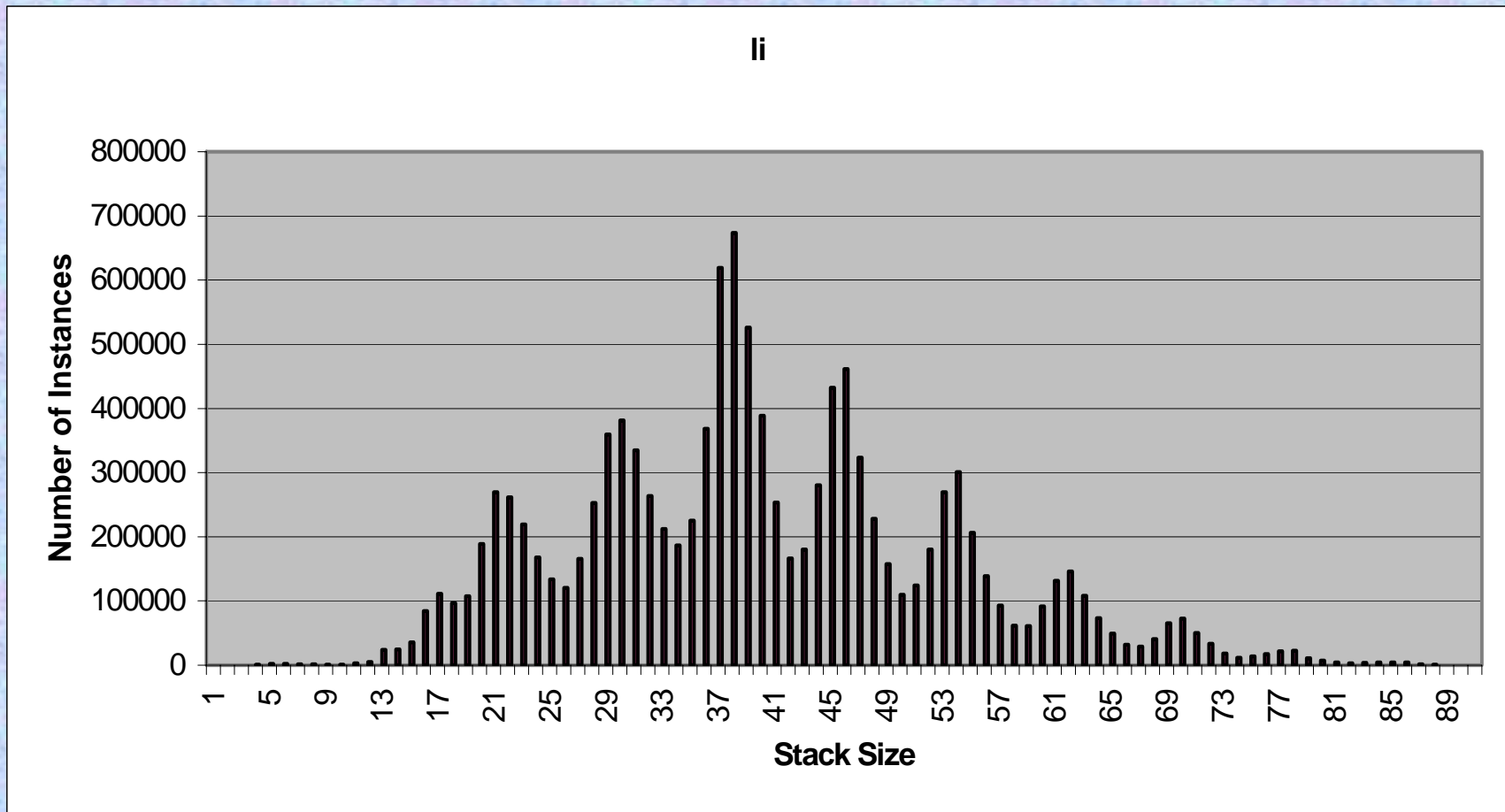
Stack Size Histograms - parser



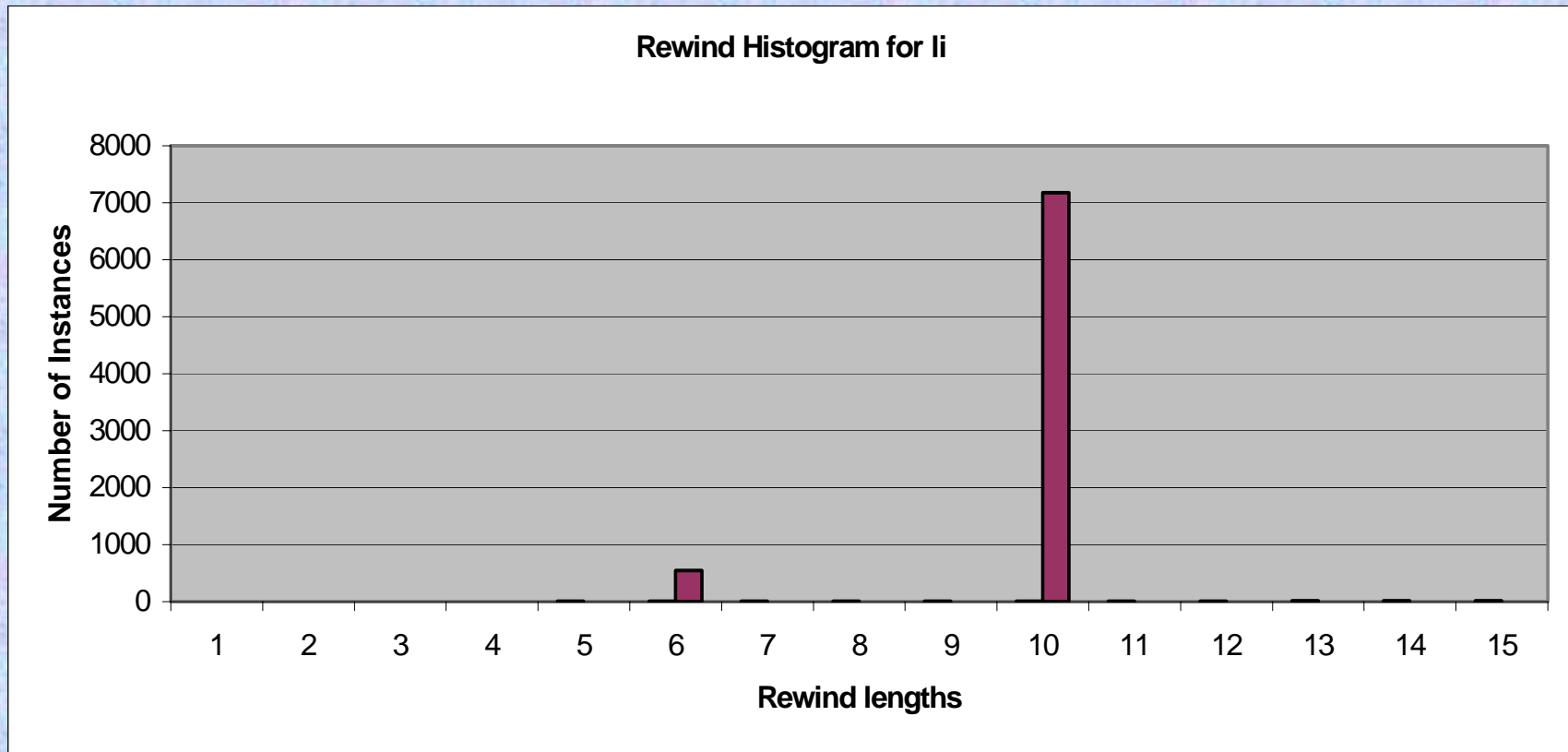
Stack Size Histograms - jpeg



Stack Size Histograms - li



Rewind Length Histogram - li



Performance Penalty

Benchmark	Original IPC	Modified IPC w/Shadow Stack	% Performance Degradation
li	1.4912	1.3702	12.1
gcc	0.9103	0.8877	2.26
go	0.9162	0.8990	1.72
parser	1.5549	1.4621	9.28
jpeg	2.0959	2.0807	1.52

Practicality Matters

- Scope of legitimacy
 - What about interpreters?
 - What about *other* program constructs?
 - What about context switching?
 - The holy grail is to deal with all this - The way to do it would be to figure out what other state we can record.

Practicality Matters

- What workloads **SHOULD** we be looking at?
 - Buggy daemons
 - More complex code like apache etc.
 - Cant compile these for simplescalar
 - Need full-system simulators/simulations

What have I done?

- Proposed a new solution to stack-smashing attacks *due to* buffer overflows
-- PLDI, SRF 2002.
- Evaluated designing such a stack
- Implemented a stack in a superscalar processor simulator
- Run some sizing and performance experiments
-- Masters Thesis, NCSU, 2002

Proposal vs. Reality -- Summary

- Generality is a big deal
- The shadow stack is a practical and feasible mechanism to detect stack smashing attacks
- The shadow stack is transparent to applications
- Need OS intervention however to implement exceptions

Future Work

- Numbers show that the stack does not grow too much
- Is this true for most vulnerable daemons and applications?
- Evaluate more realistic workloads
- Full system simulation to model recovery in more detail

Architecture Research

- ~~*ILD*~~
- ~~*Poseidon*~~
- DataScalar
- CMP Synchronization

DataScalar Architectures [Fall 2002]

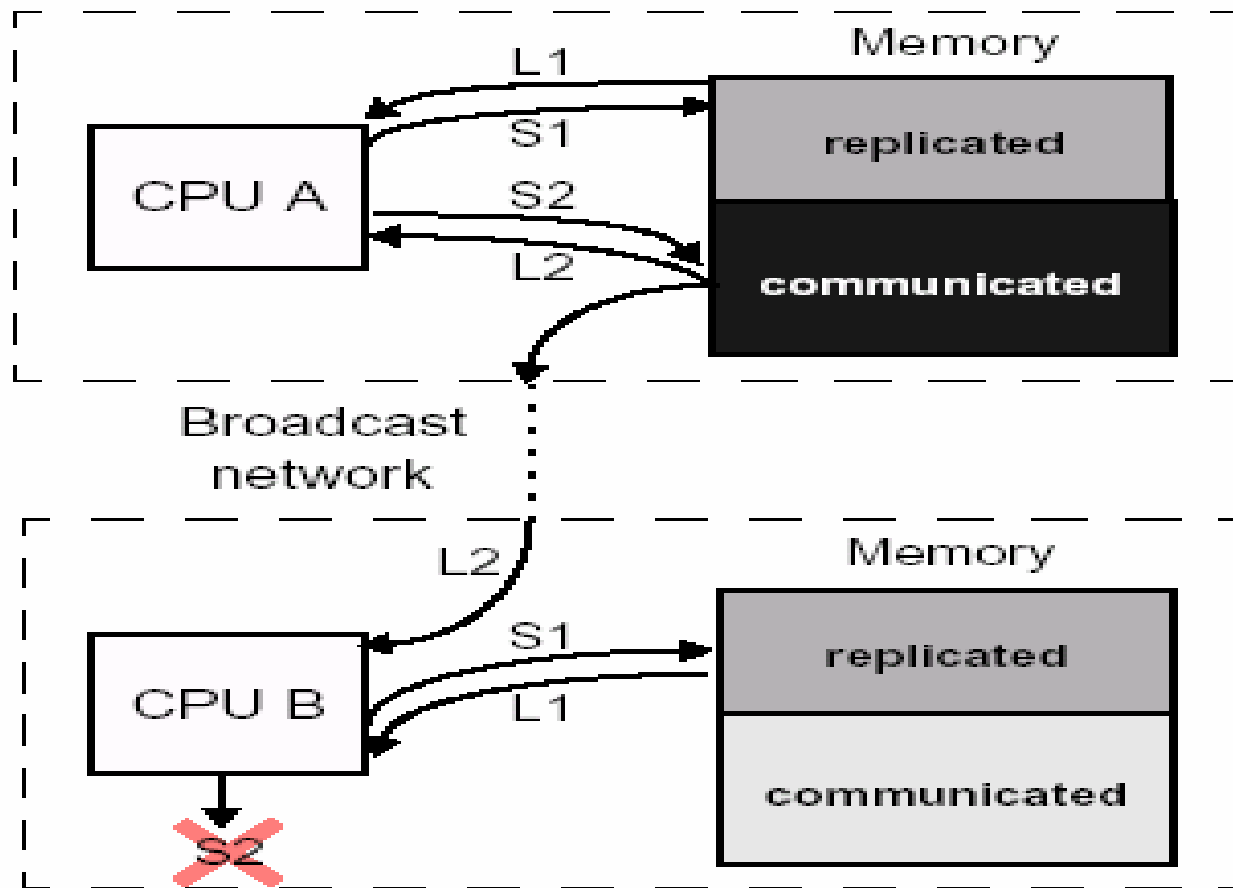
*Hard work never killed anybody, but why
take a chance?*

Charlie McCarthy

Lazy Broadcasting

- Hypothesis - Cache correspondence can be pushed further for speed
- Related Work
 - IBM RP3
 - Massive Memory Machine (MMM)
- Motivation - Memory Level parallelism to reduce processor-memory gap
- My contribution - Design and implementation of Lazy B

DataScalar Architectures (Burger, Goodman ISCA '98)



DataScalar Architectures

- All processing elements run the *SAME* binary
- Overall execution time is smaller because of redundant processing units working on disparate data-sets
- Data can belong to either replicated or communicated memory
- Two kinds of replication - Static and Dynamic

Two kinds of replication

- Static - hot pages form part of the replicated memory, then round-robin across processors.
- Dynamic - replicate data across processors *on the fly* (caching)
- Fold the decision of what to replicate dynamically into the L1 cache
- => If a miss in L1, *broadcast* (to remain correspondent)

Another Idea..

- Don't broadcast unless you *have to*.
- So who decides what data is to be dynamically replicated?
 - The lazyB predictor!
- Mark some loads as (to-be-broadcast-immediately) while others, we can be lazy about.
- What??

Analyze this

- Big picture, an application like PCA
- Smaller step, calculating several eigenvalues/eigenvectors
- Even smaller step, calculate several determinants of symmetric matrices
- At the lowest level, Gaussian elimination

An example

- One of the many steps in Gaussian Elimination --
Converting the matrix to triangular form

Example. Solve the following system via Gaussian elimination

$$\begin{cases} 2x - 3y - z + 2w + 3v = 4 \\ 4x - 4y - z + 4w + 11v = 4 \\ 2x - 5y - 2z + 2w - v = 9 \\ \quad 2y + z + 4v = -5 \end{cases}$$

The augmented matrix is

$$\left(\begin{array}{ccccc|c} 2 & -3 & -1 & 2 & 3 & 4 \\ 4 & -4 & -1 & 4 & 11 & 4 \\ 2 & -5 & -2 & 2 & -1 & 9 \\ 0 & 2 & 1 & 0 & 4 & -5 \end{array} \right).$$

We use elementary row operations to transform this matrix into a triangular one. We keep the first row and use it to produce all zeros elsewhere in the first column. We have

$$\left(\begin{array}{ccccc|c} 2 & -3 & -1 & 2 & 3 & 4 \\ 0 & 2 & 1 & 0 & 5 & -4 \\ 0 & -2 & -1 & 0 & -4 & 5 \\ 0 & 2 & 1 & 0 & 4 & -5 \end{array} \right).$$

Which looks like

```

:
:
for (i=1;i<4;i++)
{
    for (j=0;j<5;j++)
    {
        line: 4 A[i][j] = A[i][j] - (A[0][j]*x);
    }
}
:
:
```

Division of Labor

- If we had a two processor data scalar infrastructure:
 - P1 had rows 1 and 2 in communicated memory
 - P2 has rows 2 and 3 in communicated memory.
 - P1 also has the load_value that is loaded into r0

In machine instructions

```
    ld r0← i
cont:  cmp rj← i,4
        blt rj,end
        ld r0←j
cont2: cmp rj←j,5
        blt rj,end2
        ld A[i][j]
<< continue
    w/calculation
    in line 4 >>
        jump cont2
end2:  jump cont
end:   NOP
```

```
    ld r0← i
cont:  cmp rj← i,4
        blt rj,end
        ld r0←j
cont2: cmp rj←j,5
        blt rj,end2
        ld A[i][j]
<< continue
    w/calculation
    in line 4 >>
        jump cont2
end2:  jump cont
end:   NOP
```

First two iterations on P1

Next two on P2

In machine instructions

```
    ld r0 ← i
cont:  cmp rj ← i,4
        blt rj,end
        ld r0 ← j
cont2: cmp rj ← j,5
        blt rj,end2
        ld A[i][j]
<< continue
    w/calculation
    in line 4 >>
        jump cont2
end2:  jump cont
end:   NOP
```

```
    ld r0 ← i
cont:  cmp rj ← i,4
        blt rj,end
        ld r0 ← j
cont2: cmp rj ← j,5
        blt rj,end2
        ld A[i][j]
<< continue
    w/calculation
    in line 4 >>
        jump cont2
end2:  jump cont
end:   NOP
```

Next two on P2

What do we broadcast ?

- Immediately, the loads that affect branch decisions
- Lazily, the loads that are part of the communicated memory that are not needed until a certain time t .
- In this example, the values of the first row are not broadcast till the 2nd iteration completes

Where do we save?

- P1 computes iteration 1, iteration 2, broadcasts the contents of $A[0][\text{all}]$ at the end of iteration 2
- P2 does not compute anything in iteration 1 and iteration 2
- Receives the broadcast and then crunches iterations 3 and 4

Motivation for LazyB

- Parallelize the parts of the program that are hard to analyze.
- Maybe a program has certain phases that can be parallelized by letting a processor crunch data locally, while other processors do not process that computation.
- => collapsing the pipeline on other processors for a portion of the program

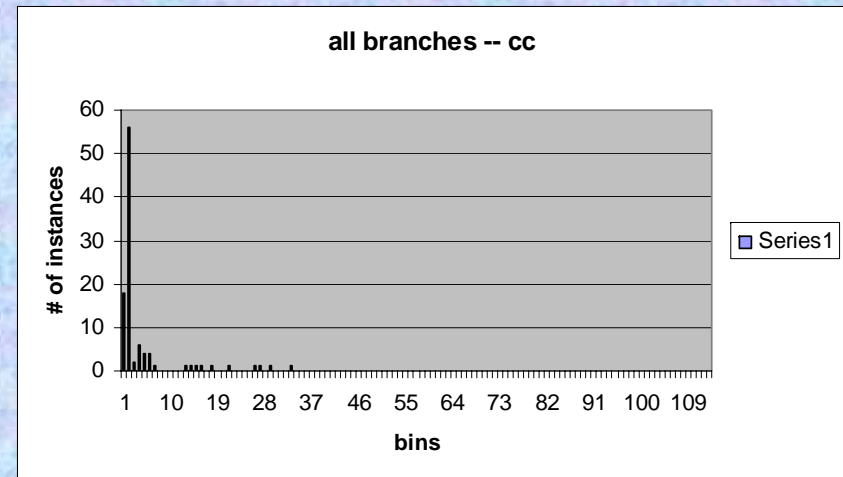
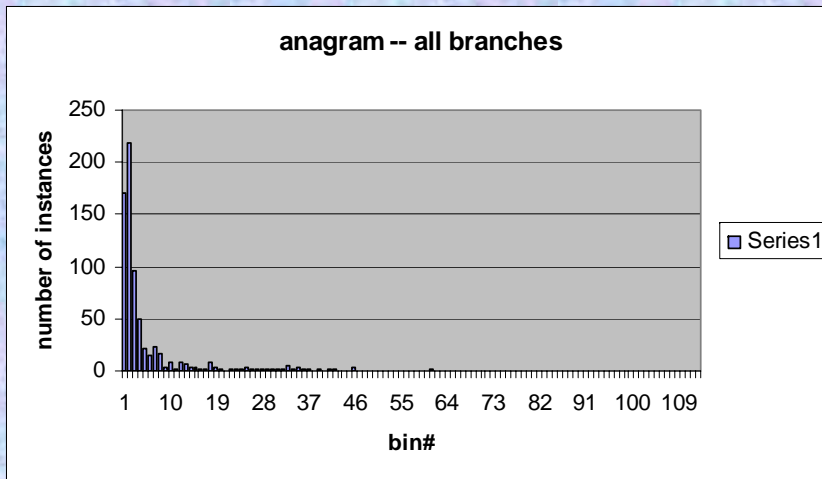
Candidates for LazyB

- Non-Branching loads
 - Loads that do not affect a branch decision
- Non-Loopy Loads
 - Loads that do not affect loop condition variables

Loads that affect Branches

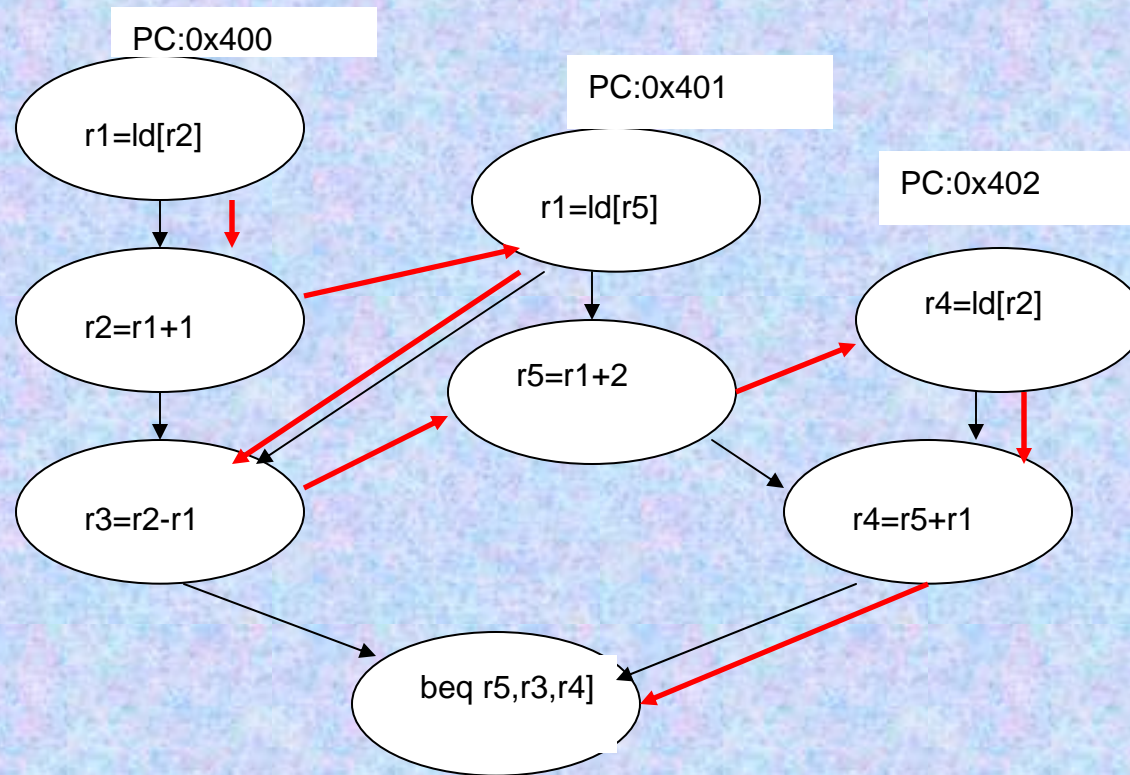
- Intuition: If a particular load contributes a value that is used, possibly down a chain of instructions, that end up in a branch, we need to broadcast that load.
- Experiments:
 - How many loads affect all conditional branches?
 - How many loads affect looping branches?

Limit Study - more runs coming soon!



Note: less than 1% of load instructions affect branch decisions

Algorithm

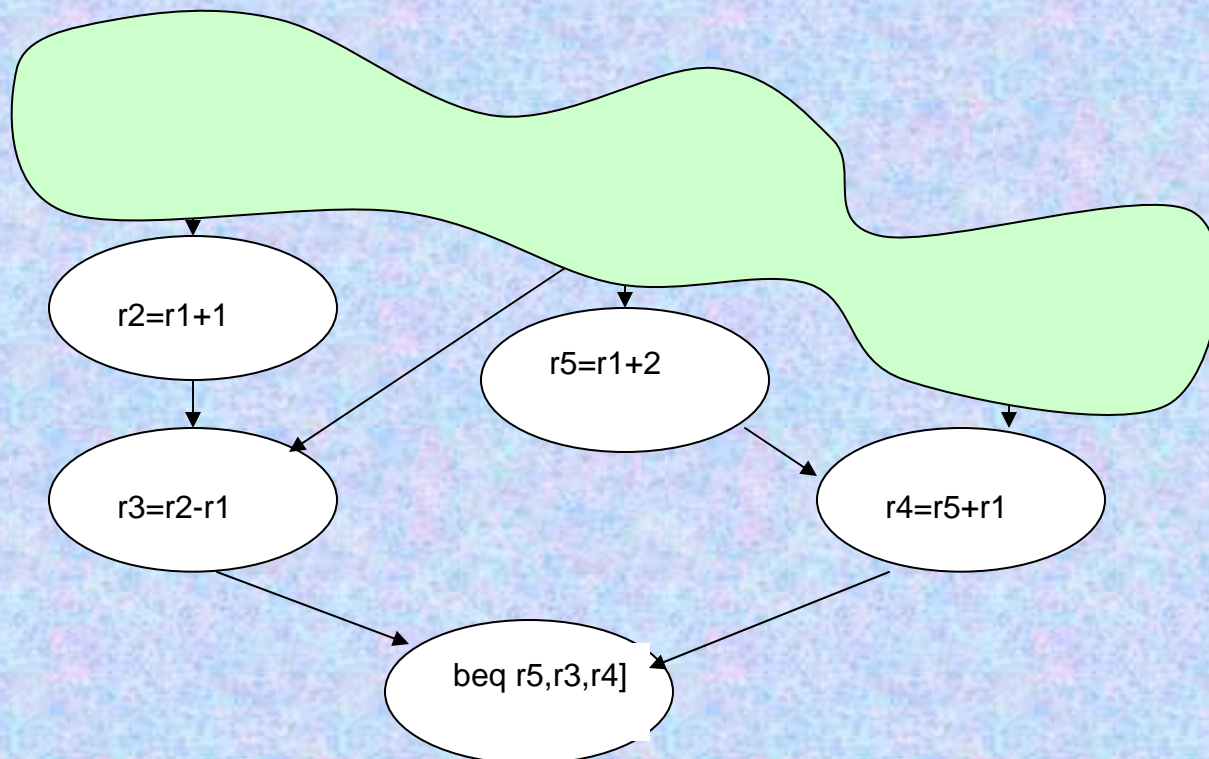


Algorithm Contd.

Code Snippet		R0	R1	R2	R3	R4	R5
r1=ld[r2]	T0		0x400				
r2=r1+1	T1		0x400	0x400			
r1=ld[r2]	T2		0x401	0x400			
r3=r2-r1	T3		0x401	0x400	0x400 0x401		
r5=r1+2	T4		0x401	0x400	0x400 0x401		0x401
r1=ld[r3]	T5		0x403	0x400	0x400 0x401		0x401
r4=r5+r1	T6			0x400	0x400 0x401	0x401 0x403	0x401
beqr5,r3,r4	T7						

Algorithm contd.

- For the branch instruction, we are interested in the loads at the periphery.



Hysteresis

- This is problematic
- Not only do we need to know how “soon” a load to a register was re-written, but also, the sequence of instructions between the two writes to the same register.
- Plus, if the first instance of the load contributes to a branch, no way that you can, not broadcast it! - This may not work outside of loop code.

Passing the Buck

- Need a forward view of the world for an *implementable* design.
- Profile to understand, verify.
- Build to implement!
- Steal ideas from the critical path predictor (ISCA 2001)

Architecture Research

- ~~*ILD*~~
- ~~*Poseidon*~~
- ~~*DataScalar*~~
- CMP Synchronization

CMP Synchronization

- **Hypothesis**- Spin-locks are an attractive mutex mechanism for CMPs
- **Related Work**
 - CMP research is rich!
- **Motivation** - good synchronization primitives for CMPs
- **My contribution** - Team effort, combed Solaris code, hacked Simics suite (please refer to back-up slides)

Systems Research

- **Gray Box Systems [User-Mode Linux]**
- IBM Spring 03
- K42 Scheduler
- CFor - IDS

Gray Box Systems

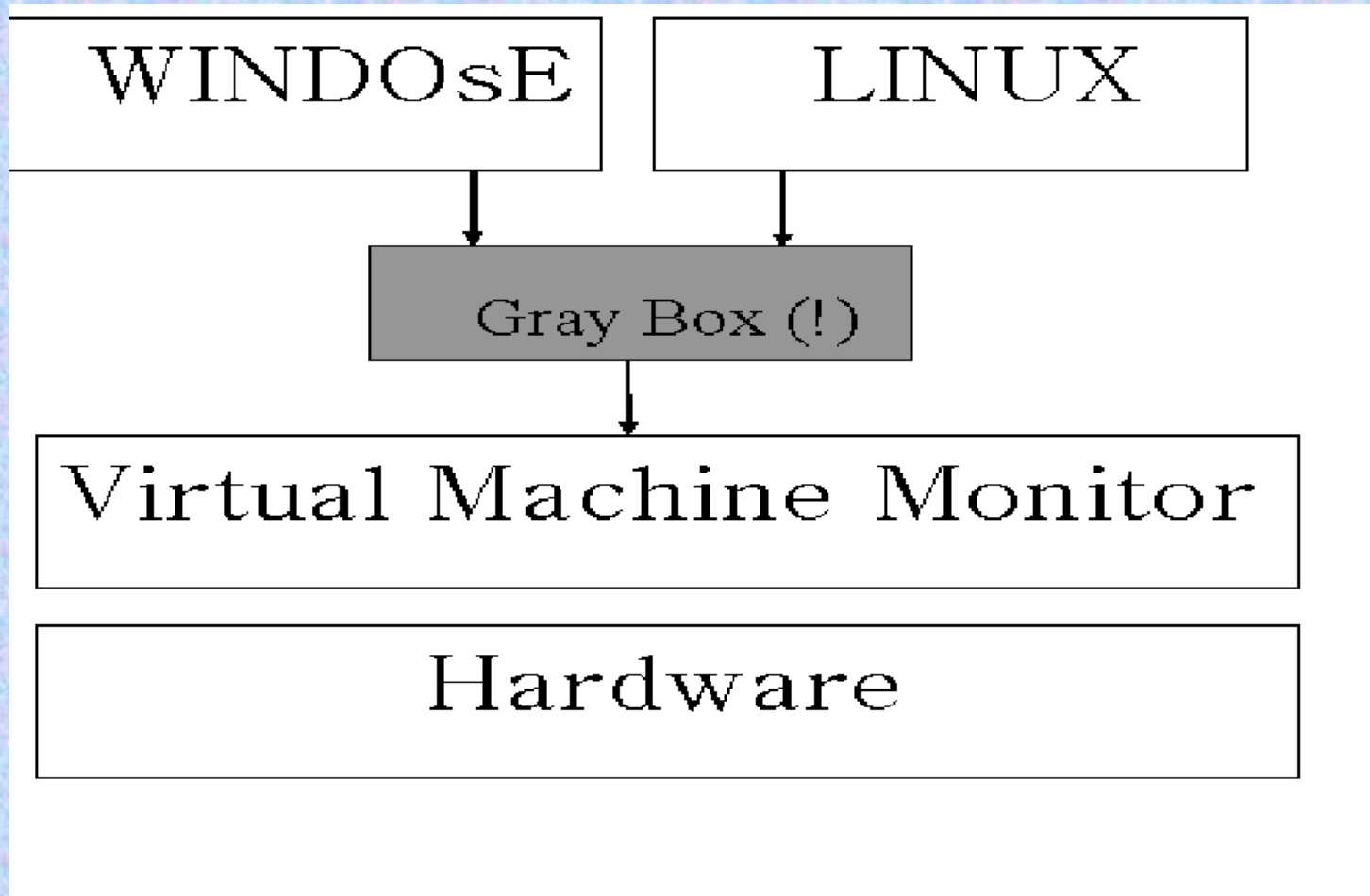
Motivation

- Virtual Machines might catch on
- Virtual Machine Monitor needs to overcome many limitations
- Better Virtual Machine Design

Approach

- Gray-box Virtual Machine policies
- This would give us the following information:
 - VM Scheduling Policies
 - VM memory requirements
- Feed these as inputs/hints to the VMM (so that it can better allocate resources)

Big Picture



Cast & Crew

- Need an environment with Virtual Machines running
 - Virtualized resources
 - Many users of the same resources
 - An arbitrator for those resources
- Need an understanding of what these resources are

Star of the Show

- User-Mode Linux
- We get a VM that can have more resources than the physical machine seems to have.
- Disk storage is a single file on the physical machine
- Host scheduler implements UML scheduler decisions

Preliminary Results

- Got a version of User Mode Linux running
- Identified what the Virtual Machine vs. the Virtual Machine Monitor was in this environment
- Identified idle loop in VM
- Identified inputs and outputs of the Gray Box

Background

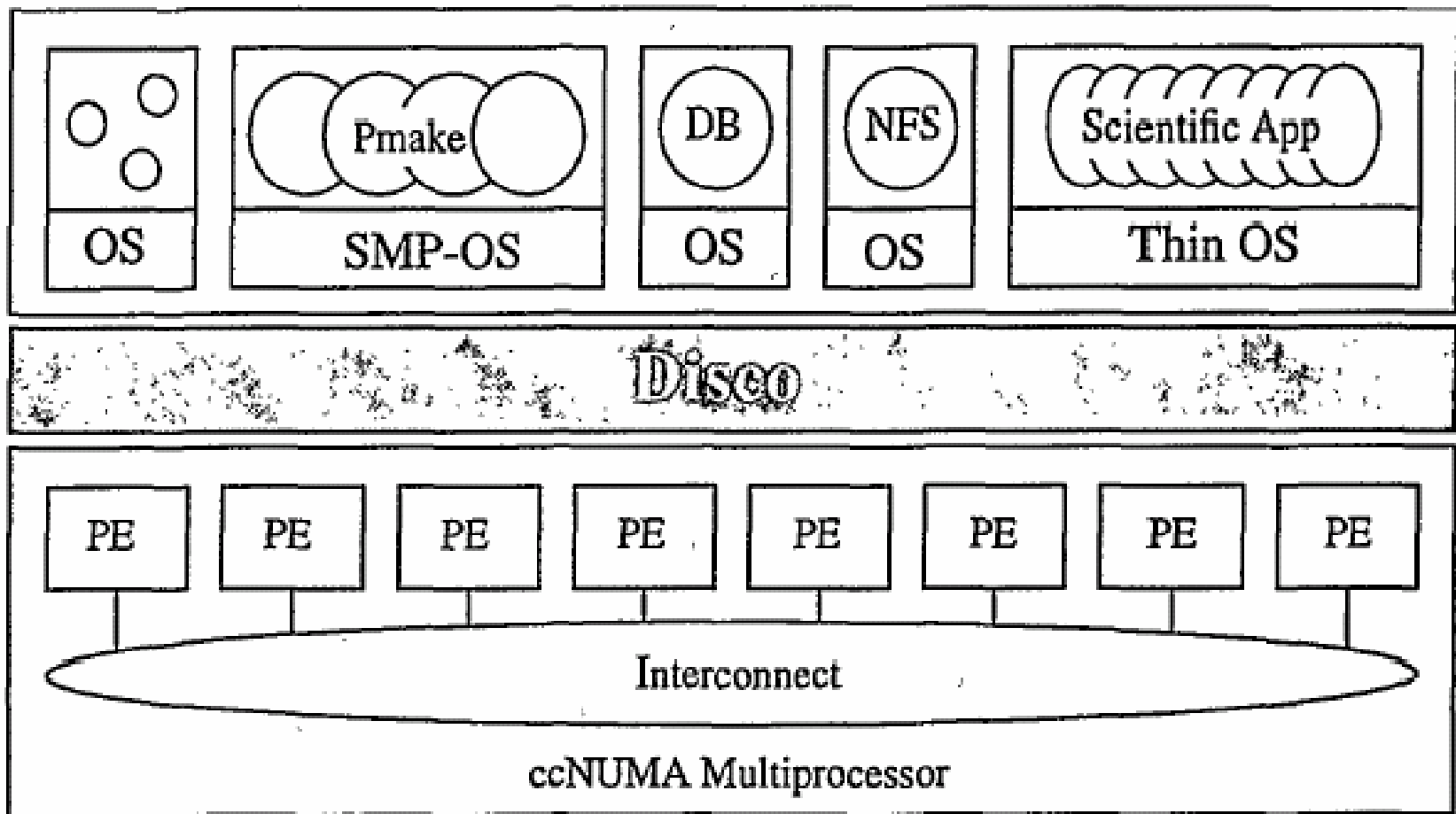
- Common Theme of OS research: *extensibility*
- Many ideas, one that was particularly successful
- DISCO, that went on to become VMware
- Used Virtual Machines



Background

- Run Commodity Operating Systems
- Virtualize Resources
 - Hardware
 - Memory
 - Network Interface
- Common Pool of resources, VMM (or DISCO) arbitrates control of those resources

DISCO



Problems

- Overhead of Virtualization
 - Hardware Emulation
 - Exception Handling
- VMM is myopic
 - Resource Allocation is a challenge with limited information
 - Memory
 - Processing Power
- Inter-VM Communication

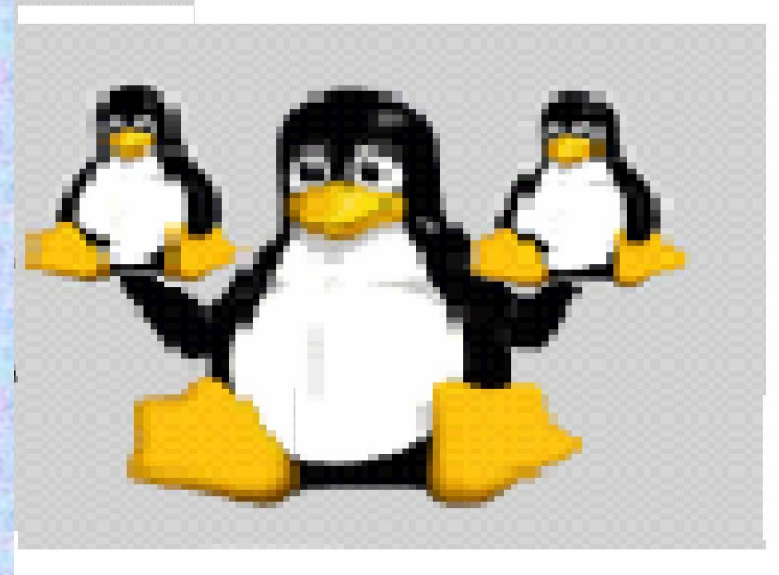


Methodology

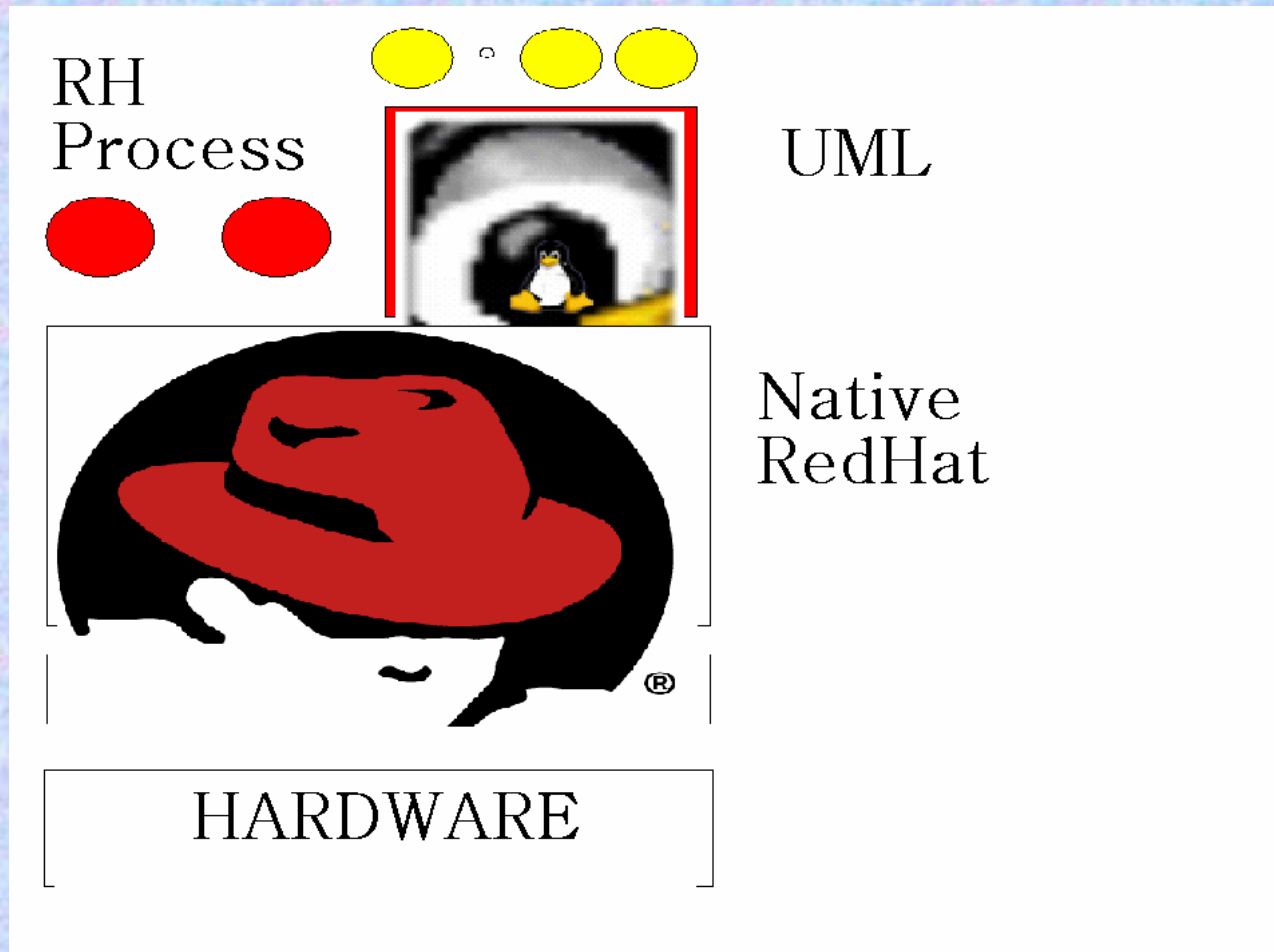
- What the heck is User Mode Linux?
- Who is the Virtual Machine Monitor?
- *When* is idle?
- What are the inputs/outputs of the gray box?
- Show me the money

UML

- Used UML to simulate Virtual Machine environment
- A kernel that runs in user-mode
- Brain-child of Jeff Dike
- Originally conceived to debug kernels
- Now, even used for web-hosting



Abracadabra!



UML

- The UML kernel is a port to the original kernel to the syscall interface
- Linux Kernel - hardware specific code like drivers
- No emulation of user-space code (processes run like they would normally)
- Emulation of kernel-space code

UML

- Two ways for processes to get into kernel code:
 - Syscall → ptrace()
 - Trap → deal with it using signals
- VM Initialization
 - Modify idle_thread to call the kernel_start module
 - Permanent tracer thread to keep track of processes

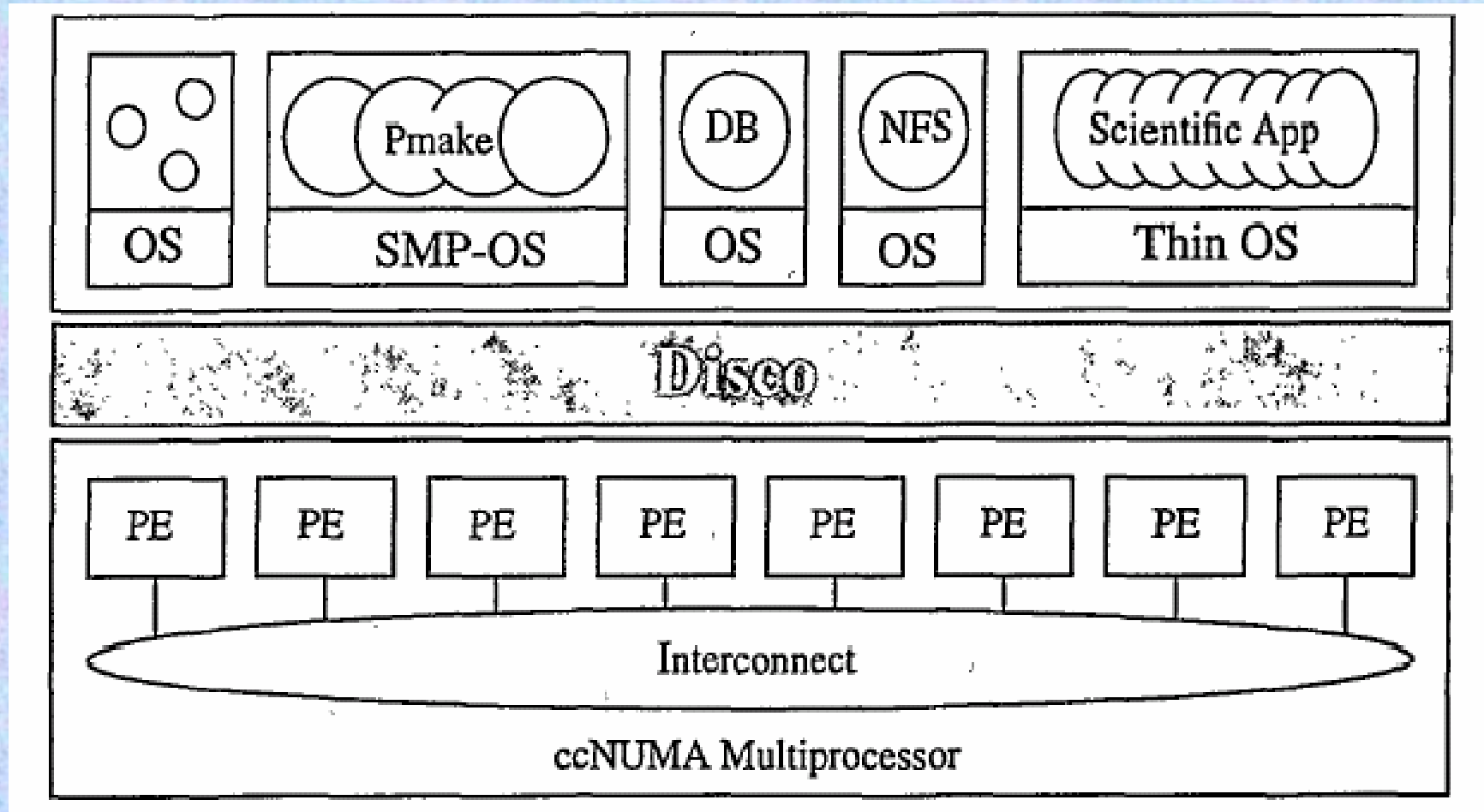
Million \$\$ Question

- Who is the VMM?
 - "basic design decision is that this port will directly run the host's unmodified user space."
 - System calls are implemented in the virtual kernel
 - Device Drivers/Hardware: constructed from software abstractions provided by the host

The line is not clear

- Virtualization of processor → Native Kernel (or nothing is done)
- Virtualization of system calls → UML port
- Memory → Single large file so, where is the virtualization?
- If you ask me, it's the host that's the VMM

Compare this to DISCO



Idle-ness

- Simple definition - when the UML kernel is executing the `do_idle()` loop
 - Problem: too fine-grained
- More complicated definition - when the tracing thread has no children?
 - This might imply all processes are sleeping OR there ARE no processes
- Third definition - more quantitative prediction, how *long* are processes going to be idle?

Demo - Idle when do_idle()

- Switch to `crash.cs.wisc.edu`!

The Gray-Box

Inputs from the VM

- I'm idling()
- I'm done using page x
- I might sleep for n-seconds



Inputs to the VMM

- VM_n is idling
- So many pages can be added to the global pool of available pages
- So many time-slots may be available for others

SMTM

- A snapshot of top: Only Linux

```
18746 root R 32.5 0.8 1:43 linux
12392 root R 32.1 0.1 194:23 linux
10567 root R 31.3 0.7 194:24 linux
15468 root R 2.1 4.8 0:16 kdeinit
  980 root S 0.9 6.8 0:18 X
18702 root R 0.5 0.9 0:07 top
   1 root S 0.0 0.0 0:36 init
   2 root SW 0.0 0.0 0:00 keventd
```

SMTM

- A snapshot of top: Linux and one instance of UML

```
18746 root R 31.5 0.8 4:08 linux
10567 root R 31.1 0.7 196:48 linux
12392 root R 30.9 0.1 196:48 linux
18702 root R 1.7 0.9 0:13 top
15468 root S 0.9 4.9 0:24 kdeinit
  980 root S 0.7 6.8 0:21 X
19779 root S 0.7 1.5 0:02 linux1
 1143 root S 0.3 2.0 0:01 kdeinit
   8 root SW 0.1 0.0 0:03 kupdated
19783 root T 0.1 3.6 0:01 linux1
19789 root T 0.1 2.1 0:00 linux1
20631 root T 0.1 3.1 0:01 linux1
20924 root T 0.1 3.0 0:00 linux1
   1 root S 0.0 0.0 0:36 init
```

SMTM

A snapshot of top: Linux and two instances of UML

```
12392 root      RW    25.5  0.1 197:45 linux
10567 root      R     25.3  0.3 197:46 linux
18746 root      R     25.1  0.7   5:05 linux
22143 root      T     5.0   3.2   0:00 linux2
  980 root      S     2.0   6.8   0:22 X
18702 root      R     1.5   0.9   0:16 top
15468 root      S     1.3   3.5   0:26 kdeinit
21224 root      T     0.9   2.5   0:01 linux2
21718 root      S     0.5   0.3   0:02 linux2
21721 root      S     0.3   1.1   0:01 linux2
  1148 root      S     0.1   2.6   0:03 kdeinit
19779 root      S     0.1   1.4   0:03 linux1
19789 root      T     0.1   1.7   0:00 linux1
20924 root      T     0.1   1.6   0:00 linux1
21725 root      T     0.1   1.3   0:00 linux2
```

SMTM

Now: One kernel (#2) is executing a really busy while loop –

```
22243 root      R      23.8  2.8    0:06 linux2
10567 root      R      23.6  0.3   198:04 linux
18746 root      R      23.6  0.7    5:23 linux
12392 root      RW     23.2  0.1   198:04 linux
18702 root      R      1.7   0.9    0:17 top
21224 root      T      0.7   2.5    0:02 linux2
20924 root      T      0.5   1.6    0:01 linux1
  980 root      S      0.3   6.8    0:23 X
15468 root      S      0.3   3.6    0:28 kdeinit
19779 root      S      0.1   1.4    0:03 linux1
19786 root      T      0.1   2.0    0:00 linux1
19789 root      T      0.1   1.7    0:00 linux1
21348 root      T      0.1   8.0    0:14 linux2
```

Conclusions

- These are domain-specific definitions (of where the line between the VM and the VMM is)
- Still need a clear understanding of what might be interesting resources to virtualize
- Need to define the extent of information conveyed to the Gray Box

More Conclusions..

- More specifically (lets talk about scheduling)
 - We need to agree on what *idle* means in this instance.
- Generality vs. Specificity - is this going to work in non-Linux environments?

Future Work

- Write a user-tunable priority system?
- Right now, processes are run natively
- Should we change this? So that all VMs get equal priority? ← hmm..
- Suggestions?

Systems Research

- ~~User-Mode Linux~~
- IBM Spring 03
- K42 Scheduler
- CFor - IDS

IBM Spring 03

- Built a hypervisor
 - Please ask questions! (NDA?)

Systems Research

- ~~User-Mode Linux~~
- ~~IBM Spring 03~~
- K42 Scheduler
- CFor - IDS

K42 Scheduler

- Idea - Implement clean Scheduler Activations interface
- Related Work
 - Scheduler Activations, Anderson et. al.
 - Exokernel, MIT
- Motivation - Dispatcher Default is very heavyweight
- My contribution - Implemented clean SA interface, cleaned up DispatcherDefault

SA Calls

- User → Kernel
 - Give me another processor
 - I'm idle (take my processor away for another process)
- Kernel → Kernel Scheduler
 - This processor can be re-assigned to common pool
 - Notify user-level that this SA has blocked

SA Calls

- Kernel → Kernel Scheduler
 - SA has unblocked, notify user-level and add thread to the readyQ
 - A processor belonging to the SA has been preempted, callback
- K42's story:
 - Pre-emption was a challenge

Seperation

- Work in progress..
 - Please ask questions!

Cfor - IDS

- Idea - Yet Another Sandboxing System
- Related Work
 - Osita, Janus
 - Consh, MAPBox
 - Static Analysis
- My contribution - Cleaned up a really bad code-base.. Research Question is in implementation

Cfor - IDS

- Host-Based IDS
- Methodology:
 - Ptrace() system calls and intercept
 - Build on old methods to interpose on /proc
- Components:
 - GUI, Engine, Library (cford)

Cfor - IDS

- Engine:
 - Rule Mangers (per process)
 - Virtual Process
 - Server to intercept and alter program state
- Funding ended project-life
 - Please ask questions - lots of open problems for example MT programs and state interception.

Method?? To all the madness!!

- Hypothesis - Knowing the stack bottom up is very important..
 - Cannot be "just a" systems guy or an architect
- Related Work
 - Richard Hamming "You and your research"
- Motivation - To be a "rainbow" -multi-lingual researcher
- My contribution - Taking the path less travelled

Conclusions

- A "good systems programmer" needs to write a File System (distributed of course!)
 - Throw all the abstractions in.. I/O, hardware, software, modular programming, networking into a washing machine, and out comes a clean FS.
 - Project Kali - my final dissertation?

Thanks!!

Backup Slides

CMP Talk - Outline

- Problem Overview
- Mutex Characterization
- Lock Arbiter Optimization
- Conclusions

Problem

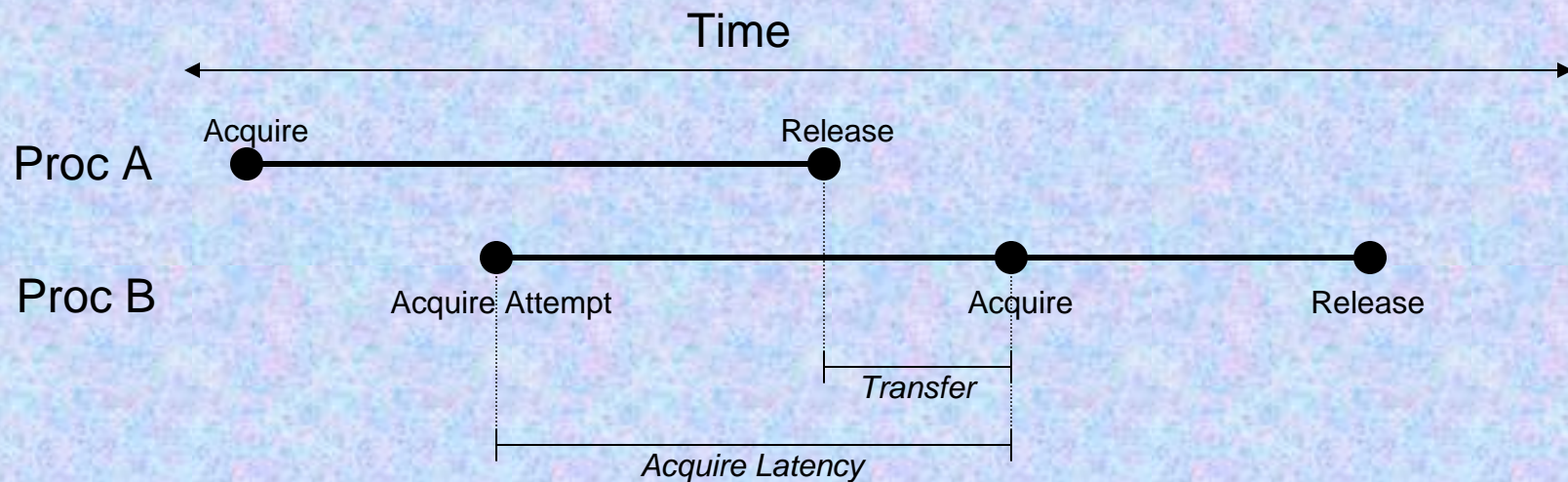
- Cost of Synchronization in a CMP (Focus on Mutex)
 - Waiting on Contention
 - Lock Transfer Mechanism
 - Characterize latency and bandwidth of above
- Maintaining Mutual Exclusion
 - Spin-locks - *simple, fast if uncontended*
 - Queue-based locks - *better if contended, fair*
 - Can we expect significant difference in a CMP context?
 - Can we get benefits of both with no SW changes?

Expectations

- Hypotheses:
 - Low cost (latency and bw) for on-chip transfers
 - Little contention for locks
 - Relatively course-grained applications for this study
 - Little execution time spent on locks
- *Spin-locks are an attractive mutex mechanism for CMPs*
- **Granularity** refers to amount of computation between synchronization points

Example

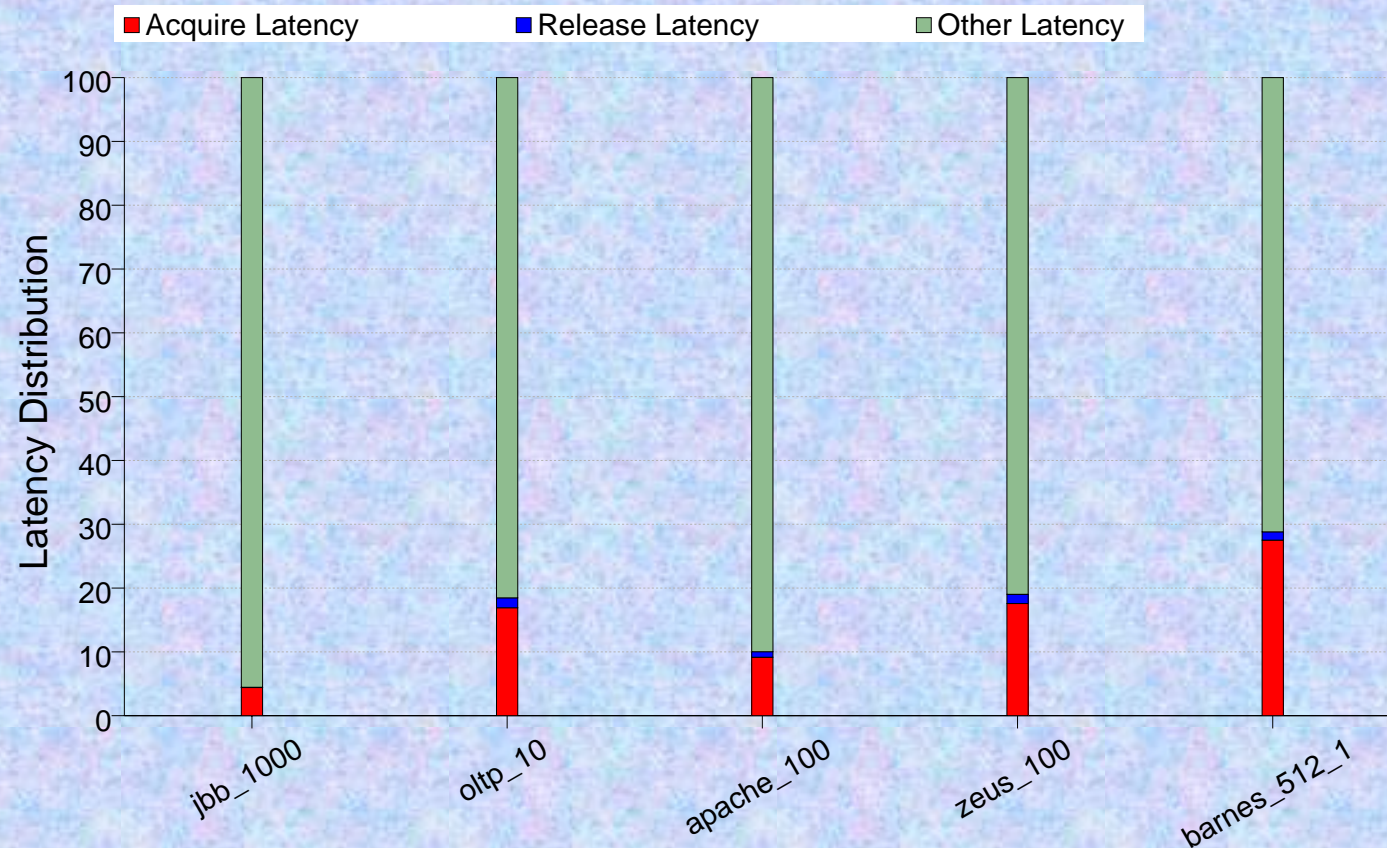
- $\text{Acquire latency} = \text{acquire time} - \text{first acquire attempt}$
- $\text{Transfer latency} = \text{acquire time} - \text{release time}$
 - Includes both contention time and transfer time
 - Instruction execution time also included



Inferring Mutual Exclusion

- SPARC provides atomic read/write primitives used to construct synchronization (test&set, swap, compare&swap,)
- For addresses which have seen an atomic operation:
 - Atomic op (e.g. test&set) returning zero assumed *acquire*
 - Atomic op returning non-zero assumed *failed acquire*
 - Store writing zero assumed *release*
 - Load on lock un-held by this proc assumed precursor to acquire attempt
(i.e. *test of test and test&set*)
- Verified these assumptions for many locks in Solaris kernel
- Inferences don't fit all observed uses of these atomics, tried to remove these addresses from stats
- Context switches are handled

Latency of Mutual Exclusion

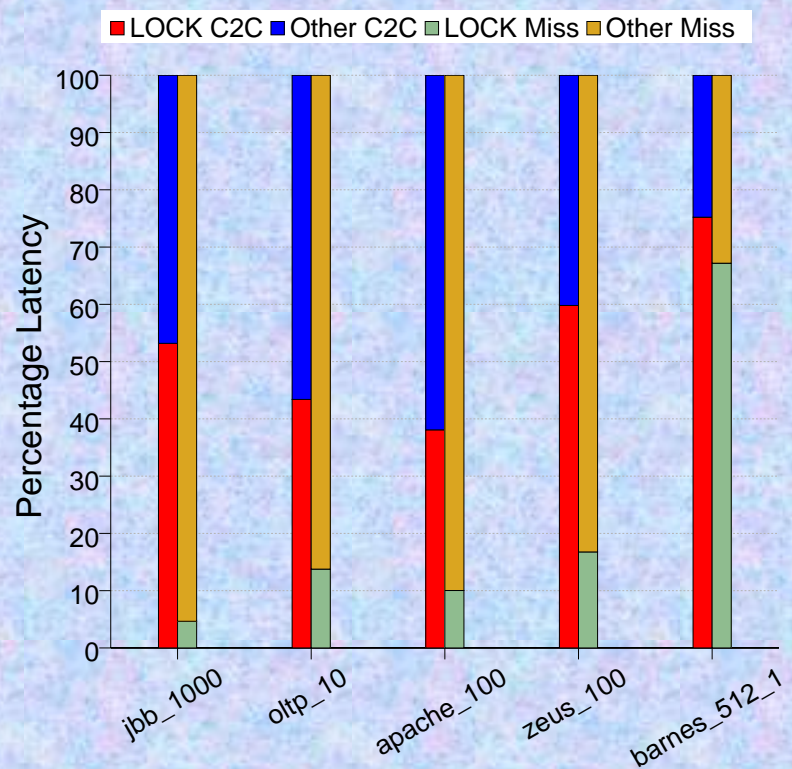
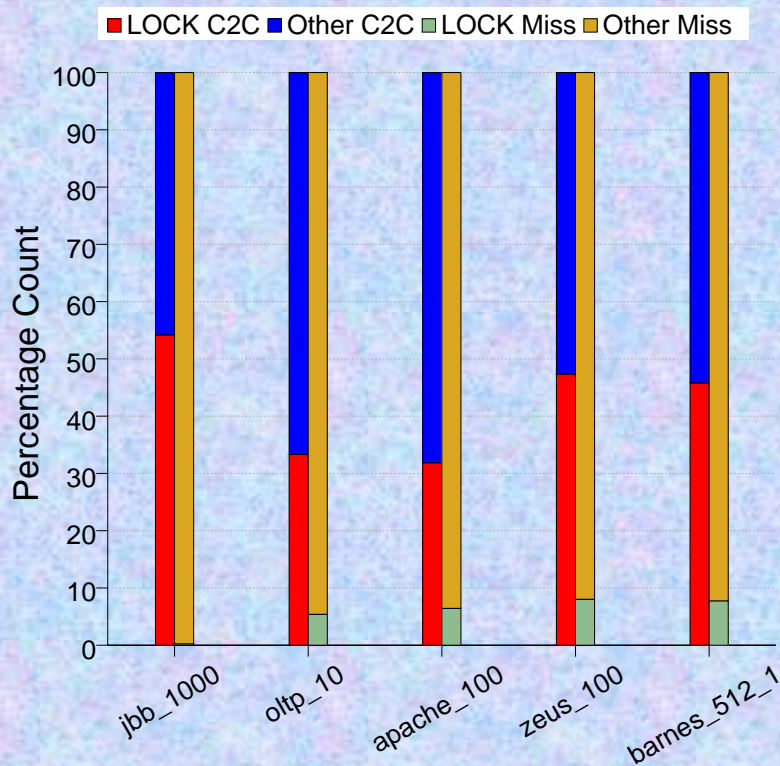


- Workloads spend a significant amount of time waiting on locks

Cache To Cache Transfers

Number of L1 Misses

Latency of L1 Misses

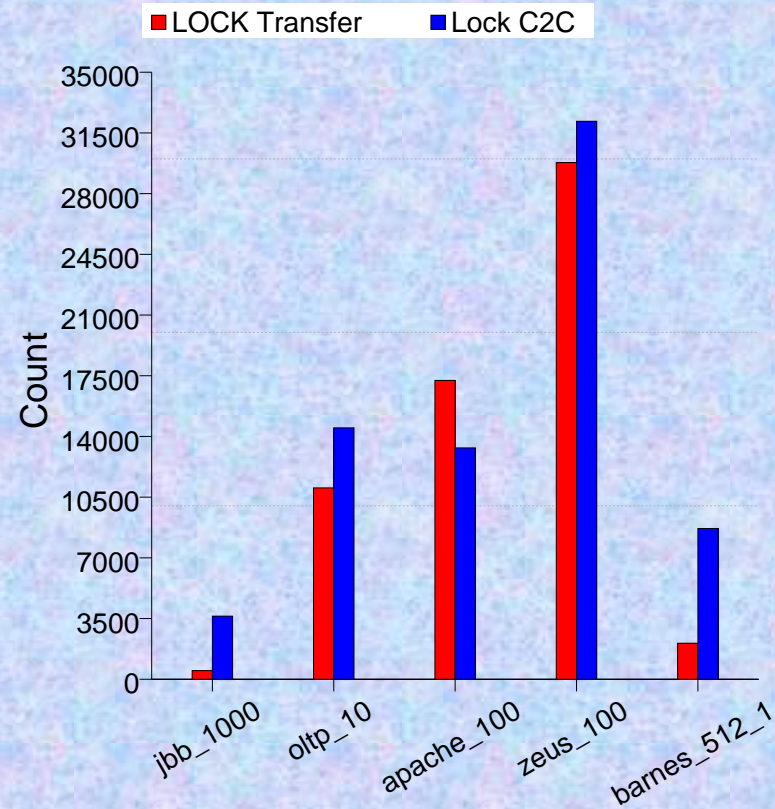
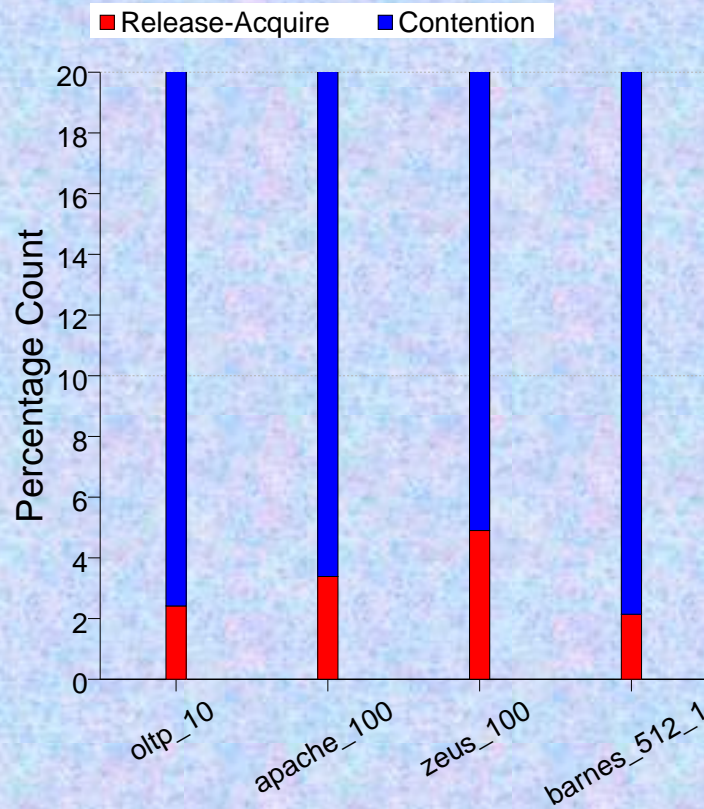


- Locks constitute a large fraction of cache to cache transfers

On-Chip Lock Arbiter

- Dynamically turn spin-locks into queue locks
 - Separate coherence protocol for cache blocks predicted to contain locks
 - Blocks not cached by L1s
- Arbiter Intercepts acquire/release operations for each lock
 - Queue acquire for a held lock (mem op doesn't complete)
 - On observed release, first queued acquire is allowed access to block (queued mem op completes)
 - Timeout necessary to avoid possible deadlock and starvation
- Benefits
 - Reduces bandwidth for contended locks
 - Provides fairness
- Probably only desirable for "hot" locks

Lock Arbiter Potential



- Expect minimal direct latency impact because of low transfer time
- More bandwidth impact

Conclusions

- Findings
 - Locks significant part of cache to cache transfers
 - Few hotly contended locks, but account for surprising amount of execution time and bandwidth
- Spin locks fine for most locks (for these workloads)
 - Lock arbiter shows some potential for reducing bandwidth
- Importance of synchronization cost goes up for finer granularity, so a different mechanism may be more beneficial
 - Hypothesis: *CMPs enable finer grained parallelism*
 - Need to try fine-grained applications

Outline

- ~~*Motivation*~~
- ~~*Approach and Preliminary Results*~~
- Background
- Methodology
- Conclusions
- Future Work