



**2006**  
**BARC**  
**Boston area ARChitecture**

*Proceedings of the  
Fourth Annual Boston-Area Architecture Workshop*

*University of Rhode Island  
Kingston, RI, USA  
February 3, 2006*

<http://www.ele.uri.edu/barc2006>



*Sponsored by:*



# Fourth Annual Boston-Area Architecture Workshop

## **Workshop Organizers (General Chairs):**

Gus Uht, University of Rhode Island  
Resit Sendag, University of Rhode Island

## **Registration Chair:**

Qing Yang, University of Rhode Island

## **Publications Chair:**

Yan Sun, University of Rhode Island

## **Webmaster:**

Gus Uht, University of Rhode Island  
(Website after that of BARC 2005 at Brown University.)

## **Steering Committee:**

Krste Asanovic, MIT  
R. Iris Bahar, Brown University  
David Kaeli, Northeastern University  
C. Andras Moritz, University of Massachusetts, Amherst  
Shubu Mukherjee, Intel

## **Program Committee:**

Gus Uht, University of Rhode Island, Chair  
R. Iris Bahar, Brown University  
David Brooks, Harvard University  
Martin Herbordt, Boston University  
C. Andras Moritz, University of Massachusetts, Amherst  
Shubu Mukherjee, Intel Corp.  
Chandrakant Patel, HP Labs  
Resit Sendag, University of Rhode Island  
Zhijie Shi, University of Connecticut  
Berk Sunar, Worcester Polytechnic Institute  
Richard Weiss, Evergreen State College, Washington  
Xinping Zhu, Northeastern University

---

## **Acknowledgments**

We are extremely grateful to AMD Corp. and Intel Corp. for sponsoring the Workshop, making it all possible.

Prof. Qing Yang and Liping W. Yang put in long hours automating the online registration process. Prof. Yan Sun lent her invaluable experience to Workshop planning and operations. The URI graduate student 'volunteers' Tim Parys, Jin Ren, Weijun Xiao, Yafei Yang and Ayse Yilmazer worked long and hard to make the final preparations and operation successful.

The Steering Committee provided much good counsel, squashing our bad ideas and encouraging the good ones.

The Program Committee worked within an incredibly short turnaround time: abstract submissions to author notifications in just five days. We are especially grateful to Program Committee member Prof. David Brooks of Harvard University; at the last minute he readily took on the task of managing the reviews of the URI submissions.

The URI University Club staff bent over backwards to accommodate an ever-increasing number of participants.

----Gus Uht & Resit Sendag, BARC 2006 Organizers

---

Abstracts and slides Copyright (C) 2005, 2006, and other years, by their respective authors and owners.  
Trademarks and other logos Copyright (C) by their respective owners.  
Other Proceeding's content Copyright (C) 2005, 2006, by Augustus K. Uht.

# Program



## Fourth Annual Boston Area Architecture Workshop

University of Rhode Island  
Kingston, RI, USA

February 3, 2006

---

*10:00am-10:25am: Continental Breakfast*

---

*10:25am-10:30am: Welcome by Dean Bahram Nassersharif, URI College of Engineering, & the Wkshp. Organizers*

---

*10:30am-11:30am: Session 1, Temperature, Power and Energy  
(Session Chair: David Kaeli, Northeastern University)*

- |   |              |
|---|--------------|
| <b>Fast Transient Thermal Simulation Based on Linear System Theory,</b><br>Yongkui Han, Israel Koren and C. Mani Krishna, UMass, Amherst                  | <i>p. 5</i>  |
| <b>Impact of Process Variations on Low Power Cache Design,</b><br>Mahmoud Bennaser and Csaba Andras Moritz, UMass, Amherst                                | <i>p. 10</i> |
| <b>Compiler-Based Adaptive Fetch Throttling for Energy-Efficiency,</b><br>Huaping Wang, Yao Guo, Israel Koren and C. Mani Krishna, UMass, Amherst         | <i>p. 16</i> |
| <b>Energy-Aware Microprocessor Synchronization: Transactional Memory vs. Locks,</b><br>Tali Moreshet, R. Iris Bahar and Maurice Herlihy, Brown University | <i>p. 21</i> |
- 

*11:30am-11:45am: Short Break*

---

*11:45am-12:45pm: Session 2, Architectures and Performance  
(Session Chair: C. Andras Moritz, University of Massachusetts, Amherst)*

- |  |              |
|--|--------------|
| <b>Investigating the Effects of Wrong-Path Memory References in Shared-Memory Multiprocessor Systems,</b><br>Ayse Yilmazer, URI; Resit Sendag, URI;<br>Joshua J. Yi, Freescale Semiconductor; and Augustus K. Uht, URI | <i>p. 27</i> |
| <b>Using R-STAGE to Optimize the DASAT Cache System,</b><br>M. Tyler Maxwell, Charles C. Weems, J. Eliot B. Moss and Robert B. Moll, UMass, Amherst  | <i>p. 33</i> |
| <b>Exploring Architectural Challenges in Scalable Underwater Wireless Sensor Networks,</b><br>Zhijie Jerry Shi and Yunsi Fei, University of Connecticut  | <i>p. 38</i> |
| <b>ILP is Dead, Long Live IPC!,</b><br>Augustus K. Uht, URI  | <i>p. 43</i> |
- 

*12:45pm-1:30pm: Sandwich Buffet Lunch*

---

---

1:30pm-2:30pm: *Keypanel Session, 'Whence Goeth the Microprocessor?' (Moderator: Gus Uht, URI)* p. 47

**Constituencies and Keypanelists:**

- \* *Academia:* Prof. Anant Agarwal, Professor of Electrical Engineering and Computer Science, MIT.
  - \* *End Users:* Dr. Atul Chhabra, Enterprise Architect and Senior IT Manager, Verizon, Inc.
  - \* *Industry:* Dr. Joel Emer, Fellow, Intel Corp.
  - \* *Everybody:* The Audience, various positions, many institutions.
- 

2:30pm-3:00pm: *Long Break*

---

3:00pm-4:00pm: *Session 3, Simulation and Design*  
(*Session Chair: Qing Yang, University of Rhode Island*)

- Branch Trace Compression for Snapshot-Based Simulation,** p. 49  
Kenneth C. Barr and Krste Asanovic, MIT
  - Requirements for any HPC/FPGA Application Development Tool Flow** p. 55  
(**that gets more than a small fraction of potential performance**),  
Tom Van Court and Martin C. Herbordt, Boston University
  - Accelerating Architectural Exploration Using Canonical Instruction Segments,** p. 60  
Rose F. Liu and Krste Asanovic, MIT
  - Tortola: Addressing Tomorrow's Computing Challenges through Hardware/Software Symbiosis,** p. 66  
Kim Hazelwood, University of Virginia and Intel Massachusetts
- 

4:00pm-4:15pm: *Short Break*

---

4:15pm-5:15pm: *Session 4, Dependability (Session Chair: R. Iris Bahar, Brown University)*

- Parity Replication in IP-Network Storages,** p. 72  
Weijun Xiao, Jin Ren and Qing Yang, URI
  - Software-based Failure Detection in Programmable Network Interfaces,** p. 77  
Yizheng Zhou, Vijay Lakamraju, Israel Koren and C.M. Krishna, UMass, Amherst
  - Software Fault Detection Using Dynamic Instrumentation,** p. 82  
George A. Reis and David I. August, Princeton University; and  
Robert Cohn and Shubhendu S. Mukherjee, Intel Massachusetts
  - Self-healing Nanoscale Architectures on 2-D Nano-fabrics,** p. 88  
Teng Wang, Mahmoud Ben Naser, Yao Guo and Csaba Andras Moritz, UMass, Amherst
- 

5:15pm-5:45pm: *Long Break*

---

5:45pm-6:30pm: *Session 5, Emerging Concepts (Session Chair: Martin Herbordt, Boston University)*

- Functional Programming in Embedded System Design,** p. 93  
Al Strelzoff, E-TrolZ, Inc.
  - The Fresh Breeze Memory Hierarchy,** p. 100  
Jack B. Dennis, MIT
  - Ideal and Resistive Nanowire Decoders,** p. 106  
Eric Rachlin and John E. Savage, Brown University
- 

6:30pm-6:35pm: *Closing Remarks*

---

# Fast Transient Thermal Simulation Based on Linear System Theory

Yongkui Han, Israel Koren, and C. Mani Krishna  
*Department of Electrical and Computer Engineering*  
*University of Massachusetts, Amherst, MA 01003*  
*E-mail: {yhan,koren,krishna}@ecs.umass.edu*

## Abstract

As power density of microprocessors is increasing rapidly and resulting in high temperatures, thermal simulation becomes a necessity for CPU designs. Current thermal simulation methods are very useful, but are still inefficient when performing thermal analysis for long simulation times. In this paper, we propose a new transient thermal simulation method for CPU chips at the architecture level, which allows us to calculate transient temperatures on a chip over long simulation times. Based on a linear system formulation, our proposed method has the same accuracy as that of traditional thermal simulation tools and is orders of magnitude faster than previous algorithms. The time-consuming integration computations are replaced by simpler matrix multiplications in our method. Compared to the HotSpot simulator, our TILTS algorithm achieves speedups of 67 and 47 for the Pentium Pro and Alpha processors, respectively. With some additional memory space, our revised algorithm CTILTS is 268 and 217 times faster than the HotSpot simulator for the Pentium Pro and Alpha processors, respectively. A significant property of our method is that it does not incur any accuracy loss compared to the HotSpot simulator.

## 1 Linear System Theory Overview

A linear system is described by its state equation, where the state variables are system internal variables. Denote the number of state variables by  $N$ , the number of inputs to the system by  $M$ , and the state and input vectors by  $\mathbf{x}(t)$  and  $\mathbf{u}(t)$ , respectively:

$$\mathbf{x}(t) = (x_1(t), x_2(t), \dots, x_N(t))^T$$

$$\mathbf{u}(t) = (u_1(t), u_2(t), \dots, u_M(t))^T$$

The linear system equation is:

$$\dot{\mathbf{x}}(t) = \mathbf{F}\mathbf{x}(t) + \mathbf{G}\mathbf{u}(t) \quad (1)$$

where  $\mathbf{F}$  is an  $N \times N$  matrix, and  $\mathbf{G}$  is an  $N \times M$  matrix. These matrices are fixed for a time-invariant linear system.

For such a linear system, the complete response is the sum of the zero-input response and the zero-state response.

$$\mathbf{x}(t) = e^{\mathbf{F}t}\mathbf{x}(0) + \int_0^t e^{\mathbf{F}(t-\tau)}\mathbf{G}\mathbf{u}(\tau)d\tau \quad (2)$$

## 2 CPU Chip as a Linear System

A CPU chip is a thermal system that can be described by its equivalent thermal circuit composed of thermal resistors and capacitors. All these components are linear components, making it a linear system. The input to this linear system is the power dissipated by each functional unit on the chip, and its state variables are the temperatures of the internal nodes in the thermal circuit.

Let  $M$  be the number of functional units which dissipate power, and  $N$  the number of internal nodes in the thermal circuit ( $N \geq M$ ). Denote the thermal resistance between node  $i$  and  $j$  by  $r_{ij}$ , and the thermal capacitance to the thermal ground (the ambient environment) by  $c_i$  for node  $i$ . For convenience, let  $1/r_{ii} = 0$  in the following summation equation. The CPU thermal system obeys the following differential equation:

$$c_i \dot{x}_i(t) = - \sum_{j=1}^N \frac{1}{r_{ij}} (x_i(t) - x_j(t)) + \tilde{u}_i(t), i = 1, 2, \dots, N \quad (3)$$

where  $\mathbf{x}(t) = (x_1(t), \dots, x_N(t))^T$  is the temperature vector and  $\tilde{\mathbf{u}}(t) = (u_1(t), \dots, u_M(t), 0, \dots, 0)^T$ , i.e.,  $\tilde{\mathbf{u}}(t)$  is

$\mathbf{u}(t)$ (the power vector) extended by  $N - M$  zeros, corresponding to nodes which have no power dissipation associated with them (e.g., thermal interface material, heat spreaders, heat sinks, etc.).

Let  $\mathbf{D} = (d_{ij})_{N \times N}$ ,  $\mathbf{C} = (c_{ij})_{N \times N}$ , where

$$d_{ij} = \begin{cases} -\sum_{k=1}^N \frac{1}{r_{ik}} & \text{if } i = j, \\ \frac{1}{r_{ij}} & \text{if } i \neq j. \end{cases} \quad (4)$$

$$c_{ij} = \begin{cases} c_i & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases} \quad (5)$$

Then equation (3) can be rewritten as:

$$\mathbf{C}\dot{\mathbf{x}}(t) = \mathbf{D}\mathbf{x}(t) + \tilde{\mathbf{u}}(t) \quad (6)$$

$\mathbf{C}$  is a diagonal matrix and thus, it is easy to compute its inverse  $\mathbf{C}^{-1}$  and obtain the standard differential equation:

$$\dot{\mathbf{x}}(t) = \mathbf{C}^{-1}\mathbf{D}\mathbf{x}(t) + \mathbf{C}^{-1}\tilde{\mathbf{u}}(t) \quad (7)$$

Note that since  $\tilde{u}_i = 0$  for  $i > M$ , only the left  $M$  columns of  $\mathbf{C}^{-1}$  are useful in the second term in (7). We can therefore, construct an  $N \times M$  matrix  $\mathbf{G}$  out of the left  $M$  columns of  $\mathbf{C}^{-1}$  and replace  $\tilde{\mathbf{u}}$  by  $\mathbf{u}$ . Thus, we obtain an equation similar to (1) with

$$\mathbf{F} = \mathbf{C}^{-1}\mathbf{D} \text{ and } \mathbf{G} = \text{left } M \text{ columns of } \mathbf{C}^{-1} \quad (8)$$

Therefore, a formula similar to (2) can be used to calculate the transient temperature of the CPU.

The input power trace to the CPU is usually given as a series of power vectors. In a sampling interval  $\Delta t$ , the power vector  $\mathbf{u}(t)$  is constant allowing us to simplify equation (2) as follows:

$$\mathbf{x}(\Delta t) = e^{\mathbf{F}\Delta t}\mathbf{x}(0) + \left[ \int_0^{\Delta t} e^{\mathbf{F}(\Delta t - \tau)} \mathbf{G} d\tau \right] \cdot \mathbf{u} \quad (9)$$

We use this simplified equation to reduce the amount of computation. Denoting

$$\mathbf{A} = e^{\mathbf{F}\Delta t}, \quad \mathbf{B} = \int_0^{\Delta t} e^{\mathbf{F}(\Delta t - \tau)} \mathbf{G} d\tau \quad (10)$$

we obtain the equation:

$$\mathbf{x}(\Delta t) = \mathbf{A}\mathbf{x}(0) + \mathbf{B}\mathbf{u} \quad (11)$$

Because the system is a time-invariant linear system, we obtain the same equation for any interval  $\Delta t$  with the same matrices  $\mathbf{A}$  and  $\mathbf{B}$ :

$$\mathbf{x}(n\Delta t) = \mathbf{A}\mathbf{x}((n-1)\Delta t) + \mathbf{B}\mathbf{u}(n-1) \quad (12)$$

where  $\mathbf{u}(n-1)$  is the power vector in the time interval  $[(n-1)\Delta t, n\Delta t]$ .

We will use  $\mathbf{x}(n)$  to represent  $\mathbf{x}(n\Delta t)$  for conciseness, resulting in:

$$\mathbf{x}(n) = \mathbf{A}\mathbf{x}(n-1) + \mathbf{B}\mathbf{u}(n-1) \quad (13)$$

### 3 Time Invariant Linear Thermal System (TILTS) Method

Our transient thermal simulation method TILTS is based on equation (13). Suppose the number of power vectors (called data points) in the input power trace  $\mathbf{u}$  is  $n$ , and the initial temperature is  $\mathbf{x}_0$ .

### 4 A Revised Algorithm CTILTS

The performance of the TILTS algorithm can be further improved without any loss of accuracy. Based on equation (13),

$$\begin{aligned} \mathbf{x}(p) &= \mathbf{A}^p\mathbf{x}(0) + \sum_{j=0}^{p-1} \mathbf{A}^{p-1-j}\mathbf{B}\mathbf{u}(j) \\ &= \mathbf{A}_p\mathbf{x}(0) + \sum_{j=0}^{p-1} \mathbf{L}_j\mathbf{u}(p-1-j) \end{aligned} \quad (14)$$

where

$$\mathbf{A}_p = \mathbf{A}^p, \quad \mathbf{L}_j = \mathbf{A}^j\mathbf{B}, j = 0, 1, \dots, p-1 \quad (15)$$

We can precompute the matrices  $\mathbf{A}_p$  and  $\mathbf{L}_j$ ,  $j = 0, 1, \dots, p-1$ , and save them in a table for later use. This is the motivation behind the revised algorithm.

The second term on the righthand side of equation (14) is the convolution of the input power trace and the step response of the thermal system. We call the revised algorithm Convolutional TILTS (CTILTS).



# Fast Transient Thermal Simulation Based on Linear System Theory

Yongkui Han, Israel Koren, C. Mani Krishna  
ARTS Lab, ECE Dept  
University of Massachusetts at Amherst

## Thermal Simulation is Important

- Thermal simulation has become a necessity for contemporary microprocessors
- Transient Thermal Simulation Models
  - 1. Finite Elements Method (FEM)
    - Very accurate but computation-intensive
  - 2. Compact thermal RC models
    - Less accurate but faster simulation
- The HotSpot simulator
  - based on the compact thermal RC model - at architecture level
  - developed by LAVA group at Univ. of Virginia

2 of 10

## HotSpot is Inefficient

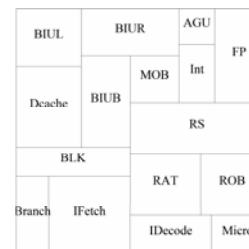
- HotSpot uses the fourth order Runge-Kutta method (rk4) for integration
  - To reduce the truncation error, the step size in rk4 must be small
  - Thousands of rk4 iterations are required when simulating for a 1ms time interval
- HotSpot becomes inefficient when attempting to obtain the temperature profile for a given benchmark
- Our new method improves the simulation speed of HotSpot

3 of 10

## Experiments

- Two microprocessors
- SPEC2000 benchmarks
- Most recent HotSpot v3.0.2 is used

Pentium Pro processor:  
16 power inputs, 58 nodes.



Alpha 21364 processor:  
18 power inputs, 97 nodes.



4 of 10

## CPU Chip as a Linear System

$$\dot{x}(t) = \mathbf{F}x(t) + \mathbf{G}u(t) \quad (1)$$



The response of the linear system is:

$$x(t) = e^{\mathbf{F}t}x(0) + \int_0^t e^{\mathbf{F}(t-\tau)}\mathbf{G}u(\tau)d\tau \quad (2)$$

During a time interval  $[(i-1)\Delta t, i\Delta t]$ , the input power  $u$  is fixed:

$$\text{Let } \mathbf{A} = e^{\mathbf{F}\Delta t}, \quad \mathbf{B} = \int_0^{\Delta t} e^{\mathbf{F}(\Delta t-\tau)}\mathbf{G}d\tau \quad u(i) \text{ is the input power during the interval } [(i-1)\Delta t, i\Delta t]$$

$$x(n) = \mathbf{A}x(n-1) + \mathbf{B}u(n-1) \quad (3)$$

The dimensions of  $x$ ,  $u$  depend on specific processor analyzed.

5 of 10

## Time Invariant Linear Thermal System (TILTS) method

- TILTS
  - Precompute matrices  $\mathbf{A}$  and  $\mathbf{B}$  using rk4 method
  - Compute temperatures using (3)
- Many rk4 iterations are replaced by simple matrix multiplications
- TILTS does not incur any accuracy loss compared to the HotSpot simulator

6 of 10

## Computation Reductions in TILTS

Comparing the number of Floating-Point Multiplications (FPM) in HotSpot and TILTS

processor	$\Delta t$	#FPM in HotSpot	#FPM in TILTS	ratio
Pentium Pro	5 $\mu s$	267670	4292	62
Alpha 21364	3.33 $\mu s$	514197	11155	46

7 of 10

## Convolutional TILTS

- Convolutional TILTS (CTILTS) algorithm
  - Precompute matrices  $\mathbf{A}^j$  and  $\mathbf{A}^j\mathbf{B}$ ,  $j=1,2,\dots,p$
  - Perform the convolution between the input power trace and the step response of the system
- CTILTS reduces the number of floating-point multiplications (FPM) by 5 or 6 times
- Small memory overhead

processor	interval	memory size	#FPM in TILTS	#FPM in CTILTS	ratio
Pentium Pro	5.1 ms	7.25 MB	4395008	953636	4.61
Alpha	3.4 ms	13.64 MB	11422720	1797313	6.36

8 of 10



## Experimental Results

processor	prog	HotSpot	TILTS	speedup	CTILTS	speedup
Pentium Pro	<i>gcc</i>	15800s	237s	67x	59s	268x
Pentium Pro	<i>gzip</i>	15790s	238s	66x	59s	268x
Pentium Pro	<i>bzip2</i>	15792s	237s	67x	59s	268x
Pentium Pro	<i>art</i>	15800s	238s	66x	59s	268x
Pentium Pro	<i>mgrid</i>	15790s	238s	66x	59s	268x
Alpha	<i>gcc</i>	28010s	592s	47x	129s	217x
Alpha	<i>gzip</i>	28030s	591s	47x	129s	217x
Alpha	<i>bzip2</i>	28020s	592s	47x	129s	217x
Alpha	<i>art</i>	28010s	591s	47x	129s	217x
Alpha	<i>mgrid</i>	28020s	592s	47x	129s	217x

- **TILTS**

- 67x for Pentium Pro, 47x for Alpha

- **CTILTS**

- 268x for Pentium Pro, 217x for Alpha

9 of 10

## Conclusions

- **Our Time Invariant Linear Thermal System (TILTS) method can greatly improve transient thermal simulation performance:**
  - 268 times faster than HotSpot for Pentium Pro processor
  - 217 times faster than HotSpot for Alpha 21364 processor
- **TILTS does not incur any accuracy loss compared to the HotSpot simulator**

10 of 10

# Impact of Process Variations on Low Power Cache Design

Mahmoud Bennaser and Csaba Andras Moritz  
University of Massachusetts, Amherst, MA 01003  
{mbennase, andras}@ecs.umass.edu

## I. INTRODUCTION

As technology scales, the feature size reduces thereby requiring a sophisticated fabrication process. The manufacturing process causes variations in many different parameters in the device, such as the effective channel length  $L_{eff}$ , the oxide thickness  $t_{ox}$ , and the threshold voltage  $V_{th}$ . These variations increase as the feature size reduces due to the difficulty of fabricating small structures consistently across a die or a wafer [1]. Controlling the variation in device parameters during fabrication is becoming therefore a great challenge for scaled technologies.

The performance and power consumption of integrated circuits are greatly affected by these variations. This can cause deviation from the intended design parameters for a chip and severely affects the yield as well as performance. Thus, process variations must be taken into consideration while designing circuits and architectures. We present a new adaptive cache architecture design which takes into consideration the effect of process variations on access latency. Preliminary results show that our new design can achieve a 13% to 29% performance improvement on the applications studied compared to a conventional design.

## II. IMPACT OF PROCESS VARIATIONS ON CACHES

In this paper, we analyze the impact of different sources of process variations on low power cache circuits. We use a state-of-the-art low power cache that we have presented in the last year's BARC as the start-point for our evaluation. We are interested in exploring delay and power consumption issues related to process variations and gathering insights that could be used in new cache designs as well as possibly in developing new architectural techniques for fetch units and load-store units in microprocessors.

Process variations in caches affect the performance and power dissipation of circuits like sense amplifiers that require identical device characteristics, and SRAM cells that require near-minimum-sized cell

stability for large arrays in embedded, low-power applications. Also, the delay of the address decoders suffer from the process variations that can result in shorter time left for accessing the SRAM Cells.

In order to examine delay and power consumption tradeoffs under process variations, we have evaluated the impact of process variations at 32-nm CMOS process technologies. We used the HSPICE circuit simulator and PTM technology.

We have found that the use of longer effective channel lengths ( $L_{eff}$ ) tends to increase the word-line and bit-line capacitances in caches, thus increasing access time as shown in Fig. 1. The access time can increase by as much as 16% per bit.

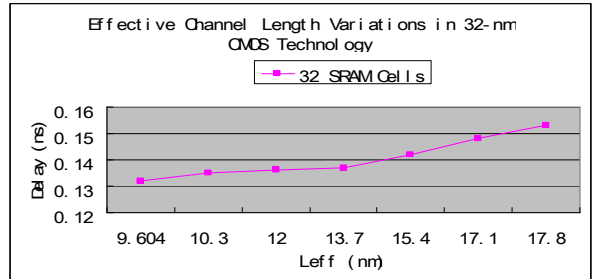


Fig. 1. Effect of  $L_{eff}$  Variation on Cache Delay (1-bit Read).

A small variation in the  $L_{eff}$  value causes a significant change in the power in the device: by as much as 40X from the nominal value (see Fig. 2).

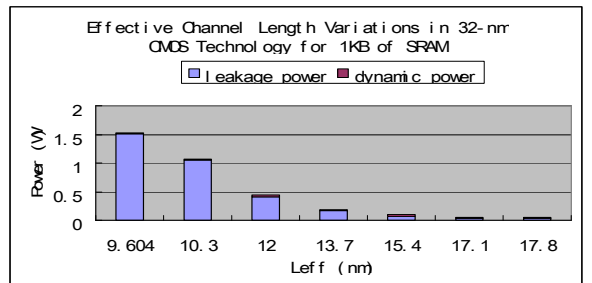


Fig. 2. Effect of  $L_{eff}$  Variation on Power (1-bit Read).

Higher transistor threshold voltages  $V_{th}$ , due to process variations, similarly negatively impact on access time and leakage power due to the lower read current as shown in Table I.

Table I  
Effect of Threshold Voltage Variation

Variation in $V_{th}$	Delay	Leakage Power
0.14 V	0.126 ns	0.801 W
0.18 V	0.129 ns	0.280 W
0.20 V	0.133 ns	0.167 W
0.22 V	0.142 ns	0.094 W
0.24 V	0.153 ns	0.055 W

Clearly, process variation can have a significant impact on delay, and in the worst-case leads to timing violations. In addition, power dissipation, especially leakage power, is shown to be significantly affected by the parameter variations. This could have important ramifications to both the circuit designer and the architect.

At the architectural level, process variations have an impact on memory hierarchy design. There are several ideas that could be exploited to cope with this problem. These could range from utilizing smaller first level caches (that would meet the preferred access time even under worst case variation) to more adaptive cache architectures/schemes.

### III. AN ADAPTIVE PROCESS RESILIENT CACHE ARCHITECTURE

Fig. 3 shows the proposed delay resilient cache architecture. It consists of two phases of operation: classification and execution.

The cache is equipped with a BIST circuitry, which tests the entire cache and a double sensing technique (e.g., as in [2]) to detect speed of the cells during the classification phase. Each cache line is tested using BIST when the test mode signal is on. For example, a block is considered fast, medium, or slow. BIST feeds this information into the delay storage.

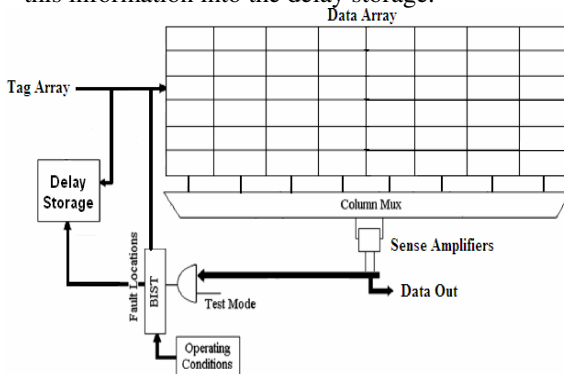


Fig. 3. Adaptive cache architecture during classification phase.

The speed information stored in the delay storage is used to control sense amplifiers during regular operations of the circuit (see Fig. 4).

We have conducted simulations of SPEC benchmarks using the adaptive cache design. Preliminary results on application performance are shown in Fig. 5. The comparison is made between conventional caches that require 3 cycles per access (worst case due to process variation) vs. our adaptive cache that has variable cache access time. The adaptive cache has 3% of 3 cycles, 12% of 2 cycles and 85% of 1 cycle cache line accesses.

Our preliminary results (for one example delay distribution) show an improvement of up to 28% possible in a 4-way superscalar equipped with a 16KB L1 data cache. Much additional work is required to quantify all the effects.

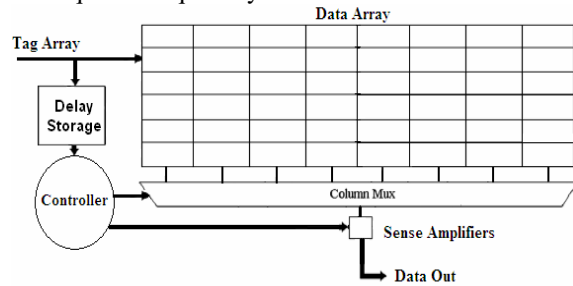


Fig. 4. Adaptive cache architecture during the execution phase.

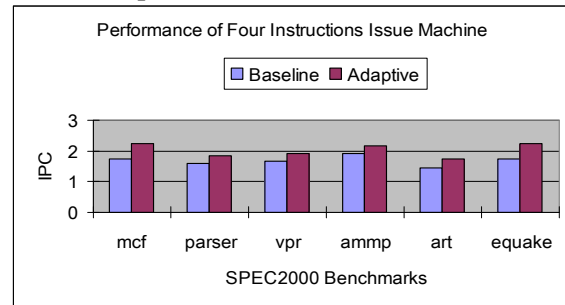


Fig. 5. Performance improvement with our adaptive cache vs. a cache using worst case access time.

Clearly, process variation introduces new delay and power tradeoffs that must be considered. This talk will explore some of the challenges for both circuit designers and architects working on memory system architectures.

### REFERENCES

- [1] A. Chandrakasan, et al., "Design of High Performance Microprocessor circuits", IEEE Press, 2001.
- [2] Q. Chen, et al. "Modeling and Testing of SRAM for New Failure Mechanisms due to Process Variations in Nanoscale CMOS", VLSI Test Symposium, May, 2005.

BARC 2006

Boston area ARChitecture Workshop

**Impact of Process Variations on  
Low Power Cache Design**

Mahmoud Bennaser and Csaba Andras Moritz  
Department of Electrical and Computer Engineering  
University of Massachusetts, Amherst  
February 3, 2006

1 of 16

**Introduction**

- Process variations increase as the feature reduces due to the difficulty of fabricating small structures consistently across a die or wafer.
- In order to analyze the delay and power consumption of a cache under process variation, we must consider both inter-die and intra-die variation
  - **Intra-die variations** are the variations in device parameters within a single chip, which means different devices at different locations on a single die may have different device features
  - **Inter-die variations** are the variations that occur from one die to the other, from wafer to wafer, and from wafer lot to wafer lot
- Two main sources of variation:
  - **Physical factors**
  - **Environmental factors**

2 of 16

**Introduction**

- The physical factors are permanent and result from limitations in the fabrication process
  - **Effective Channel Length** (Geometric Variations):
    - Imperfections in photolithography
    - Variations in  $L_{eff}$  can be as high as 50% within a die
  - **Threshold Voltage** (Electrical Parameter Variation):
    - Variation in device geometry
    - Variations in  $V_{th}$  can be modeled as 10% of  $V_{th}$  of the smallest device in a given technology [A. Chandrakasan et al., IEEE press 2001]
- The environmental factors depend on the operation of the system and include variations in:
  - **Temperature, Power Supply, Switching Activity**

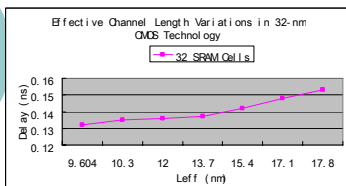
3 of 16

**Impact of Process Variations  
on Caches**

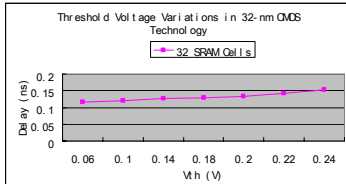
- The parameter variations are random in nature and are expected to be more pronounced in minimum geometry transistors commonly used in memories.
  - Caches in processors like UltraSPARC III, Itanium 2, StorgARM110, and Alpha 21164 can occupy more than 50% of die area.
- Process variations impact the components of a memory subsystem:
  - SRAM Cell
  - Sense Amplifier
  - Address Decoder
- Can cause failure in data access
  - E.g., due to incorrect sensing or slow cell access

4 of 16

## Effect of Process Variations on Delay Accessing 1-bit in SRAM Column of 32 Bit Height



The delay can increase as such as 16% per cell.

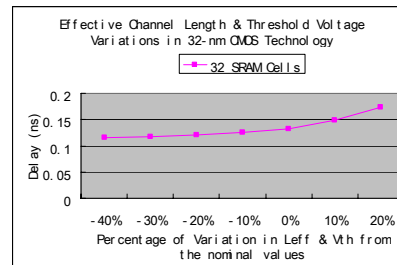


The Threshold voltage ( $V_{th}$ ) variation can impact the delay by 30% per cell access

[HSPICE simulation]

5 of 16

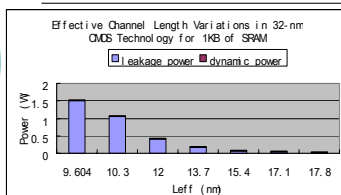
## Worst-case Delay



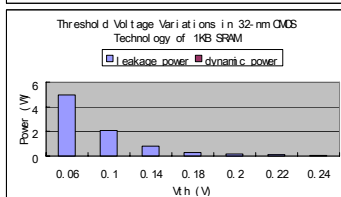
The delay can increase as such as 50% combining the effects of  $V_{th}$  and  $Le_{ff}$ .

6 of 16

## Effect of Process Variations on Power Consumption of 1KB SRAM



A small variation in the  $Le_{ff}$  value causes a change in the leakage power by as such as 40X from the nominal value.



The Threshold voltage ( $V_{th}$ ) variation can impact the power consumption by 65X

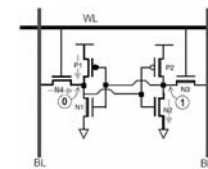
[HSPICE simulation]

7 of 16

## Cache Access Failure?

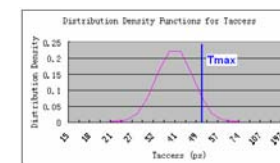
o A failure in a cell can occur due to:

- Access Time Failure (due to increase in the access time)
- Read Stability Failure
- Write Stability Failure
- Hold Failure



o Failure Probability of a Read

- E.g., the minimum differential voltage required for correct sensing ( $T_{access}$  in figure) needs to be  $< T_{max}$  for a correct read
  - o Threshold voltage distributions are approximated as Gaussian

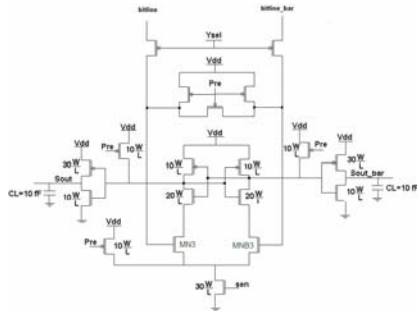


[Classification is take from S. Mukhopadhyay, et al. Symposium on VLSI Circuits, June 2004 ]

8 of 16

## Failure in Sense Amplifiers

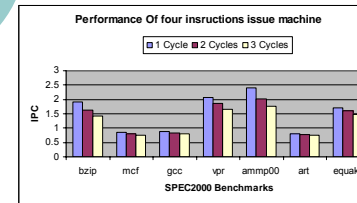
- Circuits like differential sense amplifiers are affected
  - Changing offset voltage may lead to erroneous behavior (e.g., due to access Transistors MN3 and MN3B).



9 of 16

## What About Application Performance?

- To account for the worst case scenario we might need to increase the cache access time
- Performance impact as much as 30-40% in the example on the left



[simplescalar simulations]

10 of 16

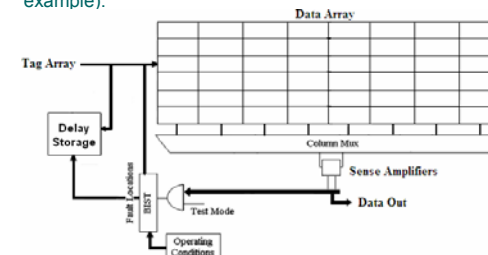
## Possible Architectural Directions

- How do we design caches that work in face of these problems?
- We can select a cache design using worst case assumptions
  - ALL VARIATIONS and ALL COMPONENTS on the critical path
- Alternatively, we need to design circuits and architectures that would work *adaptively* depending on actual delay
  - Process variation resilient design
  - Resilience against delays in different parts of the cache

11 of 16

## Our Choice: An Adaptive Process Resilient Cache Architecture

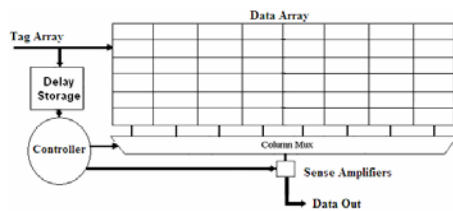
- Two phases of operation: classifying and execution
- Classifying phase
  - The cache is equipped with a built-in-self-test (BIST) to detect speed difference due to process variation.
  - Each cache line is tested using BIST when the test mode signal is on. A block is considered fast, medium, or slow (this is for the sake of an example).



12 of 16

## An Adaptive Process Resilient Cache Architecture

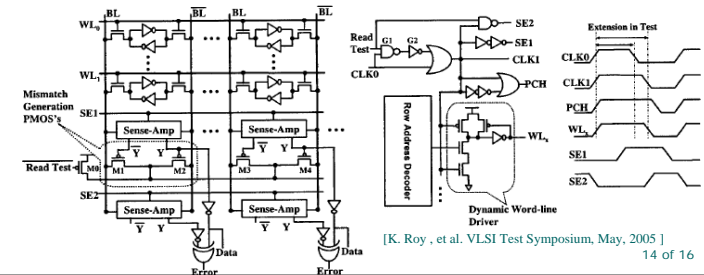
- Since the speed of the accessed cells (cache lines) changes depending on operating condition (e.g., supply voltage, frequency), such tests are conducted whenever there is a change in operating condition.
  - BIST feeds this information into the delay storage.
- Execution phase
  - The speed information stored in the delay storage is used to control sense amplifiers during regular operations of the circuit.



13 of 16

## Circuit Level Support: Double Sensing

- We need a mechanism to avoid sensing prematurely
- The basic idea of double sensing is to have parallel sense amplifiers to sample the bitline twice during a read cycle. This is required in an adaptive cache design with different cache line latencies.
- The first sensing is performed as the conventional one. The second sensing is delayed and has to be fired as late as required.

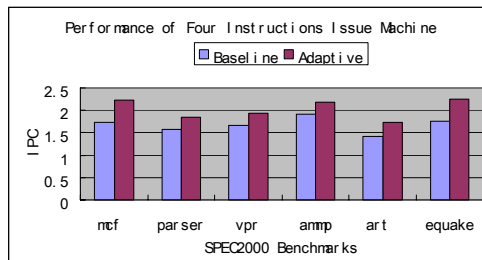


[K. Roy, et al. VLSI Test Symposium, May, 2005]

14 of 16

## Preliminary Results

Baseline: 3 cycle D-cache. Out of order issue.  
 Adaptive caching scheme: e.g.,  
 3% 3 cycle, 12% 2 cycle, 85% 1 cycle cache line access.  
 Results below show performance is improved by 13% to 29%!



15 of 16

## Conclusion

- Parameter variations will become worse with technology scaling.
- Robust variation tolerant circuits and architectures needed.
- We have shown that process variation can have a significant impact on delay (expected > 2X with all factors included), and in worst-case leads to timing violations.
- In addition, power dissipation, especially leakage power has been shown to be significantly affected (>60X) by the parameter variations.
- Shown new resilient cache architecture

16 of 16

# Compiler-Based Adaptive Fetch Throttling for Energy-Efficiency

Huaping Wang, Yao Guo, Israel Koren, C. Mani Krishna  
Department of Electrical and Computer Engineering  
University of Massachusetts, Amherst, MA 01003  
E-mail: {hwang,yaoguo,koren,krishna}@ecs.umass.edu

## 1. Introduction

Front-end instruction delivery accounts for a significant fraction of energy consumption in dynamically scheduled superscalar processors. Different front-end throttling techniques have been introduced to reduce the chip-wide energy consumption caused by redundant fetching. The techniques can be categorized as hardware-based runtime [1, 3] and software-based static [2, 4] techniques.

Hardware-based techniques assume the program state is stable and use the current history information to predict future behavior. These can catch dynamic behavior such as cache misses but cannot catch irregular situations such as abrupt phase changes.

Software-based throttling techniques can estimate the instruction-level parallelism (ILP) based on compile-time program analysis and provide indications of sharp changes in ILP (or ILP bursts). Static techniques may however, produce inaccurate predictions due to their inability to capture dynamic effects such as branch mispredictions and cache misses. Previous research [4] employed compiler techniques to estimate the IPC and used the estimated IPC to drive its fine-grained fetch-throttling energy-saving heuristic. A fetch will be stalled in the following cycle if the estimated IPC is lower than a predefined threshold. Throttling using a low threshold will have a small effect on performance but yield relatively small energy savings.

There are two potential problems using a fixed low value of the IPC-threshold to drive fetch throttling. The first one is that it limits the throttling opportunities at high IPC values. If there are many instructions left unexecuted in the previous cycle, we can throttle at a higher IPC-threshold with probably no performance decrease. The second problem is that the fixed IPC-threshold technique may throttle at an inappropriate time, resulting in a performance loss. Assume for example, that the estimated IPC in the following cycle is 2, but if there are no instructions left in the issue queue from the previous cycle; a throttling at this time is inappropriate and will result in a performance loss. If we can drive fetch throttling using adaptive

IPC-thresholds instead of a fixed one, we could increase the throttling opportunities and achieve higher energy savings.

In this paper, we present a new approach called Compiler-based Adaptive Fetch Throttling (CAFT), which allows changing the throttling IPC-threshold adaptively at runtime. Our technique is based on compile-time static IPC estimation, but we use the Decode/Issue Difference (DID) to assist the fetch throttling decision based on the statically estimated IPC. DID is the difference between the numbers of decoded and issued instructions in the previous cycle, which can be considered as recent history information. The IPC-threshold is changed dynamically according to the DID value, making it possible to throttle at a higher estimated IPC. This increases the throttling opportunities and thereby results in larger energy savings.

## 2. The CAFT framework

### 2.1. Compiler-based IPC Estimation

We use compile-time static IPC-estimation to drive throttling, which is similar to what has been proposed by Unsal *et al.* [4].

Our implementation considers only true data dependencies (Read-After-Write or RAW). We statically determine true data dependencies using data dependency analysis at the assembly-code level. We identify data dependencies at both registers and memory accesses. Register analysis is straightforward. However, for memory accesses, we performed an approximate and speculative alias analysis by instruction inspection that provides ease of implementation and sufficient accuracy. In this scheme, we distinguish between different classes of memory accesses such as static or global memory, stack and heap. We also consider indexed accesses by analyzing the base register and offset values to determine if different memory accesses are referenced.

We use SUIF/MachSUIF as our compiler framework. We added new passes to both SUIF and



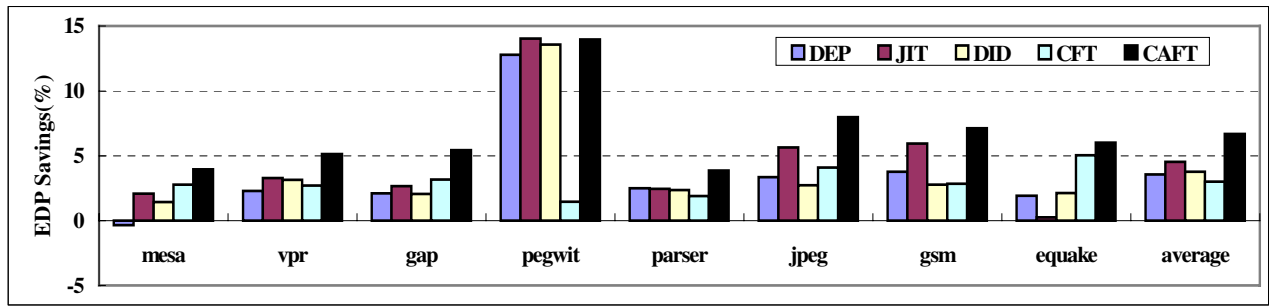


Figure 1. Energy Delay Product (EDP) savings

MachSUIF to annotate and propagate the static IPC-estimation. Our IPC-estimation is at the basic block or loop level: loop beginnings and ends serve as natural boundaries for the estimation. The high-level loop annotation pass works with expression trees and traverses the structured control flow graph (CFG) of each routine. The other added pass, the IPC-prediction pass, is a lower-level MachSUIF pass that runs just prior to assembler code generation.

## 2.2. CAFT: Compiler-Based Adaptive Fetch Throttling

As mentioned above, we will use the instruction Decode/Issue Difference (DID) to assist the IPC-estimation throttling technique to throttle at changeable thresholds. If the instruction decoding rate matches the instruction issuing rate (i.e., the DID value is zero), no fetch throttling is needed. Additional fetching will introduce the possibility of miss-fetching and increase the number of Icache accesses, resulting a waste of energy. If the DID value in the last cycle is greater than zero, which means that redundant instructions were decoded, there exist opportunities to throttle the fetch at the following cycle. For example, if DID in the previous cycle is 3 and the IPC estimate in the next cycle is less than 3, we can safely throttle for one cycle during instruction fetching. If the instructions left unused in the previous cycle can provide the needs of the next cycle, stopping fetching for one cycle will not decrease the performance. For different DID values, we will throttle for all the estimated IPCs up to the DID value.

Dynamic effects such as cache misses and branch mispredictions are captured by the DID value. In contrast, if only compiler-based fetch throttling is used, these dynamic effects cannot be considered, forcing the technique to use a low fixed IPC-threshold.

## 3. Results

Compared to the previous fixed threshold approach (CFT), the total number of throttling cycles increases substantially because CAFT can throttle fetching at higher estimated IPC values. The increase in the number of throttling cycles results in a considerable reduction in energy consumption. We show in Figure 1 that CAFT achieves a 3.7% additional Energy-Delay Product (EDP) saving compared to CFT and 6.7% overall EDP reduction. In comparison with a previous hardware dependence-based fetch throttling scheme (DEP), CAFT has a lower performance degradation, achieving a 3.2% additional EDP reduction.

## References

- [1] A. Baniasadi and A. Moshovos. “Instruction Flow-based Front-end Throttling for Power-aware High-performance Processors”. In *Proceedings of the International Symposium on Low Power Electronics and Design, ISLPED '01*, 2001.
- [2] T. M. Jones, M. F. P. O’Boyle, J. Abella, and A. González. “Software Directed Issue Queue Power Reduction”. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture, HPCA-11*, pages 144–153, 2005.
- [3] T. Karkhanis, J. E. Smith, and P. Bose. “Saving Energy with Just in Time Instruction Delivery”. In *Proceedings of the 2002 international symposium on Low power electronics and design, ISLPED '02*, pages 178–183, 2002.
- [4] O. S. Unsal, I. Koren, C. M. Krishna, and C.A.Moritz. “Cool-fetch: Compiler-enabled Power-aware Fetch Throttling”. In *IEEE Computer Architecture Letters*, volume 1, 2002.

## Compiler-Based Adaptive Fetch Throttling for Energy-Efficiency

Huaping Wang, Yao Guo,  
Israel Koren and C. Mani Krishna

ECE Dept, UMass at Amherst

### Introduction

- ❑ Power consumption increases significantly in modern computer architecture.
- ❑ Fetch throttling can reduce executions of miss-fetched instructions and number of lcache accesses.

2 of 11

### Fetch throttling techniques

- ❑ Hardware-based runtime techniques
  - Use past behavior to predict future behavior.
  - Can not catch irregular situations such as abrupt program phase changes.
  - Cause substantial performance degradation.
- ❑ Software-based static techniques
  - Estimate Instruction Level Parallelism (ILP) based on compile-time program analysis.
  - Can not capture dynamic effects, such as cache misses.
  - Use fixed low IPC threshold for throttling - to avoid high performance loss.
  - Energy savings is small if IPC threshold is low.

3 of 11

### Potential problems of fixed low IPC threshold

- ❑ Limits throttling opportunities at high IPC values:
  - If estimated IPC (e.g., 3) is less than number of instructions left unexecuted in previous cycle (e.g., 5), we can throttle fetch even at a high IPC value.
- ❑ May throttle at an inappropriate time resulting in a performance loss:
  - If estimated IPC is low (e.g., 2) but no instructions left in the issue queue (from previous cycle), throttling results in performance loss.

4 of 11

## Compiler-based Adaptive Fetch Throttling (CAFT)

- IPC estimate using compile-time analysis.
- A large Decode/Issue Difference (DID) means that many instructions were left unexecuted.
- DID value can be used as recent history information to change the IPC threshold adaptively

```
IF Estimated_IPC ≤ DID
THEN throttle for one cycle
```

5 of 11

## Compiler-level implementation

- Used SUIF/MachSUIF as our compiler framework
- Added new passes to both SUIF and MachSUIF to annotate and propagate the static IPC-estimation
- Compiler-based IPC estimate
  - Consider only true data dependencies.
  - Identify data dependencies for both registers and memory accesses.
  - Use approximate and speculative alias analysis for memory accesses.

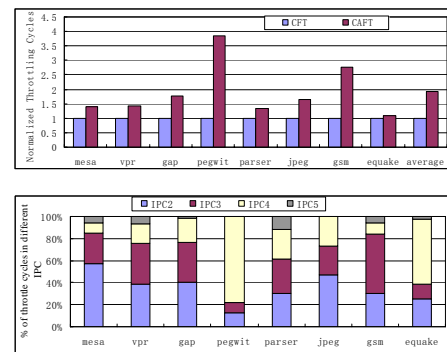
6 of 11

## Experiments

- Setup
  - SimpleScalar/Watth
  - SPEC2000 and Mediabench benchmarks
- Examined several existing throttling techniques
  - Hardware dependence-based (DEP)
  - Just-In-Time instruction delivery (JIT)
  - Compiler-based fixed IPC threshold (CFT)
- Compared CAFT to above techniques
  - Throttling cycles and IPC threshold distribution
  - Execution Time and Energy
  - Energy Delay Product (EDP)

7 of 11

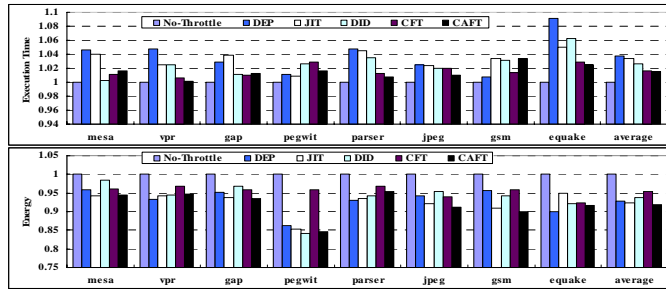
## Number of throttling cycles and IPC distribution



- Number of throttling cycles increases significantly compared to fixed low IPC-threshold
- Percent of throttling cycles above IPC-threshold of 2 is larger than 50% in most benchmarks

8 of 11

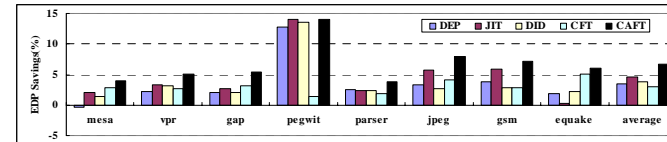
## Execution time and energy



- CAFT keeps the advantage of low performance decrease of CFT, and has a good energy savings as hardware-based techniques.

9 of 11

## Energy Delay Product (EDP)



- Compared to fixed threshold technique (CFT), CAFT achieves a 3.7% additional EDP saving and 6.7% overall EDP reduction.
- Compared to DEP, CAFT achieves a 3.2% additional EDP reduction.

10 of 11

## Conclusion

- CAFT has a better EDP savings than software- or hardware-only fetch throttling techniques.

11 of 11

## Experiment setup (Backup)

- Skip the initialization stage and simulate next 500M instructions for SPEC; run Mediabench to completion.

Processor Speed	5GHz
Process Parameters	0.18 $\mu$ m, 2V
Issue	Out-Of-Order
Fetch,Issue,Decoded,Commit	8-way
Fetch Queue Size	32
Instruction Queue Size	128
Branch Prediction	2K entry bimodal
Int.Functional Units	4 ALUs, 1 Mult./Div.
FP Functional Units	4 ALUs, 1 Mult./Div.
L1 D-cache	128Kb, 4-way, writeback
L1 I-cache	128Kb, 4-way, writeback
Combined L2 cache	1Mb, 4-way associative
L2 Cache hit time	20 cycles
Main memory hit time	100 cycles

12 of 11

# Energy-Aware Microprocessor Synchronization: Transactional Memory vs. Locks

Tali Moreshet\*, R. Iris Bahar\* and Maurice Herlihy†

\*Brown University, Division of Engineering, Providence, RI 02 912

†Brown University, Department of Computer Science, Providence, RI 02912

{tali,iris}@lems.brown.edu, mph@cs.brown.edu

Energy consumption is an increasingly important issue in multiprocessor design. The need for energy-aware systems is obvious for mobile systems, where low energy consumption translates to longer battery life, but it is also important for desktop and server systems, where high energy consumption complicates power supply and cooling.

While energy consumption in uniprocessors has been the focus of a substantial body of research, energy consumption in multiprocessors has received less attention. This issue is becoming increasingly important as multiprocessor architectures migrate from high-end platforms into everyday platforms such as desktops, laptops, and servers. In particular, the increasing availability of multi-threaded and multi-core machines means that we can expect multiprocessors to replace uniprocessors in many low-end systems. Past studies have estimated that on-chip caches are responsible for at least 40% of the overall processor power (e.g., [1]). However, the dominant portion of energy consumption in the memory hierarchy is due to off-chip caches, resulting from their significantly larger size, and the higher capacitance board buses.

The principal way in which multiprocessors differ from uniprocessors is in the need to provide programmers the ability to synchronize concurrent access to memory. When multiple threads access a shared data structure, some kind of *synchronization* is needed to ensure that concurrent operations do not interfere. The conventional way for applications to synchronize is by *locking* [2]: for each shared data structure, a designated bit in shared memory (the *lock*) indicates whether the structure is in use.

Nevertheless, conventional synchronization techniques based on locks have substantial limitations [4]. Coarse-grained locks, which protect relatively large data structures, simply do not scale. Threads block one another even when they do not really interfere, and the lock itself causes memory contention. Fine-grained locks are more scalable, but they are difficult to use. In particular, they introduce substantial software engineering problems, as the conventions associating locks with ob-

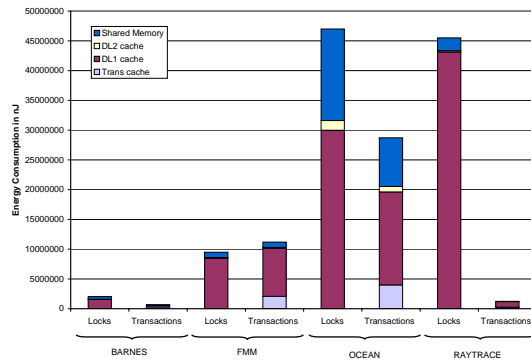
jects become more complex and error-prone. Locks are also vulnerable to thread failures and delays; if a thread holding a lock is delayed by a cache miss, page fault, or context switch, other running threads may be blocked.

*Transactional memory* [5] is a synchronization architecture that addresses these limitations. A transaction is a finite sequence of memory reads and writes executed by a single thread. Transactions are *atomic* (each transaction either completes and commits, or aborts) and are serializable. Hardware transactional memory proposals (e.g., [3], [5], [6], [8], [9], [10], [11]) exploit hardware mechanisms such as speculative execution and on-chip caching. Hardware optimistically executes a transaction and locally caches memory locations read or written on behalf of the transaction, marking them transactional. The hardware cache coherence mechanism communicates information regarding read and write operations to other processors. A data conflict occurs if multiple threads access a given memory location via simultaneous transactions and at least one thread's transaction writes the location. A transaction commits and atomically updates memory if it completes without encountering a synchronization conflict.

Transactional memory was originally proposed as a means of increasing throughput and improving ease of programming relative to locks. Although it seems plausible that transactions may be more energy-efficient than locks, the precise tradeoffs are not clear, especially under high rates of conflict. In this paper, we consider the energy/performance tradeoffs associated with these two approaches to multiprocessor synchronization. We conclude that transactional models are a promising approach to low-power synchronization, but that further work is needed to fully understand how transactions compare to locks when synchronization conflict rates are high.

To compare our hardware transactional memory to a baseline system that uses locks, we started by running the SPLASH-2 benchmark suite [12] since it is the most commonly used benchmark suite for parallel applications. We arbitrarily selected a few benchmarks, ran each benchmark after its initialization stage for 200

**Figure 1. Energy consumption of a splash2 benchmarks using locks vs. transactions.**



locks, and then ran it again with locks replaced by transactions. When replacing the lock with a transaction, the critical section defines the bounds of a transaction.

Figure 1 shows the energy consumption resulting from cache and memory accesses when synchronization was handled with either locks or transactions. We see that in most cases replacing locks with transactions reduced the number of cache and memory accesses, thereby reducing the energy consumption. With this initial analysis, it appears that transactions do indeed have a large benefit over locks in terms of energy consumption. However, we have found that the SPLASH-2 benchmarks do not test our assumptions well when the system is operating under high contention. We show that these initial results do not allow for a complete comparison.

Synchronization conflicts cause transactions to abort and restart, causing the system to consume energy doing useless work. Motivated by this tradeoff, in this paper we propose a *serial execution mode* for transactional memory in which transactions are adaptively serialized at the hardware level with the intent of decreasing energy consumption at the possible cost of degraded throughput. We note that in previous work [7] we presented an initial investigation of this topic and showed a single case of using locks vs. transactions as a motivation for the advantage of transactions over locks for energy consumption. Here we extend that work by providing a more detailed analysis into the various tradeoffs.

## References

- [1] David H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *32nd Intl. Symposium on Microarchitecture*, November 1999.
- [2] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138,

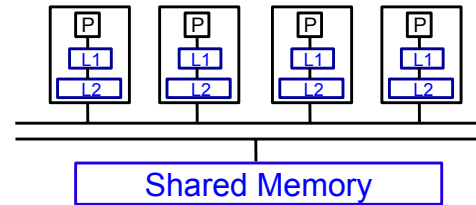
- 1971.
- [3] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstorm, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *31st Intl. Symposium on Computer Architecture*, June 2004.
- [4] Maurice Herlihy, Victor Luchangco, Mark Moir, and William Scherer. Software transactional memory for dynamic-sized data structures. In *Symposium on Principles of Distributed Computing*, July 2003.
- [5] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *23rd Intl. Symposium on Computer Architecture*, May 1993.
- [6] Jose F. Martinez and Josep Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.
- [7] Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. Energy reduction in multiprocessor systems using transactional memory. In *Intl. Symposium on Low Power Electronics and Design*, August 2005.
- [8] Jeffrey Oplinger and Monica S. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.
- [9] Ravi Rajwar and James Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.
- [10] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *32nd Intl. Symposium on Computer Architecture*, June 2005.
- [11] Janice M. Stone, Harold S. Stone, Phil Heidelberger, and John Turek. Multiple reservations and the Oklahoma update. *IEEE Parallel and Distributed Technology*, 1(4):58–71, November 1993.
- [12] Steven C. Woo, Moriyoishi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *22nd Intl. Symposium on Computer Architecture*, pages 24–36, June 1995.

# Energy-Aware Microprocessor Synchronization: Transactional Memory vs. Locks

**Tali Moreshet**   **R. Iris Bahar**   **Maurice Herlihy**  
 Division of Engineering   Department of Computer Science  
 Brown University   Brown University



## Shared Memory Architecture



- Atomic memory access  
Increment variable in address A

Load (R1, A)  
 Add (R1, R1, 1)  
 Store (A, R1)

## Synchronization of Accesses to Shared Memory

### Lock

- Represented by field in memory
- Repetitive accesses until free
- Coarse/Fine-grain
- Disadvantages:
  - High contention
  - Low throughput
  - High energy consumption

### Transaction

- Lock-free execution
- Speculative, optimistic
- Ease of programming
- Disadvantages:
  - Requires HW support
  - Roll-back and reissue if conflict detected (wasted cycles and energy)

## During a Transaction

Tag	Data	Status
		Invalid
12		<del>Exclusive</del>

Tag	Data	Status	Trans.Tag
12		Exclusive	Xabort
12		Exclusive	Xcommit

- Lookup in both DL1 and transactional cache
- If the line is found in DL1, move it to transactional cache
- If a miss, bring from L2 to transactional cache

## Considerations

- In the past designers only considered ease of programming and throughput
- Synchronization has a cost in terms of throughput and energy
- We take a first look at tradeoffs for
  - Ease of programming*
  - Throughput*
  - Energy*

5 of 15

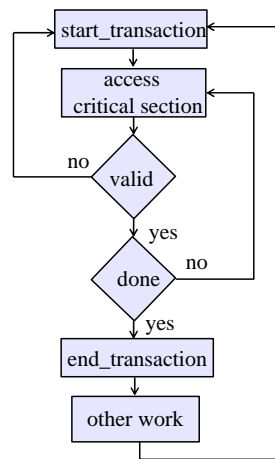
## Energy Consumption per Access

L1Data Cache	8KB 4-way; 32B line; 3 cycle latency	0.47nJ
Transactional Cache	64-entry; fully associative	0.12nJ
L2 Cache	128KB 4-way; 32B line; 10 cycle latency	0.9nJ
Shared Memory	256MB; 64-bit bus; 200 cycle latency;	<b>33nJ</b>

Sources: Micron SDRAM power calculator  
CACTI  
Private industrial communication

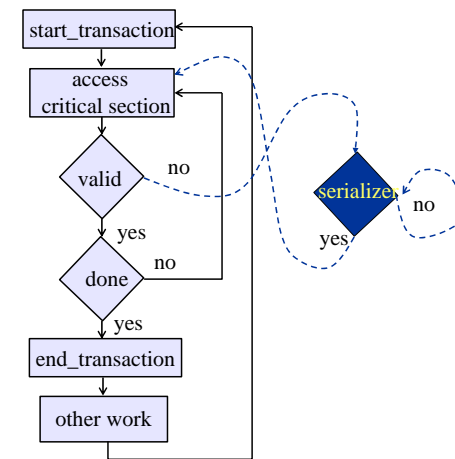
6 of 15

## Standard Transactions



7 of 15

## Standard Transactions



8 of 15

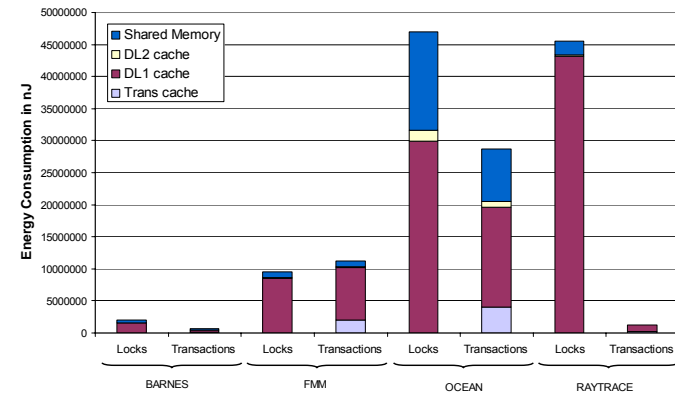


## Serializer

- Only impacts conflicting transactions
- Small overhead in hardware
- Reduce useless execution
- Reduce energy consumption
- Potentially negative impact on throughput

9 of 15

## Standard Benchmarks Results



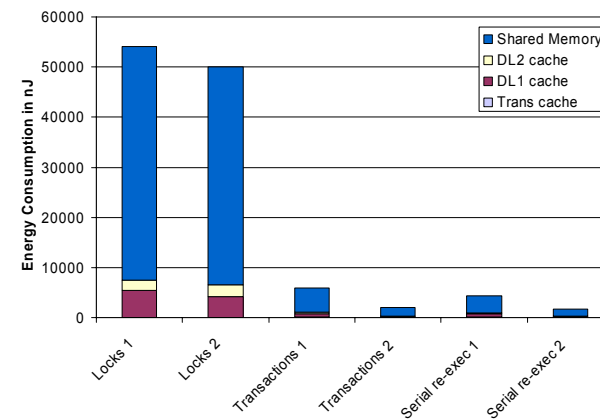
10 of 15

## Synthetic Benchmarks

- Standard benchmarks have little contention
- Realistic applications include intervals of high contention
- Synthetic benchmarks
  - High contention
  - Various conflict scenarios
- Parallel accesses to a shared array

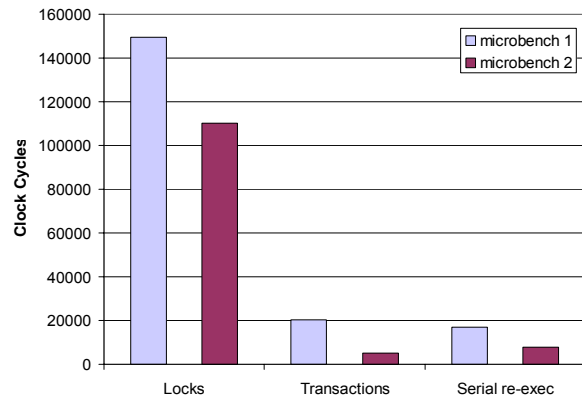
11 of 15

## Energy Consumption Locks vs. Transactions



12 of 15

## Performance Locks vs. Transactions



13 of 15

## Conclusion

- Throughput and energy need to be balanced
- Speculative approach has a clear advantage in both energy and throughput in low contention
- Speculative approach needs modification in high contention for energy efficiency:
  - ➔ *serialized transactions*

14 of 15

## Future Work

- Simulate a wider range of applications
- Various memory configurations
- Compare alternative locking schemes
- Consider longer running transactions
  - A trace-based analysis
  - Software transactions

15 of 15

# Investigating the Effects of Wrong-Path Memory References in Shared-Memory Multiprocessor Systems

Ayşe Yilmazer<sup>1</sup>, Resit Sendag<sup>1</sup>, Joshua J. Yi<sup>2</sup>, and Augustus K. Uht<sup>1</sup>

<sup>1</sup> - Department of Electrical and Computer Engineering  
University of Rhode Island, Kingston, RI  
yilmazer, sendag, uht@ele.uri.edu

<sup>2</sup> - Networking and Computing Systems Group  
Freescale Semiconductor, Inc., Austin, TX  
joshua.yi@freescale.com

## Abstract

*Uniprocessor studies have shown that wrong-path memory references pollute the caches by bringing in data that are not needed for the correct execution path and by evicting useful data or instructions. Additionally, they also increase the amount of cache and memory traffic. On the positive side, however, they may have a prefetching effect for loads and instructions on the correct path. While the wrong-path effects are well studied for uniprocessors, there is no work on its effects on multiprocessor systems. In this paper, we explore the effects of wrong-path memory references on the memory system behavior of shared-memory multiprocessor (SMP) systems with broadcast (snoop-based) and directory-based cache coherence. We show that in contrast to uniprocessor systems, these wrong-path memory references can increase the amount of cache-to-cache transfers by 32%, invalidations by 8% and 20% for broadcast and directory-based SMPs, respectively, and the number of write-backs by up to 67% for both systems. In addition to the extra coherence traffic, wrong-path memory references also increases the number of cache line state transitions by 21% and 32% for broadcast and directory-based SMPs, respectively.*

## 1 Introduction

Shared-memory multiprocessor (SMP) systems are typically built around a number of high-performance out-of-order superscalar processors, each of which employs aggressive branch prediction techniques in order to achieve high issue rate. During execution, these processors speculatively execute the instructions following the direction and target of a predicted branch instruction. If later detected incorrect, these wrong-path memory references do not change the processor's architectural state, however, they do change the data and instructions that are in the memory hierarchy, which can affect the processor's performance.

Several authors have studied the effects that speculatively executed memory references have on the performance of out-of-order superscalar processors [2, 3, 4] and have shown that wrong-path memory references may function as indirect prefetches by bringing data into the cache that are needed later by instructions on the correct execution path [1, 3, 4]. However, these wrong-path memory references also increase the amount of

memory traffic and can pollute the cache with cache blocks that are not referenced by instructions on the correct path [1, 3].

In this paper, we focus on the effect that wrong-path memory references have on the memory system behavior of SMP systems, in particular, broadcast-based and directory-based SMP systems. For these systems, not only do the wrong-path memory references affect the performance of the individual processors, they also affect the performance of the entire system by increasing the number of coherence transactions, the number of cache line state transitions, the number of write-backs and invalidations due to wrong-path coherence transactions, and the amount of resource contention (buffer usage, bandwidth, etc.).

## 2 Evaluating Wrong-Path Effects

In this section, we discuss the potential effects that wrong-path memory references can have on the memory behavior of SMP systems. To measure the various wrong-path effects, we track the speculatively generated memory references, and mark them as being on the wrong-path when the branch misprediction is known. Due to the space limitation, we only give a subset of the effects. For a complete version refer to [5].

### 2.1 L1 and L2, and Coherence Traffic

We observe that wrong-path loads increase the total number of memory references issued to the memory system on average by 17% and 14%, respectively, for broadcast and directory-based SMPs. Additionally, these loads increase the percentage of L2 cache accesses by 23% and 21% for broadcast and directory-based SMP systems, respectively. Our results also show that wrong-path loads increase the number of coherence transactions by an average of 32%.

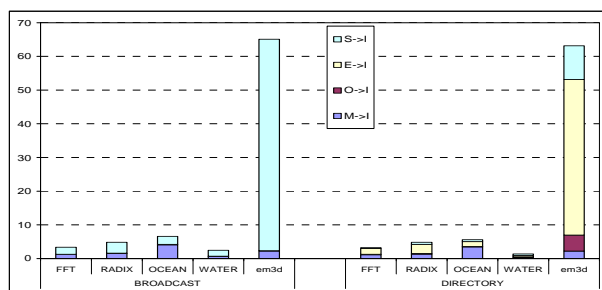
### 2.2 Replacements and Write-backs

A speculatively-executed (*i.e.*, later detected as wrong-path) load instruction may bring a cache block into processor's data cache that replace another block that may be needed later by a correct-path load. Due to this replacement, these wrong-path load can cause extra cache misses *i.e.* pollution [3]. Wrong-path replacements may also cause extra write-backs that would not occur otherwise. For example, if the requested wrong-path block

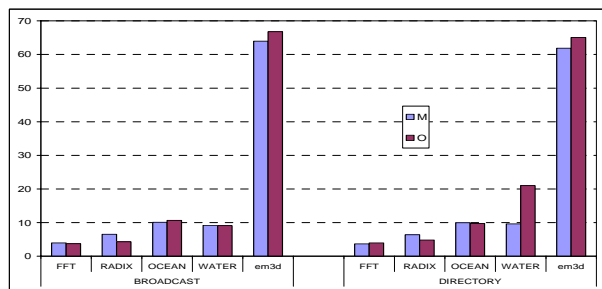
has been modified by another processor, *i.e.*, its cache coherence state is M, a shared copy of that block is sent to the requesting processor's cache, which subsequently may cause a replacement. When the evicted block has a cache coherence state of M (exclusive, dirty) or O (shared, dirty) state, this also causes a write-back.

Figure 1 shows the percentage increase in the number of *E* (for directory MOESI) and *S* line replacements.  $E \rightarrow I$  transitions – which increased by 2% to 63% – are particularly important since the processor loses the ownership of a block and the ability to silently upgrade its value, potentially significantly increasing the number of invalidations for write upgrades. The number of *S* line replacements account for a significant fraction of the total number of the replacements due to wrong-path load in broadcast SMPs; in directory-based SMPs, they are relatively insignificant.

In Figure 2, we observe that wrong-path reads increase the number of write-backs from 4% to 67%.



**Figure 1** Percentage increase in the number of *replacements* due to wrong-path references in broadcast and directory-based SMPs.



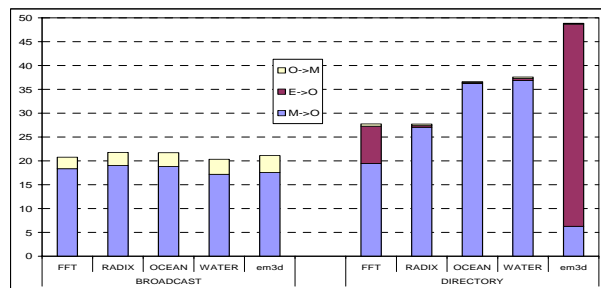
**Figure 2** Percentage increase in the number of *write-backs* due to wrong-path references in broadcast and directory-based SMPs.

### 2.3 Cache Line State Transitions

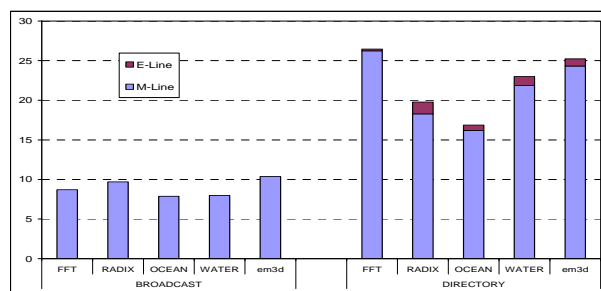
Finally, Figure 3 shows the impact of wrong-path memory references on the number of cache line state transitions. The results show that the number of cache line state transitions increases by 20% to 24% for broadcast SMPs and by 27% to 44% for directory-based SMPs.

An exclusive cache block (modified or clean) loses its ownership when another processor attempts to load that cache block. In order to gain ownership again, that processor has to first invalidate all other copies of that cache block, *i.e.*, Shared  $\rightarrow$  Invalidate for all other processors. In Figure 4, we can see that for broadcast

SMPs, there is 8% to 11% increase in the number of write misses, each of which subsequently causes an invalidation. This percentage is higher, 15% to 26%, for the directory SMPs.



**Figure 3** Percentage increase in the number of cache line *transitions* for MOSI broadcast and MOESI directory SMPs



**Figure 4** Increase in the *write misses* and extra *invalidations* due to wrong-path references.

## 3 Conclusion

In this paper, we evaluate the effects of executing wrong-path memory references on the memory behavior of cache coherent multiprocessor systems. Our evaluation reveals the following conclusions: (1) Modeling wrong-path memory references in a cache coherent shared memory multiprocessor is important and not modeling them may result in wrong design decisions, especially for future systems with longer memory interconnect latencies and processors with larger instruction windows. (2) In general, wrong-path memory references are beneficial because they prefetch data into caches. However, there can be significant amount of pollution caused by these references. (3) For a workload with many cache-to-cache transfers, the coherence actions can be significantly affected by wrong-path memory references.

## References

- [1] O. Mutlu *et al.* Understanding the effects of wrong-path memory references on processor performance. WMPI 2004.
- [2] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In MICRO, pages 165–175, 1996.
- [3] R. Sendag *et al.* Exploiting the prefetching effect provided by executing mispredicted load instructions. Euro-Par 2002.
- [4] R. Sendag *et al.*, The Impact of Incorrectly Speculated Memory Operations in a Multithreaded Architecture, IEEE TPDS, 2005.
- [5] R. Sendag *et al.*, Quantifying and Reducing the Effects of Wrong-Path Memory References in Cache-Coherent Multiprocessor Systems, IPDPS 2006.



## Investigating the Effects of Wrong-Path Memory References in Shared-Memory Multiprocessor Systems

**Ayşe Yilmazer(1)**, Resit Sendag(1), Joshua Yi(2), and Gus Uht(1)

(1) Microarchitecture Research Institute, University of Rhode Island  
(2) Freescale Semiconductor, Inc.

## Motivation

- Wrong-path (WP) effects in Uniprocessors
  - Negative Effects: Pollution
    - L1 and L2 cache pollution
  - Positive Effects: Prefetching
    - Up to 20% better performance for mcf
  - Important to simulate WP for some applications
- No work on WP effects in Multiprocessors
  - In contrast to uniprocessor effects, WP cause:
    - Extra coherence traffic:
      - Data, invalidations, write-backs, acknowledgements
    - Additional cache state transitions

2/3/2006

University of Rhode Island

2 of 16

## Outlines

- Wrong Path Effects on Shared-Memory Multiprocessor Systems (SMPs)
  - Broadcast (snoop-based) and directory-based SMPs
  - Simulation Methodology
- Evaluation Results
- Summary

2/3/2006

University of Rhode Island

3 of 16

## Wrong-path effects on SMPs

- Same issues in uniprocessors apply
  - Pollution effect
    - Evicts blocks needed later by correct-path execution
  - Prefetching effect
    - Brings blocks, which may later be requested down the correct-path, closer to the processor hiding the latency
  - Extra cache/memory traffic
- Besides, extra effects may occur on SMPs
  - Coherence Traffic (extra requests/data communication)

2/3/2006

University of Rhode Island

4 of 16

## Wrong-path effects on SMPs (Cont'd)

### ■ Replacements

A speculatively replaces B

<p><b>1 Initial: P1 writes on block A</b></p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 0</p> <p>LRU: Block B   M</p> </div> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 1</p> <p>Block A   I→M</p> </div> </div> <ul style="list-style-type: none"> <li>• State of block A: Invalid (not present in P0)</li> <li>• State of block B: Modified</li> <li>• A and B maps to the same set</li> <li>• State of block A: Invalid</li> <li>• Event: Write miss</li> <li>• Action: a) Broadcast invalidate b) read cache block c) modify cache block</li> <li>• Next state of A: I→M</li> </ul>	<p><b>2 P0 speculatively reads block A</b></p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 0</p> <p>Block A   I→S</p> </div> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 1</p> <p>Block A   M→O</p> </div> </div> <ul style="list-style-type: none"> <li>• State of block A: Invalid</li> <li>• Event: Request a read-only copy of block A</li> <li>• Action: a) write back block B b) Read cache block A</li> <li>• Next State of A: I→S</li> <li>• State of block A: Modified</li> <li>• Event: snoop hit on read</li> <li>• Action: Forward block A to the requester processor's cache</li> <li>• Next state of A: M→O</li> </ul>
<p><b>3 Speculation Resolves: Mis-speculation !</b></p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 0</p> <p>Block A   S</p> </div> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 1</p> <p>Block A   O</p> </div> </div> <ul style="list-style-type: none"> <li>• State of A: Shared</li> <li>• Speculation resolves in P0</li> <li>• P0 rolls back and continues execution down the correct path</li> <li>• Block A is a wrong path block!!</li> <li>• State of A: Owned</li> <li>• WP effect</li> <li>• It should have been still in M state if there wasn't any WP request by P0</li> </ul>	<p><b>4 P1 writes on block A</b></p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 0</p> <p>Block A   S→I</p> </div> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 1</p> <p>Block A   O→M</p> </div> </div> <ul style="list-style-type: none"> <li>• State of A: Shared</li> <li>• Event: Snoop hit on invalidate</li> <li>• Action: invalidate shared copy of block A</li> <li>• Next state of A: S→I</li> <li>• State of A: Owned</li> <li>• Event: Write miss</li> <li>• Action: a) broadcast Invalidate</li> <li>• Next state of A: O→M</li> </ul>

A is a Wrong-path Block !

University of Rhode Island 5 of 16

## Wrong-path effects on SMPs (Cont'd)

### ■ Write-backs

Write-back dirty copy of B

<p><b>1 Initial: P1 writes on block A</b></p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 0</p> <p>LRU: Block B   M</p> </div> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 1</p> <p>Block A   I→M</p> </div> </div> <ul style="list-style-type: none"> <li>• State of block A: Invalid (not present in P0)</li> <li>• State of block B: Modified</li> <li>• A and B maps to the same set</li> <li>• State of block A: Invalid</li> <li>• Event: Write miss</li> <li>• Action: a) Broadcast invalidate b) read cache block c) modify cache block</li> <li>• Next state of A: I→M</li> </ul>	<p><b>2 P0 speculatively reads block A</b></p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 0</p> <p>Block A   I→S</p> </div> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 1</p> <p>Block A   M→O</p> </div> </div> <ul style="list-style-type: none"> <li>• State of block A: Invalid</li> <li>• Event: Request a read-only copy of block A</li> <li>• Action: a) write back block B b) Read cache block A</li> <li>• Next State of A: I→S</li> <li>• State of block A: Modified</li> <li>• Event: snoop hit on read</li> <li>• Action: Forward block A to the requester processor's cache</li> <li>• Next state of A: M→O</li> </ul>
<p><b>3 Speculation Resolves: Mis-speculation !</b></p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 0</p> <p>Block A   S</p> </div> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 1</p> <p>Block A   O</p> </div> </div> <ul style="list-style-type: none"> <li>• State of A: Shared</li> <li>• Speculation resolves in P0</li> <li>• P0 rolls back and continues execution down the correct path</li> <li>• Block A is a wrong path block!!</li> <li>• State of A: Owned</li> <li>• WP effect</li> <li>• It should have been still in M state if there wasn't any WP request by P0</li> </ul>	<p><b>4 Write-back dirty copy of A Only for MESI (or MSI)</b></p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 0</p> <p>Block A   S→I</p> </div> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 1</p> <p>Block A   O→M</p> </div> </div> <ul style="list-style-type: none"> <li>• State of A: Shared</li> <li>• Event: Snoop hit on invalidate</li> <li>• Action: invalidate shared copy of block A</li> <li>• Next state of A: S→I</li> <li>• State of A: Owned</li> <li>• Event: Write miss</li> <li>• Action: a) broadcast Invalidate</li> <li>• Next state of A: O→M</li> </ul>

Write-back dirty copy of A Only for MESI (or MSI)

University of Rhode Island 6 of 16

## Wrong-path effects on SMPs (Cont'd)

### ■ Invalidations

P1 loses its write privileges for block A

<p><b>1 Initial: P1 writes on block A</b></p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 0</p> <p>LRU: Block B   M</p> </div> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 1</p> <p>Block A   I→M</p> </div> </div> <ul style="list-style-type: none"> <li>• State of block A: Invalid (not present in P0)</li> <li>• State of block B: Modified</li> <li>• A and B maps to the same set</li> <li>• State of block A: Invalid</li> <li>• Event: Write miss</li> <li>• Action: a) Broadcast invalidate b) read cache block c) modify cache block</li> <li>• Next state of A: I→M</li> </ul>	<p><b>2 P0 speculatively reads block A</b></p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 0</p> <p>Block A   I→S</p> </div> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 1</p> <p>Block A   M→O</p> </div> </div> <ul style="list-style-type: none"> <li>• State of block A: Invalid</li> <li>• Event: Request a read-only copy of block A</li> <li>• Action: a) write back block B b) Read cache block A</li> <li>• Next State of A: I→S</li> <li>• State of block A: Modified</li> <li>• Event: snoop hit on read</li> <li>• Action: Forward block A to the requester processor's cache</li> <li>• Next state of A: M→O</li> </ul>
<p><b>3 Speculation Resolves: Mis-speculation !</b></p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 0</p> <p>Block A   S</p> </div> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 1</p> <p>Block A   O</p> </div> </div> <ul style="list-style-type: none"> <li>• State of A: Shared</li> <li>• Speculation resolves in P0</li> <li>• P0 rolls back and continues execution down the correct path</li> <li>• Block A is a wrong path block!!</li> <li>• State of A: Owned</li> <li>• WP effect</li> <li>• It should have been still in M state if there wasn't any WP request by P0</li> </ul>	<p><b>4 P1 writes on block A</b></p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 0</p> <p>Block A   S→I</p> </div> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 1</p> <p>Block A   O→M</p> </div> </div> <ul style="list-style-type: none"> <li>• State of A: Shared</li> <li>• Event: Snoop hit on invalidate</li> <li>• Action: invalidate shared copy of block A</li> <li>• Next state of A: S→I</li> <li>• State of A: Owned</li> <li>• Event: Write miss</li> <li>• Action: a) broadcast Invalidate</li> <li>• Next state of A: O→M</li> </ul>

P1 asks for grant to write and sends invalidation

University of Rhode Island 7 of 16

## Wrong-path effects on SMPs (Cont'd)

### ■ Cache block state transitions

States Change

<p><b>1 Initial: P1 writes on block A</b></p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 0</p> <p>LRU: Block B   M</p> </div> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 1</p> <p>Block A   I→M</p> </div> </div> <ul style="list-style-type: none"> <li>• State of block A: Invalid (not present in P0)</li> <li>• State of block B: Modified</li> <li>• A and B maps to the same set</li> <li>• State of block A: Invalid</li> <li>• Event: Write miss</li> <li>• Action: a) Broadcast invalidate b) read cache block c) modify cache block</li> <li>• Next state of A: I→M</li> </ul>	<p><b>2 P0 speculatively reads block A</b></p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 0</p> <p>Block A   I→S</p> </div> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 1</p> <p>Block A   M→O</p> </div> </div> <ul style="list-style-type: none"> <li>• State of block A: Invalid</li> <li>• Event: Request a read-only copy of block A</li> <li>• Action: a) write back block B b) Read cache block A</li> <li>• Next State of A: I→S</li> <li>• State of block A: Modified</li> <li>• Event: snoop hit on read</li> <li>• Action: Forward block A to the requester processor's cache</li> <li>• Next state of A: M→O</li> </ul>
<p><b>3 Speculation Resolves: Mis-speculation !</b></p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 0</p> <p>Block A   S</p> </div> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 1</p> <p>Block A   O</p> </div> </div> <ul style="list-style-type: none"> <li>• State of A: Shared</li> <li>• Speculation resolves in P0</li> <li>• P0 rolls back and continues execution down the correct path</li> <li>• Block A is a wrong path block!!</li> <li>• State of A: Owned</li> <li>• WP effect</li> <li>• It should have been still in M state if there wasn't any WP request by P0</li> </ul>	<p><b>4 P1 writes on block A</b></p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 0</p> <p>Block A   S→I</p> </div> <div style="border: 1px solid gray; padding: 5px; width: 45%;"> <p>Processor 1</p> <p>Block A   O→M</p> </div> </div> <ul style="list-style-type: none"> <li>• State of A: Shared</li> <li>• Event: Snoop hit on invalidate</li> <li>• Action: invalidate shared copy of block A</li> <li>• Next state of A: S→I</li> <li>• State of A: Owned</li> <li>• Event: Write miss</li> <li>• Action: a) broadcast Invalidate</li> <li>• Next state of A: O→M</li> </ul>

University of Rhode Island 8 of 16

## Wrong-path effects on SMPs (Cont'd)

### ■ Data/Bus and Coherence Traffic Increases

- L1 accesses,
- L2 accesses,
- Coherence traffic
  - Snoop, directory requests for data and invalidations

### ■ Power Consumption Increases

- Due to extra cache accesses, coherence traffic and cache line state transitions

### ■ Resource Contention

- Competing with Correct-path resources
  - Increase in the frequency of full service buffers
    - Critical when many cache-to-cache transfers

2/3/2006

University of Rhode Island

9 of 16

## Simulation Methodology

- GEMS simulator – Wisconsin Multifacet Group
  - Based on Virtutech SIMICS
  - Aggressive out-of-order superscalar processor
  - Detailed Shared-Memory Model
- We evaluate 16-processor SPARC V9 system running unmodified Solaris 9
- Evaluated both Snoop-based MOSI and Directory-based MOESI coherence
  - MOSI: Modified, Owned, Shared, Invalid
  - MOESI: Modified, Owned, Exclusive, Shared, Invalid
- We track the speculatively generated memory references
  - And mark them as being on the wrong-path when the branch misprediction is known

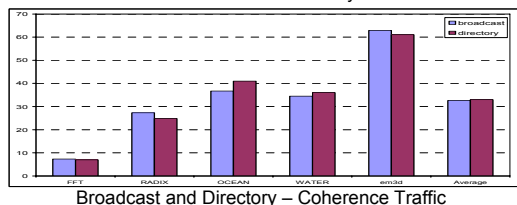
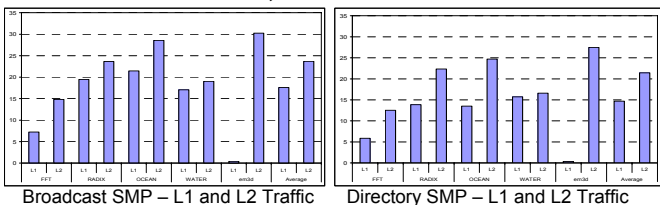
2/3/2006

University of Rhode Island

10 of 16

## Evaluation Results (Cont'd)

-- L1 and L2, and Coherence Traffic



2/3/2006

University of Rhode Island

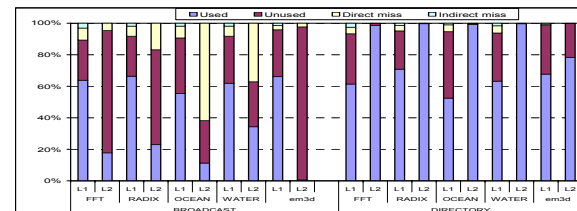
11 of 16

## Evaluation Results (Cont'd)

-- L1 and L2 cache replacements

### 4 Categories:

1. *Unused*: evicted before being used or never used by a correct-path
2. *used*: used by a correct-path reference
3. *direct-miss*: Replaces a cache block that is needed by a later correct-path load, but are evicted before being used.
4. *indirect-miss*: LRU changes in a set may eventually cause correct-path misses.



55-67% L1 and 12-35% L2 repl. Used in Broadcast. However, small number of remote cache misses in directory systems may hurt the performance more.

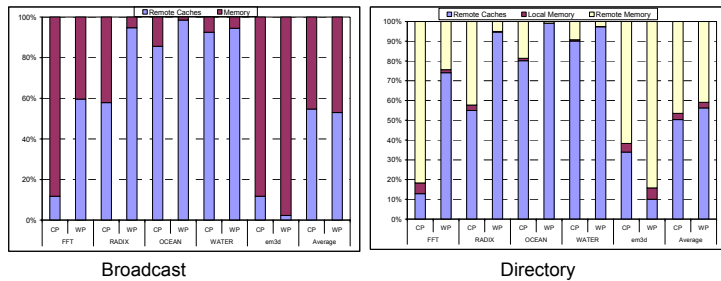
2/3/2006

University of Rhode Island

12 of 16

## Evaluation Results (Cont'd)

### -- Servicing Coherence Transactions



Misses to remote caches 12% to 80% for correct-path and 55% to 96% for wrong-path

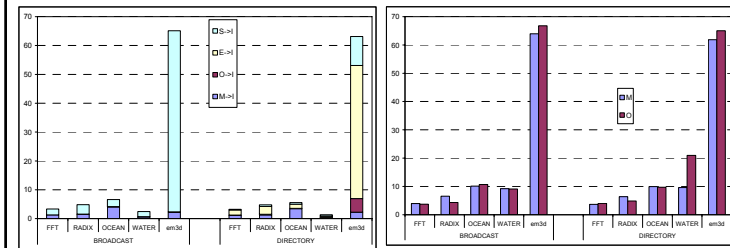
2/3/2006

University of Rhode Island

13 of 16

## Evaluation Results (Cont'd)

### -- Replacements and Write-backs



Replacements

Write-backs

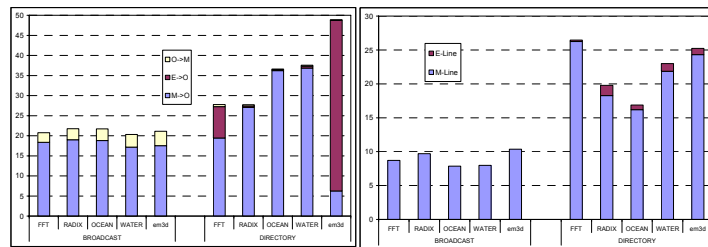
2/3/2006

University of Rhode Island

14 of 16

## Evaluation Results (Cont'd)

### -- Cache Line State Transitions



State Transitions

Write misses and invalidations

2/3/2006

University of Rhode Island

15 of 16

## Summary of Effects

- Uniprocessor effects (i.e., pollution & prefetching) apply. **Moreover,**
- Increase in Coherence Traffic
  - Cache-to-cache transfers by 32%
  - Invalidations by 8% and 20% for broadcast and directory-based SMPs, respectively
  - Write-backs by up to 67% for both systems
- Extra Cache Line State Transitions
  - 21% and 32% for broadcast and directory-based SMPs, respectively

2/3/2006

University of Rhode Island

16 of 16



## Using R-STAGE to Optimize the DASAT Cache System

*M. Tyler Maxwell, Charles C. Weems,  
J. Eliot B. Moss, Robert B. Moll  
{mtmaxwel,moss,weems,moll}@cs.umass.edu  
University of Massachusetts Amherst*

As CPU speeds increase, physical constraints will further limit the size of on-chip cache. Wire-delay and thus the size of on-chip structures becomes more problematic as we move downward to the 65nm process size, eventually requiring a data shrink of on-chip cache.

The DASAT (Dynamically-Aggressive-Spatial Adaptive-Temporal) cache system is a novel L0 data cache architecture with heuristic-guided variable size prefetching and intelligent data promotion. The structure of DASAT simultaneously exploits both temporal and spatial locality, increasing data utilization compared to conventional cache designs.

Conventional caches are structured to exploit a specific balance between the two types of locality. To exploit spatial locality, designers use large atomic access sizes (blocks) so that more data is brought into cache at once. In contrast, using small access blocks exploits temporal locality, because there are more atomic units in the cache at once, reducing the probability of a newly requested block evicting a block about to be accessed again. Because conventional caches have one block size, the designer is required to compromise on block size to attempt to maximize total locality exploitation. DASAT uses two block sizes so that both types of locality can be exploited. In addition, DASAT's variable prefetch mechanism further exploits spatial locality.

This combination of design and dual locality exploitation allows for less data space to be required in order to maintain current hit rates, allowing the size of an on-chip DASAT to be smaller than a conventional cache. With most of the data contained in a DASAT cache accessible in only one CPU cycle, prior work shows that a 9KB DASAT performs just as well as a conventional L0 cache over four times its size.

Determining good design parameters for DASAT is difficult because 1) its parameter space is 9-dimensional and thus large (7,487,690 feasible points), 2) one DASAT simulation takes days on modern hardware, and 3) there is little intuition as to what makes a good parameter set. An exhaustive simulation search of the space would take over 40,000 CPU years for one benchmark, and naïve hill-climbing on point simulation is prohibitive.

In this work we cast DASAT as an optimization problem minimizing estimated AMAT, and use the R-STAGE AI search algorithm to find better points in parameter space. By using a separate feature space, we can learn promising starting points for search. Furthermore, using regression techniques, we can speed search time during hill-climbing. This way we amortize the cost of simulation over many hosts, and are able to find 10-40% eAMAT improvements over baseline for benchmarks in less than two weeks time.

In this talk I present the DASAT cache system, benchmark results on SPEC2000 and SPECJVM98, and discuss the application of R-STAGE onto the DASAT optimization problem.

# Applying R-STAGE to DASAT

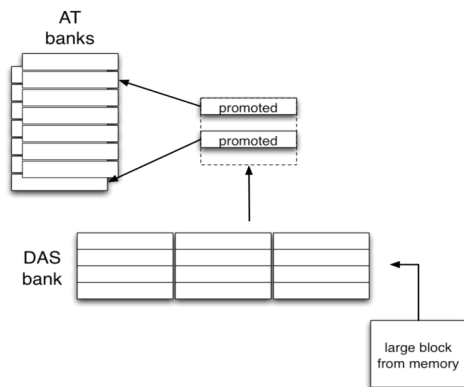
*M Tyler Maxwell*  
University of Massachusetts Amherst

## DASAT

- Dynamically Aggressive Spatial, Adaptive Temporal
- Exploits both spatial and temporal locality
  - Small blocks to exploit temporal locality
  - Large blocks to exploit spatial locality
- Heuristic-driven variable size prefetch
- Hit rates comparable to a conventional cache **4X** its size

2 of 16

## DASAT Structure



3 of 16

## Parameters of DASAT

<b>n</b>	# blocks AT
<b>m</b>	# large blocks DAS
<b>wpb</b>	# words per block
<b>sbplb</b>	# blocks per large block
<b>b1</b>	prediction bound 1
<b>b2</b>	prediction bound 2
<b>b3</b>	prediction bound 3
<b>promo</b>	promotion threshold
<b>hitmax</b>	max value for hit counter

4 of 16

## Parameter Bounds

n	$2^a$	$2 \leq a \leq 17$
m	$2^b$	$2 \leq b \leq 8$
wpb	$2^c$	$0 \leq c \leq 6$
sbplb	$2^d$	$1 \leq d \leq 5$
b1	$b_1$	$b_1 = 0$
b2	$b_2$	$b_1 < b_2 < 15$
b3	$b_3$	$b_2 < b_3 < 20$
promo	p	$0 \leq p \leq 6$
hitmax	$2^e$	$0 \leq e \leq 3$

This space contains 7,487,690 points

5 of 16

## Which Parameters are Best?

- Choose a point that gives best possible performance for (process, benchmark, miss penalty)
- Exhaustive search would take  $\sim 40,000$  CPU years
- Goodness function (eAMAT) is a function of hit rate and DASAT speed

6 of 16

## Computing eAMAT

$$eAMAT = [hitRate * \max(atTime, dasTime)] + [(1 - hitRate) * (atTime + missPenalty)]$$

$$hitRate = S(\bar{P})$$

$$atTime = C_{AT}(\bar{P})$$

$$dasTime = C_{DAS}(\bar{P})$$

$$missPenalty = k$$

7 of 16

## Computing eAMAT (cont.)

- $C_{AT}$  and  $C_{DAS}$  are computed offline by CACTI3.0
- $S$  is computed by trace event simulation, so it is time-intensive
- Define two lengths of simulation
  - $S_f$ : Full (4.6B refs,  $\sim 2$  days)
  - $S_p$ : Partial (500M refs,  $\sim 2$  hours)

8 of 16

## Standard Hill Climbing

$$\text{path } m_i : \overbrace{S(\bar{P}_{starting}) \cdots S \cdots S \cdots S(\bar{P}_{localopt})}^q$$

(where  $\cdots$  is a search of  $i$  neighbors)

$$T_{HC} = m \times qi \times T(S)$$

9 of 16

## Regression as a Heuristic

- We can substitute a regression curve for the hit rate surface (*much* faster)
- Need  $k$  source  $S_p$  points for generated curve
- Can do this in parallel using  $c$  CPUs
- Empirical results show  $k > 150$  approximates DASAT's 9-space
- Hillclimb on regression, then perform simulation

10 of 16

## Regression Hillclimbing

$$\text{path } m_i : \overbrace{R(\bar{P}_{starting}) \cdots R \cdots R \cdots R(\bar{P}_{localopt})}^q \cdots S_p(\bar{P}_{localopt})$$

$$T_{REG-HC} = m \times T(S_p) + \frac{k \times T(S_p)}{c}$$

11 of 16

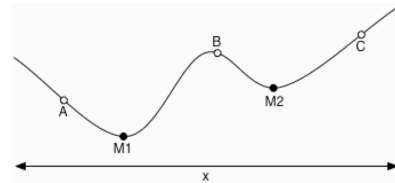
## STAGE

- Motivation: Increase starting point quality and thus decrease necessary  $m$
- Train a feature space that predicts expected maximal goodness of starting at  $P_{start}$
- STAGE works well if search space is patterned

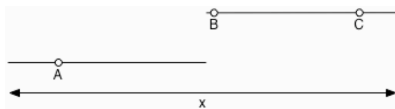
12 of 16

## A Picture of Feature Space

Normal Space



Feature Space



13 of 16

## Applying STAGE

- Define architecture space to be the set of all possible  $P$  points
- Define feature space to be some projection  $P$  into  $V$
- For every path  $m_i$ , train corresponding feature space points on arrived maximal goodness value
- To select a new starting point, hillclimb on feature space

14 of 16

## R-STAGE

- Combines regression curve with  $m$ -reducing STAGE algorithm
- Provides 10-40% eAMAT improvements in several weeks
- More work can be done to further reduce the number of paths  $m$

15 of 16

## eAMAT Results (R-STAGE)

Benchmark	Base (ns)	Opt (ns)	%improv
apsi	1.7519	1.0433	40.4%
bzip2	1.2315	0.9792	20.5%
compress	1.0623	0.9208	13.3%
javac	1.0112	0.8771	13.3%
mpegaudio	1.0453	0.8729	16.5%
wupwise	1.1829	0.9294	21.4%

16 of 16

# Exploring Architectural Challenges in Scalable Underwater Wireless Sensor Networks

Zhijie Jerry Shi\* and Yunsi Fei<sup>+</sup>

\**Department of Computer Science and Engineering, University of Connecticut*

+ *Department of Electrical and Computer Engineering, University of Connecticut*

*371 Fairfield Road, Storrs, CT 06269*

*Email: {zshi, yfei}@enr.uconn.edu*

## Abstract

*The use of sensor networks in aquatic environments has been quite limited, partially due to the harsh underwater environments and associated high system costs. Nevertheless, cost-effective and resource-efficient Underwater Wireless Sensor Networks (UWSNs) will bring significant benefits to numerous aquatic applications. The unique properties of underwater environments raise many new challenges in building an efficient UWSN. We will discuss several architectural challenges and possible research topics in this paper.*

## 1. Introduction

Although sensor networks have been deployed in a broad range of terrestrial applications [1-4], underwater use of sensors and sensor networks has been quite limited due to the harsh environment and associated high costs. Recently, there has been a growing interest in Underwater Wireless Sensor Network (UWSN) due to its advantages and benefits in a wide spectrum of applications in aquatic environments [5-6], such as lakes, ponds, rivers, and oceans. Potential applications of UWSNs include environmental monitoring and data collection, disaster early warning, tactical surveillance, military target detection, unmanned off-shore exploration, and underwater construction.

Compared to land-based wireless sensor networks, UWSNs have two unique properties: acoustic communication channels and the mobility of sensor nodes. Since radio frequency (RF) signals do not propagate well in water, acoustic channels are taken as the sole means for communications among underwater sensor nodes. Compared to RF signals, acoustic signals have much longer latencies (five orders of magnitude) and lower bandwidths. The second property is passive mobility of sensor nodes, which results in dynamic networking structure. Empirical observations suggest

underwater sensor nodes will move at the speed of 3-6 kilometers per hour with an effective diffusivity of from  $10^{-3}$  to  $10^3$   $\text{cm}^2/\text{s}$  in the vertical and from  $10^{-3}$  to  $10^5$   $\text{cm}^2/\text{s}$  in the horizontal. To ensure reliable and efficient data transmitting and forwarding, UWSNs must identify the sensor nodes' locations periodically, and adapt the network configuration accordingly.

The aforementioned properties of UWSNs raise special challenges that affect all essential components of UWSNs, including communication mechanisms, networking protocols, sensor nodes, and resource management. It is necessary to revisit various aspects of sensor node design and consider all the features in UWSNs to build a better optimized system in the new underwater environment.

The rest of the paper is organized as follows. We will describe some architectural challenges in UWSN designs in Section 2, and summarize in Section 3.

## 2. Challenges in UWSN system designs

### 2.1. Workload characterization

A typical underwater sensor node consists of a sensor probe, an acoustic modem, a controller, storage, battery, and an interface circuitry that connects all the components with the controller. Although the structure is similar to that of land-based sensor nodes, underwater sensor nodes need different designs of their components because of the unique properties of underwater environments and the distinct requirements of aquatic applications. For example, high-precision localization algorithms, needed for underwater sensor nodes to calculate their positions, impose heavy workloads on microcontrollers. As for the applications, long-term environment monitoring require little computational capability, large memory, and a long operation time; while short-term target detection applications demand more computational capability and real-time response. When building underwater sensor nodes, the first thing we need to study is the new workloads of UWSNs. It is

also important to investigate the impact on sensor node design of the different application requirements and to seek a common architecture that meets the requirements.

## 2.2. Energy-efficient node design and resource management

One primary challenge of deploying a dense, distributed, and scalable UWSN is the limited energy resources on individual sensor nodes. Power and energy optimizations are especially critical for UWSNs because 1) acoustic communications will consume more energy than RF channels, and 2) energy harvesting is much more difficult because major harvesting sources such as solar and wind energy are not available in the underwater environment.

On the one hand, UWSNs face new challenges to achieving long operation times. For example, due to the dynamic nature of UWSNs, network configuration algorithms have to run periodically and, very often, at a rate not slower than data sampling rate. Thus, energy-efficient configuration algorithms and scheduler need to be designed. On the other hand, the underwater environment may facilitate some power-saving mechanisms. For example, the microcontroller in a sensor node can go to the sleep mode during idle time and wake up to the active mode when necessary. Since the wakeup process takes time, the node shall be put in the sleep mode only when the idle time is long enough reaching a breaking point. Due to the long delay of acoustic communications, underwater sensor nodes may enter the sleep mode more frequently.

To maximize the lifetime of UWSNs, we need to achieve energy efficiency at both the sensor node and network levels. Within each sensor node, the methods include proper assignments of tasks to components and the energy-efficient implementations of the components. At the network level, it is critical to manage resource efficiently. The possible solutions include strategies that balance the trade-offs between energy consumptions and other metrics. For example, network routing protocols can trade energy for robustness. For applications like target detection, it can be power-saving to perform preliminary computations among *local* nodes that are close to each other and to transmit only useful data or results back to the processing center.

## 2.3. Lifetime estimation

The lifetime of UWSNs is one of the main design goals. Different underwater applications may have different requirements on lifetime. Accordingly, these requirements give us guidelines to determine certain design parameters, such as density of sensor nodes and battery capacity, at the design time.

Estimating the lifetime of sensor networks prior to the design and deployment of an actual network requires an analytical method and model which coarsely captures the behavior of underwater sensor networks. The model should be generic and parameterized, combining both the physical medium-specific (acoustic communications) and networking aspects in UWSN. Conversely, after the energy estimation model is derived, power optimization consideration can be woven into the design of network protocols (such as medium access control, routing, and transporting), topology (such as the distance between nodes, number of nodes in a cluster, number of tiers in the hierarchical networking architecture), and working parameters (such as acoustic frequency, data query frequency, etc.) in order to materialize a cost-effective and energy-efficient USWN.

## 3. Summary

In this paper, we briefly examined the architectural and system design issues for UWSNs. We believe that the low-power design and efficient resource management will remain the major challenges for UWSN designs. The unique features of UWSNs raise new challenges, but they also create opportunities to explore power-saving mechanisms. In order to extend the lifetime of UWSNs, power efficiency should be pursued at both the node and network levels. Moreover, an accurate lifetime estimation model is necessary for UWSN designs and can help determine important design parameters to prolong the network lifetime.

## References

- [1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proc. Architectural Support for Programming Languages & Operating Systems*, pp. 93-104, November 2000.
- [2] J. Rabaey, M. J. Ammer, J. L. DaSilva, D. Patel, and S. Roundy, "PicoRadio supports ad hoc ultra-Low power wireless networking," *IEEE Computer*, vol. 33, no.7, pp. 42-48, July 2000.
- [3] A. Chen, R. Muntz, S. Yuen, I. Locher, S. Sung, and M. Srivastava, "A Support infrastructure for the smart kindergarten," *IEEE Pervasive Computing Magazine*, vol. 1, no. 2, pp. 49-57, 2002.
- [4] G. Pottie and W. Kaiser, "Wireless integrated network sensors," *Communications of the ACM*, vol. 43, pp. 51-58, May 2000.
- [5] I. F. Akyildiz, D. Pompili, and T. Melodia, "Challenges for efficient communication in underwater acoustic sensor networks," *ACM Sigbed Review*, vol. 1, no. 2, July 2004.
- [6] D. Pompili and T. Melodia, "Three-dimensional routing in underwater sensor networks," in *Proc. ACM Int. Wkshp on Performance Evaluation of WASUN*, pp. 214-221, October 2005.

## Exploring Architectural Challenges in Scalable Underwater Wireless Sensor Networks



Z. Jerry Shi and Yunsi Fei  
 Department of Computer Science and Engineering  
 Department of Electrical and Computer Engineering  
 University of Connecticut

BARC, February 3, 2006

### Outline

- Motivation of underwater wireless sensor networks (UWSNs): Aquatic applications
- Design challenges in UWSNs
  - Communications
  - Networking algorithms/protocols
- Architectural issues in UWSNs
  - Workload characterization
  - Energy-efficient design and resource management
  - Lifetime estimation
- Summary

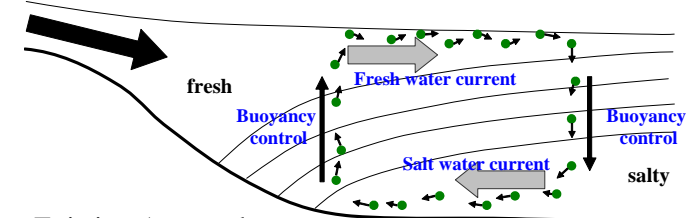
2 of 12

### Underwater wireless sensor networks: application-driven

- Environmental monitoring and data collection
  - Temperature, salinity, ocean currents, etc.
    - Influences on climate and living conditions of plants and animals
  - Marine microorganism
  - Pollution
- Disaster early warning and prevention
  - Seismic monitoring
  - Tsunami
- Off-shore exploration and underwater construction
- Coastline protection and tactical surveillance
- Target detection
  - Mine
  - Shipwreck

3 of 12

### Application example: estuary monitoring



- Existing Approaches
  - Ship tethered with chains of sensors moves from one end to the other
  - Cons: no 4D data, either  $f(x, y, z, \text{fixed } t)$ , or  $f(\text{fixed } (x, y, z), t)$ ; and high cost
- Using UWSN
  - Easily get 4D data,  $f(x, y, z, t)$ , mobile sensors
  - Reduce cost significantly
  - Increase coverage
  - Have high reusability

4 of 12

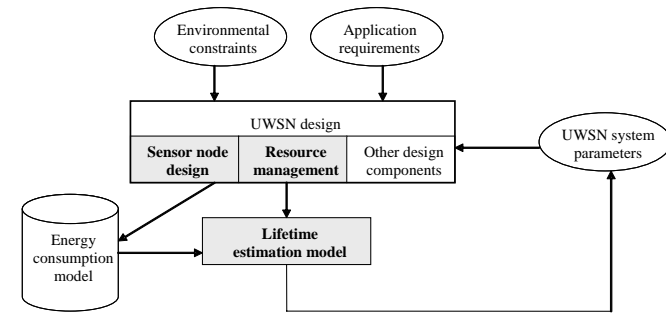


## Challenges in different aspects of UWSNs

- Communications
  - Radio does not work well in water
    - 120cm at 433 MHz reported at USC
    - Low frequency → large antennae and high transmission power
  - Acoustic channels adopted
    - Limited bandwidth: Bandwidth × Range product = 40 kbps-km
    - Long delay:  $1.48 \times 10^3$  m/s vs.  $3 \times 10^8$  m/s
    - High bit error rates
    - Multi-path and fading problems
- Networking
  - Medium access control: **high channel utilization**
  - 3-D networking, geographical-based routing: **robust to dynamic topology**
  - Data transfer: **reliable and high throughput**
  - Localization & time synchronization : **GPS-free**
  - Robustness: **resilient to network disconnection**

5 of 12

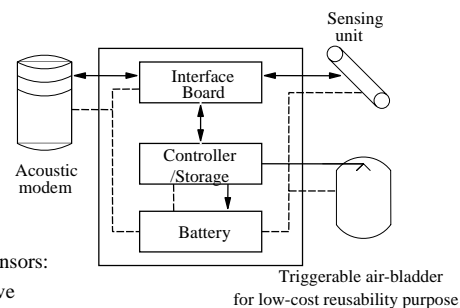
## System design of UWSNs



6 of 12

## Typical structure of a sensor node

- Sensor probes
  - Interface circuitry
- Controller (processors)
- Trans-receiver
  - Acoustic modem
- Storage
- Battery
- Triggerable air-bladder



Different from land-based sensors:

- Larger and more expensive
- More power hungry
- Prone to failures

7 of 12

## Goals of underwater sensor nodes

- Easy to customize for different applications: workload characterization
  - Satisfying performance
    - Computing capacity
  - Storage
  - Bandwidth
- Long operation time: low power
  - Energy becomes more critical
    - Acoustic communications, memory, air-bladder, etc., more power-hungry
    - Energy harvesting difficult: solar and wind energy are not available
- Reliable operations
- Low cost: allows deployment of large amounts of nodes
  - Decomposable or retrievable

8 of 12

### Energy-efficient design at the node level

---

- Design choices: ASIC, ASIP, FPGA, microcontroller
- Power-efficient design of individual components
  - Acoustic communication modules
  - Flexible packet relaying circuit
    - Only wake up the microcontroller when needed
- Proper task assignments and scheduling
  - Sampling, processing, storing, transmitting, receiving, and forwarding
- Exploiting opportunities in the underwater environment
  - Long and frequent sleep mode due to the long delay of acoustic channels

9 of 12

### Power management at the network level

---

- Power-aware routing algorithms
  - Short-range vs. long-range communications
  - Reliability vs. energy trade-offs
- Power-aware localization algorithms
  - Accuracy vs. energy trade-offs
- Configuration strategy
  - Choosing working parameters adaptively in the field
- In-network computations
  - Utilizing short-range one-hop communications
  - Balance the power consumption of nodes located in different areas

10 of 12

### Lifetime estimation model

---

- Impact of network design parameters on power consumption
  - Average one-hop signal transmission distance
  - Data transmission period
  - Acoustic channel frequency
  - Network topology (3-D, distances, clustering, etc.)
  - Sensor lifetime
- Simulation of UWSNs
  - Hierarchical energy model
  - Output: statistic information, e.g., data communication throughput, retransmission rate, data drop rate, average power consumption, and sensor network lifetime.

11 of 12

### Conclusions

---

- Opportunities: interesting and promising area
  - Requires interdisciplinary collaborations
- Challenges: a lot of new challenges, especially in resource management and energy-efficient system design – a cross-layer effort!

12 of 12

# ILP is Dead, Long Live IPC!

(A position paper.)

Augustus K. Uht  
Microarchitecture Research Institute  
Dept. of Electrical and Computer Engineering  
University of Rhode Island  
February 3, 2006

## Overview: the State of the Art

We will discuss the current state of microprocessor architecture, where it is currently headed, and where it should be headed. Specifically, until recently processors consisted of one copy of a CPU, the latter exploiting as much Instruction-Level Parallelism as possible. This improved performance.

Unfortunately, two trends collided and caused a rapid shift in processor architecture: 1) The architecture community's ability to extract ILP from typical code asymptotically approached zero, so processor companies kept increasing the CPU clock rate to compensate and improve performance in a brute force way. Marketing also played a large role here: it's relatively easy to sell processors based on a single {higher} number {clock frequency}. 2) Processor temperatures were becoming excessive. The newer processors from Intel (Prescott Pentium 4's) have power dissipations measurable in light bulb or toaster equivalents.

The net result was the almost-appearance of the ill-fated 4.0 GHz Pentium 4. It could not be reliably sold or used due to its large power dissipation; in short, it would burn up. The industry 'fix' to this problem is to put two or more CPU's on a chip or package ('multi-core' processors) and run them at lower speeds, thereby reducing power to acceptable levels while increasing performance. Is there a fallacy here? Is this the right way to go?

## Multicore Performance

Looking at Figure 1, a chart from a recent Intel talk given by Benson Inkley [1], the two processors being compared are a single-core 3.73 GHz Pentium 4 vs. a dual-core Pentium 4 –based 3.2 GHz 840 processor. Both processors use dual-threading (Hyperthreading). The performance numbers are based on the execution of the SPEC 2000 benchmarks, both integer and floating point. We will

focus on the left-most comparison: the processors running the SPECint2000 benchmarks without tuning.

We see that by doubling the number of cores, performance increases by only 18%. While the single-core processor is 'faster' than the individual cores of the 840 processor (3.73 vs. 3.2 GHz), on a cycle-by-cycle basis one would expect a performance gain of 72%.

Examining the power requirements of the two processors, from the datasheets [2, 3] we see that the total suggested design power (NOT the peak) for the single-core processor is: 115 W, and 125 W for the dual-core processor. However, the Intel-supplied fan runs at 24 W.

Therefore, for about the same power dissipation, and about twice the cost, we get a slight performance increase of about 18%.

I have been unable to determine the specifics of the experiments. For example, were the SPECint benchmarks initially recoded to take advantage of the multithreading and dual core options? This seems likely. In a more realistic scenario, a shrink-wrapped program, unable to be recompiled, would likely have experienced a performance *decrease* when going to the dual-core processor, since each core runs at a lower speed than the 3.73 single-processor.

The conclusion is that for normal, nominally sequential programs, multi-core processing is a 'lose.' (Of course, if a program can be recompiled, and it contains lots of parallelism, like many media applications, then there will likely be a performance benefit from multi-core processors, but not necessarily proportionally; see the right-hand side of Figure 1.)

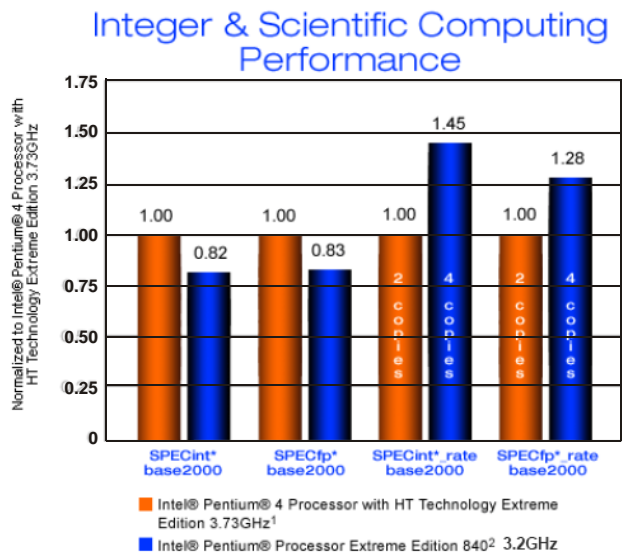


Figure 1. Intel single vs. multi-core performance [1].

## Parallelizing Compilers

In its current state, the hardware community can not make progress in improving the performance of the vast majority of existing or future programs. As currently envisioned, multi-core processors are not useful. However, industry has thrown the problem over the wall to the programmers, and said: 'You fix it. You MUST learn how to program in parallel.'

While dictating technological progress can sometimes have a positive effect, it is unlikely in this case. Look at the supercomputing literature: for the past thirty to forty years researchers have attempted to write auto-parallelizing compilers, with only modest success. The community has turned to such tools as OpenMP or assertions, both of which are very hard to use and port. Also, the use of assertions can easily lead to functionally wrong code.

Being realistic, we all have a hard time writing *sequential* programs, much less *parallel* programs. This is true for novice and expert programmers alike [4].

But we still want to improve performance. Is there a solution? I think so: '*Long Live IPC.*'

## Back to the Future

Whatever happened to ILP? Many, many limit studies have shown large amounts of ILP (potential parallelism) in typical programs, even `gcc` [5]. But architects have been unable to realize the potential performance in IPC (realized parallelism).

ILP exploitation is hard, no doubt about it. But computer architects have solved tougher problems.

For many years, the classic superscalar architecture has become a de facto standard. No serious deviations from its microarchitecture are allowed. Little changed, little gained.

We need to wipe the slate clean, and create dramatically new microarchitectures in order to make significant gains. This is generally frowned upon by industry, which doesn't like big changes. (But recall, the Intel P6 microarchitecture was a radical change, and it paid off big. Intel has even returned to it: the mobile Pentium M processor is based on the P6.) The results of this industry bias are a slew of incremental performance improvements.

Some of us are starting fresh, e.g., the TRIPS machine [6] and the Levo machine [7]. Both have yielded IPC's (not ILP) greater than three and five (resp.) with realistic simulation assumptions. (TRIPS requires compiler support, Levo does not.)

But this is just the start. Power is still an issue. We must get away from the frame-of-mind that a microprocessor must use as many transistors as

possible. On the contrary, it should use as few transistors as possible. (Sounds obvious, but we seem to have forgotten this.)

We must also re-examine the multi-core model in even its most basic sense. Forget about duplicating entire processors. Remember, we can't program the end result. Think of using less complex and less costly computation units.

## Conclusions

Everyone has fallen in behind the microprocessor manufacturers in the multi-core futility, even the major operating systems' and applications' programmers. Further, (at least) the latter say they won't be able to do anything for years [4].

'Those who cannot remember the past are condemned to repeat it.'

Let's not waste another 30-40 years. If there was ever a time to think out-of-the-box, this is it. Let's do some real envelope-pushing. Multi-core machines (the same as multiprocessors) and parallelizing compilers are well known and are not likely to produce any meaningful new results.

We need IPC; we don't need PPC (processors per cycle).

## References

- [1] B. Inkley, "Intel Multi-core Architecture and Implementation." Hillsboro, OR: Intel Corp., 2005. Talk given at GSPx.
- [2] Intel Staff, "Intel Pentium Processor Extreme Edition 840," Intel Corp., Datasheet 306831-002, October 2005.
- [3] Intel Staff, "Thermal Management for Boxed Intel Pentium 4 Processor in the 775-land Package," Intel Corp., 2005. URL: [http://www.intel.com/cd/channel/reseller/asm-na/eng/products/box\\_processors/desktop/proc\\_dsk\\_p4/technical\\_reference/99346.htm](http://www.intel.com/cd/channel/reseller/asm-na/eng/products/box_processors/desktop/proc_dsk_p4/technical_reference/99346.htm).
- [4] K. Krewell, "Software Grapples With Multicore," *Microprocessor Report*, December 12, 2005.
- [5] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*. Gold Coast, Australia: IEEE and ACM, May 1992, pp. 46-57.
- [6] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*. San Diego, California, USA: ACM and IEEE, June 9-11 2003.
- [7] A. K. Uht, D. Morano, A. Khalafi, and D. R. Kaeli, "Levo - A Scalable Processor With High IPC," *The Journal of Instruction-Level Parallelism*, vol. 5, August 2003. (<http://www.jilp.org/vol5>), URL: <http://www.ele.uri.edu/~uht/papers/JILP2003FnlLevo.pdf>.

# ILP is Dead, Long Live IPC!

(A position paper.)

Augustus K. Uht (aka Gus Uht)  
Dept. of Electrical and Computer Engineering



Copyright © 2005-2006, A. K. Uht

BARC: February 3, 2006



## Outline

1. ILP vs. IPC
2. State of the Art; two nasty trends:
  - a) Conventional wisdom: ‘...no more exploitable ILP...’  
or: **‘ILP is Dead’**
  - b) Power is killing us – CPU power in light-bulb equiv.
3. Industry/Academia Response:
  - a) Dual/multi-core chips (multiprocessors)
  - b) Does this buy us anything?
4. Solution: **‘Long Live IPC!’**

BARC: February 3, 2006 ILP/IPC

2 of 8



## 1. ILP and IPC Defined

- **ILP = Instruction-Level Parallelism:**  
**Potential parallelism:** what is in the code.  
(Assumption: unlimited resources.)
- **IPC = Instructions Per Cycle:**  
**Realized parallelism:**  
what the hardware really gets.  
(Resources limited.)
- Overall ‘performance’ = IPC \* clock frequency,  
in Instructions Per Second.

BARC: February 3, 2006 ILP/IPC

3 of 8




## 2. State of the Art

- I. Conventional wisdom: **‘ILP is Dead’**
  - a) Given: ‘Limit studies’ show **ILP in 10’s** (even **gcc**)
  - b) In reality, hardware rarely gets  $IPC > 1$ , **BECAUSE:**
    - i. Conservative research:  
Need to stick to std. CPU model if you want to get published.
    - ii. Industry cautious: doesn’t like big changes
- II. Power has become excessive, e.g.:  
Intel 4.0 GHz Pentium 4:  
Heat death of the uni-processor-verse  
→ Industry sol’n:  $n$ -core chips:  $n$  lower-freq. CPUs, e.g.:  
**dual perf.  $\geq$  solo perf., & dual pwr.  $\leq$  solo pwr.**  
**(Oh, really???)**

BARC: February 3, 2006 ILP/IPC

4 of 8

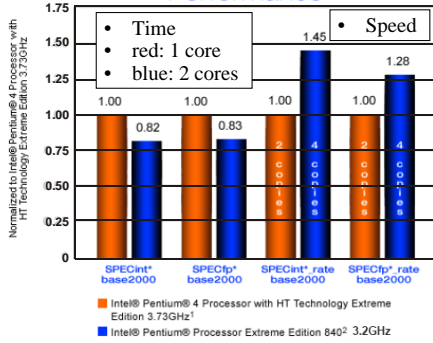


## Multicore Problem 1: (non)-Performance

---

- Time, 1 to 2 cores:
  - For same power,
  - Twice the cost,
  - Get 18% perf. increase.
  - Likely perf. would actually *decrease*
- Speed, 1 to 2 cores:
  - < 50% perf. gain
  - Code copies used

### Integer & Scientific Computing Performance




Benchmark	1 Core (Time)	2 Cores (Time)	1 Core (Speed)	2 Cores (Speed)
SPECint*_rate_base2000	1.00	0.82	1.00	0.83
SPECfp*_rate_base2000	1.00	0.83	1.00	0.83
SPECint*_rate_base2000	1.00	1.45	1.00	1.45
SPECfp*_rate_base2000	1.00	1.28	1.00	1.28

■ Intel® Pentium® 4 Processor with HT Technology Extreme Edition 3.73GHz<sup>1</sup>  
 ■ Intel® Pentium® Processor Extreme Edition 840<sup>2</sup> 3.2GHz

© Intel Corp.

BARC: February 3, 2006
ILP/IPC
5 of 8




## Multicore Problem 2: Can't Program 'em

---

- Chief of some chip company:  
*[Programmers will **have** to learn how to write parallel programs.] !!!*
- ...and pigs have wings, to wit:
- Since ~1964:
  - Tried to build auto-parallelizing compilers. *NG.*
  - Tried to make parallel programming easy. *No cigar.*
- We don't know how to write good sequential programs; now good PARALLEL programs?

BARC: February 3, 2006
ILP/IPC
6 of 8




## Solution: Back to the Future

---

'Long Live IPC!'

- Recall: community is stuck on old model.
- Radically new models are promising; examples:
  - TRIPS (IPC ~3, needs compiler support)
  - Levo (IPC ~5, with legacy binaries)
 (Remember: the P6 was radical too, way back when.)
- Don't use as many transistors as possible.  
→ power less of an issue.
- Multicore processors? Maybe, but used differently.
  - Use simple cores:  
Cores do not need to be complete or standard processors.
  - Eliminate basic cross-chip communications.

BARC: February 3, 2006
ILP/IPC
7 of 8



## Summary

---

'Those who cannot remember the past are condemned to repeat it.'

Parallel programming is a losing proposition for most/all programs, even scientific (time-to-solution).

(But: *APL*, anyone?)

We need IPC!

(We don't need PPC [processors per cycle]...)

BARC: February 3, 2006
ILP/IPC
8 of 8

11000010 11000001 01010010 01000011 00110010 10110000 10110000 10110110

# Keypanel Session



## Fourth Annual Boston Area Architecture Workshop Keypanel Session

---

### Description:

The **Keypanel Session** combines the best features of keynote speeches and panel sessions. The end user, industry, and academic communities will each be represented by one keypanelist. Three 15-minute keypanelist presentations will be followed by one 15-minute audience Q & A session. The order of the keypanelists' presentations will be randomly determined during the session.

---

### General Topic: *Whence goeth the microprocessor?*

**Specific questions** to each keypanelist representative:

- \* to **End Users**: What does the commercial user want today? In 5 years? In 10 years?
  - \* to **Industry**: What will/should industry provide today? In 5 years? In 10 years?
  - \* to **Academia**: What will/should academia be looking at today? In 5 years? In 10 years?
- 

### The Keypanelists:

We are extremely fortunate to have the following practitioners and researchers representing:

- \* **End Users**: [Dr. Atul Chhabra](#), Verizon Corp., Enterprise Architect and Senior IT Manager.
  - \* **Industry**: [Dr. Joel Emer](#), Intel Corp., Intel Fellow.
  - \* **Academia**: [Prof. Anant Agarwal](#), MIT, Professor of Electrical Engineering and Computer Science.
- 

**Keypanelist Biographies:** (see next page)

---

---

## Keypanelist Biographies:

### \* Dr. Atul Chhabra

Dr. Chhabra is Enterprise Architect and Senior IT Manager at Verizon Communications Corp., a Fortune 14 company with over 200,000 employees and local, national and international communications responsibilities. He received his Ph.D. from the University of Cincinnati in 1990, and his B.Tech., Indian Institute of Technology, New Delhi, India in 1984. Dr. Chhabra has over fifteen years of IT, product development, e-business, and R&D experience at Verizon and its former companies. He has developed and managed the architecture for enterprise content management systems, enterprise portals as personalized employee desktops, integration of enterprise resource planning systems, and systems for managing internal controls. He has performed capacity planning, performance modeling, and evaluation of several enterprise systems. In the mid to late 90's, Dr. Chhabra led Verizon's research into automated recognition and interpretation of scanned images of engineering drawings and benchmarking of the available methods. At the same time, he managed the process of scanning about half a million network drawings and the automated extraction of key information from the drawings.

### \* Dr. Joel Emer

Dr. Emer is an Intel Fellow, Digital Enterprise Group, and Director of Microarchitecture Research. Dr. Emer received his bachelor's and master's degrees in electrical engineering from Purdue University in 1974 and 1975, respectively. He earned a doctorate in electrical engineering from the University of Illinois in 1979. Dr. Emer has worked in industry for over 25 years, spending much of his career at Digital Equipment Corp. (DEC), and then Compaq, where he was Director of Alpha Architecture Research. He was responsible for innovations in almost every aspect of micro-architecture for several generations of Alpha processors, widely considered to be the highest-performing processors of their time. As part of this work and in his earlier work on several generations of VAXes, Dr. Emer stressed the then-unusual quantitative approach to performance analysis. In recent years he was a pioneer in the research and implementation of simultaneous multithreading in a commercial processor. Dr. Emer is both an IEEE Fellow and an ACM Fellow.

### \* Prof. Anant Agarwal

Prof. Agarwal is a member of the EECS department at MIT, and is also a member of MIT's Computer Science and Artificial Intelligence Laboratory (CSAIL). Anant Agarwal earned a Ph.D. in 1987 and an MS in Electrical Engineering both from Stanford University. He got his bachelor's degree in Electrical Engineering from IIT Madras in 1982. His teaching and research interests include VLSI, computer architecture, compilation, and software systems. He is the CTO and founder of Tiler Corp., the *n*th startup company of his career. Prof. Agarwal has led many well-known prototyped radically-novel architecture projects, including Alewife, VirtualWires and the current RAW machine project.

---

11000010 11000001 01010010 01000011 00110010 10110000 10110000 10110110



# Branch Trace Compression for Snapshot-Based Simulation\*

Kenneth C. Barr and Krste Asanović

MIT Computer Science and  
Artificial Intelligence Lab

32 Vassar St., Cambridge, MA 02139

{kbarr, krste}@csail.mit.edu

## 1. Introduction

As full-system simulation of commercial workloads becomes more popular, methodologies such as statistical simulation and phase detection have been proposed to produce reliable performance analysis in a small amount of time. With such techniques, the bulk of simulation time is spent fast-forwarding the simulator to relevant points in a program rather than performing detailed cycle-accurate simulation. To amortize the cost of lengthy fast-forwarding, snapshots can be captured at each sample point and used later to initialize different machine configurations without repeating the fast-forwarding.

If the snapshot contains just register file contents and memory state, one must perform lengthy “detailed warming” of caches and branch predictors to avoid cold-start bias in the results. Microarchitectural state can also be captured in the snapshot, but this will then require regeneration of the snapshot every time a microarchitectural feature is modified.

For caches, various microarchitecture-independent snapshot schemes have been proposed, which take advantage of the simple mapping of memory addresses to cache sets. Branch predictors, however, are much more difficult to handle in the same way, as they commonly involve branch history in the set indexing function which smears the effect of a single branch address across many locations in a branch predictor. One possibility is to store microarchitectural state snapshots for a set of potential branch predictors, but this limits flexibility and increases snapshot size, particularly when many samples are taken of a long-running multiprocessor application.

We explore an alternative approach in this paper, which is to store a compressed version of the complete branch trace in the snapshot. This approach is microarchitecture-independent because any branch predictor can be initialized before detailed simulation begins by uncompressing and replaying the branch trace.

The main contribution of our paper is a branch predictor-based compression scheme (BPC), which exploits software branch predictors in the compressor and

decompressor to reduce the size of the compressed branch trace snapshot. When BPC is used, the snapshot library can require less space than one which stores just a single concrete predictor configuration, *and* it allows us to simulate *any* sort of branch predictor.

## 2. Design

BPC uses a collection of *internal predictors* to create an accurate, adaptive model of branch behavior. A software branch predictor has two obvious advantages over a hardware predictor. First, the severe constraints that usually apply to branch prediction table sizes disappear; second, a fast functional simulator can provide oracle information to the predictor such as computed branch targets and directions. When the model correctly predicts many branches in a row, those branches need not be emitted by the compressor; instead, it concisely indicates the fact that the information is contained in the model.

The output of the compressor is a list of pairs. The first element indicates the *skip amount*, the number of correct predictions that can be made beginning with a given branch; the second element contains the data for the *branch record* that cannot be predicted. We store the output in two separate files and use a general-purpose compressor called PPMd [3] to catch patterns that we have missed and to encode the reduced set of symbols.

The decompressor reads from these files and outputs the original branch trace. After reversing the general-purpose compression, the decompressor first reads from the *skip amount* file. A positive skip amount,  $x$ , indicates that BPC’s internal predictors are sufficient to produce  $x$  correct branch records in a row. When  $x = 0$ , the unpredictable branch may be found in the *branch record* file. As the decompressor updates its internal predictors using the same rules as the compressor, the state matches at every branch, and the decompressor is guaranteed to produce correct predictions during the indicated skip intervals.

We define a *concrete branch predictor* to be a predictor with certain fixed parameters such as size of global history, number of branch target buffer entries, etc. To measure the performance of various concrete branch predictors, the output of the BPC Decompressor is used to update state in each concrete predictor according to the update policies of that predictor.

## 3. Evaluation

To evaluate BPC, we use the 20 traces from the Championship Branch Prediction (CBP) competition [4]. The trace suite comprises four categories: integer, floating point, server, and multimedia. Traces contain approximately 30 million instructions compris-

\*A full-length version of this paper appears in ISPASS 2006 [1]

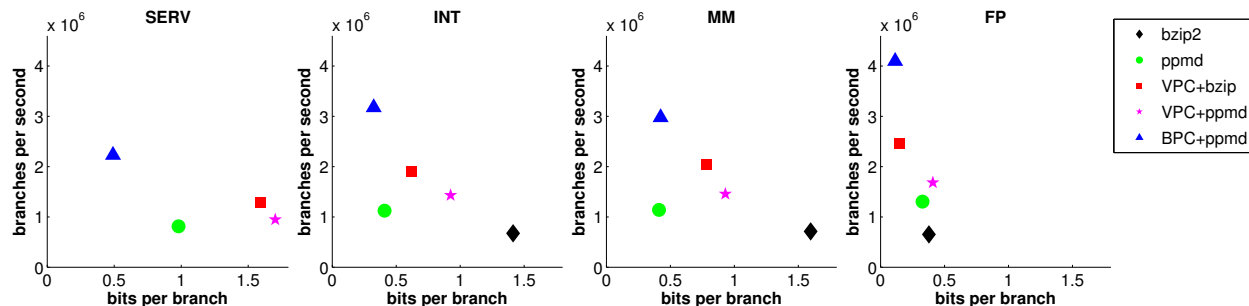


Figure 1. The optimal compressed trace format is in the upper left of each plot. Decompression speed across applications is reported with harmonic mean. The times were collected on a Pentium 4 running at 3 GHz.

ing both user and system activity and exhibiting a wide range of traits in terms of branch frequency and predictability.

The state of a given branch predictor (a *concrete snapshot* in our terminology) has constant size of  $q$  bytes. However, to have  $m$  predictors warmed-up at each of  $n$  detailed sample points (multiple short samples are desired to capture whole-program behavior), one must store  $mn$   $q$ -byte snapshots. Concrete snapshots are hard to compress so  $p$ , the size of  $q$  after compression, is roughly constant across snapshots. Since a snapshot is needed for every sample period, we consider the cumulative snapshot size:  $mnp$ . In our experiments, cumulative snapshots grow *faster* than a BPC-compressed branch trace even for reasonable  $p$  and  $m = 1$ .

On average, BPC+PPMd provides a  $3.4\times$ ,  $2.9\times$ , and  $2.7\times$  savings over a concrete snapshot compressed with gzip, bzip2, and PPMd respectively. When broken down by workload, the savings of BPC+PPMd over concrete+PPMd ranges from  $2.0\times$  (integer) to  $5.6\times$  (floating point). Note that this represents the *lower bound* of savings with BPC: if one wishes to study  $m$  branch predictors of size  $P = \sum_{i=1}^m p_i$ , the size of the concrete snapshot will grow with  $mnP$ , while the BPC trace supports any set of predictors at its current size.

Figure 1 summarizes space and time results of our experiments with one plot for each application category. The most desirable techniques, those that decompress quickly and yield small file sizes, appear in the upper left. For each application domain, BPC+PPMd performs the fastest. In terms of bits-per-branch, BPC+PPMd is similar to VPC [2] for highly-compressible floating point traces and similar to PPMd for integer benchmarks. For multimedia, PPMd compresses best, while BPC+PPMd performs significantly better than all its peers for hard-to-predict server benchmarks. BPC decompression also outpaces fast functional simulation (not shown). High speed and small files across application domains are the strengths of our technique.

## 4. Related work

Value-predictor based compression (VPC) is a recent advance in trace compression [2]. Its underlying predictors are more general than BPC’s branch direction and target predictors. With its specialized predictors and focus on chains of correct predictions, we have found that BPC compresses branch trace data better than VPC in 19/20 cases and is between  $1.1\times$  and  $2.2\times$  faster.

CBP uses a simpler set of branch predictors than BPC to generate and read compressed traces. Though it uses similar techniques, direct comparison is not possible as CBP obtains near-perfect program counter compression due to the interleaving of non-branch instructions. With perfect PC prediction, CBP+bzip2 outperforms BPC in 10/20 cases, but when perfect prediction is not allowed, BPC produces smaller files.

## 5. Acknowledgments

We thank Joel Emer, Martin Burtscher, Jared Stark, the MIT SCALE group, and the anonymous reviewers. This work was partly funded by the DARPA HPCS/IBM PERCS project, NSF CAREER Award CCR-0093354, and an equipment grant from Intel Corp.

## References

- [1] K. C. Barr and K. Asanović. Branch trace compression for snapshot-based simulation. In *Int’l Symp. on Perf. Analysis of Systems and Software*, Mar. 2006.
- [2] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratana-worabhan, and N. B. Sam. The VPC trace-compression algorithms. *IEEE Trans. on Computers*, 54(11), Nov. 2005.
- [3] D. Shkarin. PPM: one step to practicality. In *Data Compression Conf.*, 2002.
- [4] J. W. Stark and C. Wilkerson et al. The 1st JILP championship branch prediction competition. In *Workshop at MICRO-37 and Journal of ILP*, Jan. 2005. <http://www.jilp.org/cbp/>.

## Branch Trace Compression for Snapshot-Based Simulation

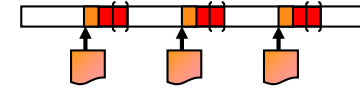
Kenneth Barr  
Krste Asanović



BARC  
February 3, 2006

## BPC: compact, fast, flexible warming of branch predictors for snapshot-based simulation.

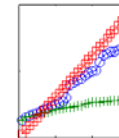
1. Motivation, simulation context, vocabulary



2. Branch Predictor-based Compression (BPC)  
– Compress traces instead of storing snapshots



3. Preview of results  
– Size  
– Scalability  
– Speed



Barr and Asanović. BARC 2006. Feb 3, 2006.

2 of 15

## Intelligent sampling gives best speed-accuracy tradeoff for uniprocessors (Yi, HPCA '05)

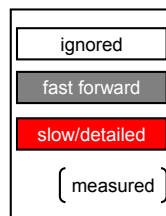
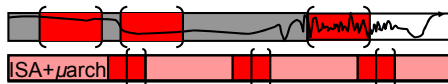
Run benchmark entirely in detailed mode: slow!



Aggregate detailed samples



Variations



Barr and Asanović. BARC 2006. Feb 3, 2006.

3 of 15

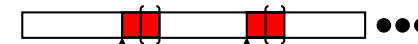
## Snapshots amortize fast-forwarding, but require slow warming or bind us to a particular $\mu$ arch.

ISA only snapshots:



Slow due to warmup, but allows any  $\mu$ arch

ISA+ $\mu$ arch snapshots:

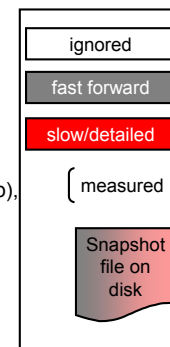


Fastest (less warmup), but tied to  $\mu$ arch

ISA+ $\mu$ arch-independent snapshots:



Fast, NOT tied to  $\mu$ arch (Cheetah, MTR)

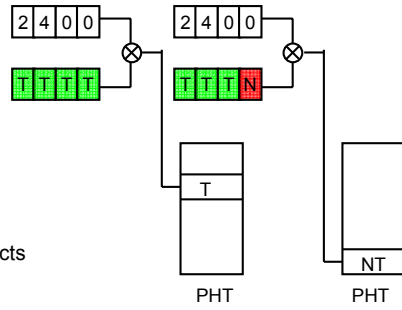


Barr and Asanović. BARC 2006. Feb 3, 2006.

4 of 15

### Why can't we create $\mu$ arch-independent snapshot of a branch predictor?

- In cache, an address maps to a particular cache set
- Branch history (global or local) "smears" static branch across the pattern history table
  - Same branch address.....
  - In a different context.....
- In a cache, we can throw away LRU accesses
- In a branch predictor, who knows if ancient branch affects future predictions?!



### If a $\mu$ arch independent snapshot is tricky, let's try to store several predictor tables?

- Suggested by [SMARTS, SimPoint]
- Is this an option?
  - If you generate snapshots via hardware dumps, you can't explore other microarchitectures
- Which ones?
  - If it takes two weeks to run a non-detailed simulation of a real workload you don't want to guess wrong
- Those branch predictors aren't as small as you think!

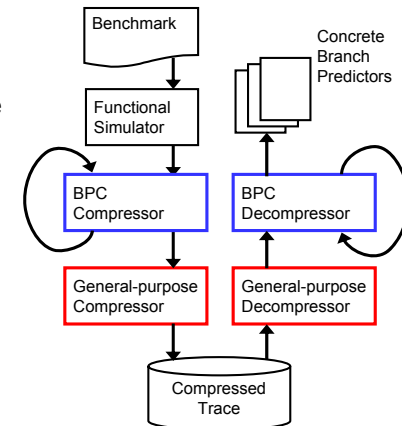
### Branch predictors are small, but multiply like rabbits! 8KB quickly becomes 1000's of MB.

- P**: gshare with 15 bits of global history 8 KBytes
- n**: 1 Billion instructions in trace sampled every million insts requires 1000 samples  $x 1000 = 8$  MBytes
- m**: 10 other tiny branch predictors  $x 10 = 78$  MBytes
- 26 benchmarks in Spec2000  $x 26 = 2.0$  GBytes
- 16 cores in design?  $x 16 = 32$  GBytes
- Now, add BTB/indirect predictor, loop predictor...
- Scale up for industry: 100 benchmarks, 10s of cores



### BPC compresses branch traces well and quickly warms up any concrete predictor.

- Simulator decodes branches
- BPC Compresses trace
  - Chaining if necessary
- General-purpose compressor shrinks output further
  - PPMd
- Reverse process to fill concrete predictors



## BPC uses branch predictors to model a branch trace. Emits only unpredictable branches.



- Contains the branch predictors you always dreamed about!
  - Large global/local tournament predictor
    - 1.44Mbit
    - Alpha 21264 style
  - 512-deep RAS
  - Large hash tables for static info
    - Three 256K-entry
  - Cascaded indirect predictor
    - 32KB leaky filter
    - path-based (4 targets)
    - 2 entries
    - PAg structure

Barr and Asanović. BARC 2006. Feb 3, 2006.

9 of 15

## BPC Compression

Input: branch trace from functional simulator

```
0x00: bne 0x20 (NT)
0x04: j    0x1c (T)
0x1c: ret          (T to 0xc4)
```



Output:

- If BPC says “I could have told you that!” (Common case): no output  
<>
- If BPC says “I didn’t expect *that* branch record!”  
< skip N, branch record >

Update internal predictors with every branch.

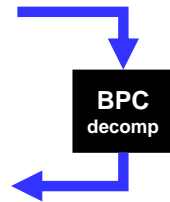
Barr and Asanović. BARC 2006. Feb 3, 2006.

10 of 15

## BPC Decompression

Input: list of pairs < skip N, branch record >

```
< 0,    0x00: bne 0x20 (NT) >
< 0,    0x04: j   0x1c (T)   >
< 13,   0x3c: call 0x74    >
```



Output:

```
if (skip==0)
  branch record
  // updates predictors

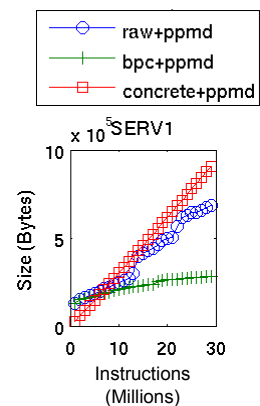
while(skip > 0)
  BPC says “let me guess!”
  // updates predictors
  // decrement skip
```

Barr and Asanović. BARC 2006. Feb 3, 2006.

11 of 15

## BPC-compressed traces grow slower than concrete snapshots

- We compare against **one** stored Pentium 4 style predictor: 2.7X smaller (avg)
- If you store 1000 samples, 10 predictors...
  - 11 MB for BPC
  - 310 MB for concrete snapshot
- Growth
  - BPC has shallow slope
  - concrete scales with *mnP*
  - Both grow with number of benchmarks and cores

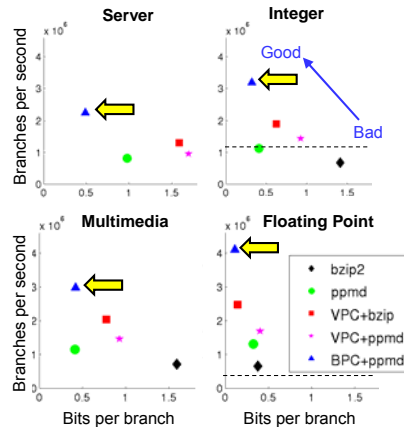


Barr and Asanović. BARC 2006. Feb 3, 2006.

12 of 15

### Summary: BPC decompresses faster, compresses as good or better than others.

- BPC+PPMd faster than other compressors **and** sim-bpred
- Know your general-purpose compressors: gzip's too big bzip2 is too slow
- Biggest help for phase-changing Server code



Barr and Asanović. BARC 2006. Feb 3, 2006.

13 of 15

### Related work: BPC is a specialized form of VPC or a modified version of CBP.

- Value-predictor based compression (VPC)
  - Prof. Martin Burtscher at Cornell
  - Trans on Computers, Nov 2005
- Championship Branch Prediction Contest (CBP)
  - Stark and Wilkerson, Intel
  - MICRO workshop, Jan 2005
  - Provided traces used a technique with similar spirit
- Our Branch Prediction-based Compression (BPC) paper identifies application to snapshot-based simulation
  - Barr and Asanović, MIT
  - ISPASS, Mar 2006

Barr and Asanović. BARC 2006. Feb 3, 2006.

14 of 15

### Conclusion

- Compressed branch traces are smaller than concrete branch predictor snapshots
  - 2.0–5.6x smaller than a **single**, simple predictor snapshot
  - Improvement multiplies for each predictor under test, size of those predictors, and each additional sample
- We introduce Branch Predictor-based Compression
  - Better compression ratios than other compressors
  - Faster than other decompressors; and 3-12X faster than functional simulation. Slower than march snapshots, but infinitely more flexible.

- Full-length paper: ISPASS, March 2006
- <http://cag.csail.mit.edu/scale>

Barr and Asanović. BARC 2006. Feb 3, 2006.

15 of 15

# Requirements for any HPC/FPGA Application Development Tool Flow

*(that gets more than a small fraction of potential performance)\**

Tom Van Court

Martin C. Herbordt

Department of Electrical and Computer Engineering  
Boston University; Boston, MA 02215  
EMail: {tvancour|herbordt}@bu.edu

## 1 Introduction

One of the most exciting innovations in computer architecture in many years is the introduction by several major vendors of FPGA-based processing nodes. FPGAs are fundamentally different from von Neuman (vN) processors: applications are configured in circuitry, rather than programmed into software. The critical problem to be solved with respect to High Performance Computing using FPGAs (HPC/FPGA) is determining an appropriate method for configuring the FPGAs; this problem involves the often inherent conflict between performance and development cost.

The argument is currently being made, in analogy to the historic assembly language versus high level language (HLL) debate—and to a lesser extent, the similarly historic quest for portable parallel programs—that the expected mode for creating HPC/FPGA applications is with HLLs, including the direct compilation of HLL programs into gates. Moreover, the argument continues, this so-called C2gates approach can be to the exclusion of logic-aware design, such as, e.g., would require using a Hardware Description Language (HDL).

In this abstract, we argue that the conflict between C2gates and logic-aware approaches is far from settled; in particular, that C2gates as so described is fundamentally flawed; and that although C2gates will be highly useful in most design flows, *logic-aware design is essential to obtain even a fraction of the possible capability of HPC/FPGA*. Putting this another way: we argue that the tremendous performance advantage (potentially) obtained by freeing the node architecture

from the vN model is not likely to remain an advantage after the vN model is reimposed back onto that node architecture.

## 2 Design Methods

In the rest of this abstract we present a number of fundamental design methods that are both essential for creating efficient HPC/FPGA applications and likely to be beyond C2gates. Or at least beyond C2gates in the sense that the developer is not particularly “FPGA-aware” in addition to not being logic-aware. These potential monkey wrenches in the FPGA machinery (particularly 1-10 below) each have the capacity of reducing performance by an order of magnitude or more.

**1. Use the correct programming model.** In particular, exactly the wrong programming model for FPGA configuration is the one used for serial computing. In the classic example, a program is created having a FOR loop with a harmless, but unnecessary, dependency among its iterations. There is no effect on serial performance, but on the FPGA, little parallelism can be extracted. There are other not-so-blatant examples: parallel programming models, such as global shared memory, message passing, and data parallel are likely to be better than serial, but still far from ideal. In particular, none is likely to inspire an appropriate FPGA algorithm. In contrast, certain programming styles work particularly well on FPGAs: 1D and 2D systolic arrays; associative computing, which is dominated by broadcast and reduction; and complex heterogeneous pipelines, i.e. pipelines of operations, rather than pipelines to execute single operations. These programming styles, collectively, could be an appropriate HPC/FPGA programming model.

---

\*This work was supported in part by the NIH through award #RR020209-01 and facilitated by donations from Xilinx Corporation. Web: <http://www.bu.edu/caadlab>.

## 2. Use an appropriate (FPGA) algorithm.

Assuming that the programming model is correct, it is still possible to use a suboptimal FPGA algorithm. HPC/FPGA implementors often create efficient FPGA versions of the corresponding serial algorithm, e.g. by getting rid of spurious inter-loop dependencies. There is often, however, a different algorithm that is superior to the one originally chosen. One example comes from modeling molecular interactions: the standard algorithm uses FFTs, but on FPGAs direct correlation is preferred. Another example comes from BLAST: the original uses tables of pointers and random access; a preferred algorithm uses streaming and a 2D systolic array.

**3. Speed match computations.** Even a good algorithm can be implemented so that it does not use resources properly. For example, (pipelined) stages of a computation can have drastically different operating frequencies. If they are not “speed-matched,” then they will all run at the slowest frequency. This problem can be solved by replicating stages for parallel execution so that the data production per unit time matches. This item is related to load balancing in parallel programming.

**4. Use all chip resources.** A related issue is that of leaving chip resources unused. Sometimes this is unavoidable, such as when an application simply does not call for a large number of multipliers or memory ports to be used. Other times, however, the design system simply will not let repeating elements in a computing array expand to fill available resources. This item is related to scalability in parallel programming.

**5. Hide latency of independent functions.** For example, let’s say a random number is required for a particular operation. There is probably no reason that the random number cannot be computed so that it is available in time.

**6. Use appropriate constructs.** (Overlaps with Use Correct Programming Model; related – do not use inappropriate constructs.) One of the best features of FPGAs is their ability to do *hardware computing*. Broadcast, reduction, and leader election all can be done at electrical speeds, resulting in a large number of instruction-equivalents being executed by a single hardware structure in a single cycle. On the other hand, some operations—sparse data structures, pointer following, and non-pipelined random data access—are very slow. Also related to this item is the use of the correct hardware construct for the correct structure: For example, there are well-known ways to implement FIFOs in hardware that yield performance far superior to naive implementations.

## 7. Use FPGA resource types appropriately.

Modern FPGAs contain much more than just loose gates. In particular, they contain hundreds of embedded block RAMS and multipliers. In a molecular dynamics application, we access 400 separate memories for two reads and two writes on every cycle, yielding an overall on-chip memory bandwidth of 20Tb/s. Just as important is to not use resources incorrectly. For example, configuring chip resources into complete CPUs (softcores) for high performance applications is unlikely to be beneficial. The small number of such CPUs that can be configured into a high-end FPGA is unlikely to match the performance of even a single microprocessor, much less offer significant speed-up.

## 8. Arithmetic 1: Use appropriate precision.

Certain applications—notably those involving biological sequences, but also many others—require only a few bits of precision. Using appropriate precision allows a proportional increase in parallelism and so performance.

## 9. Arithmetic 2: Use appropriate operations.

The most obvious of these (unfortunately) is to limit the use of floating point. An FPGA implementation of a floating point unit following the complete IEEE standard requires the bulk of a high-end FPGA. There are often substitutes, however, that cost little performance, and use far fewer resources. For example, in a molecular dynamics application, floating point was successfully replaced with a combination of table look-up, “semi-fixed floating point,” and higher order interpolation.

**10. Overall 1: Use good logic design.** Poor HDL-level design leads to even poorer logic. One of the most common pitfalls in modern logic design is the use of HDLs without an understanding of what logic will actually be generated. Inexperienced designers quickly learn that seemingly trivial oversights or imprecisions lead to logic that is correct, but that runs at a small fraction of projected performance. One vendor of a commercial C2gates tool reports that 180 lines of HLL code generated 150,000 lines of VHDL code.

## 11. Overall 2: Recognize that there are inherent limits to logic synthesis (compilation).

There exist formal results proving that an optimal design can be unreachable even from a semantically equivalent but syntactically different description.

## 12. Overall 3: Recognize the brittleness of high-performance solutions.

As we know from Amdahl’s Law, the greater the potential speed-up, the more difficult it is to obtain a large fraction of that speed-up.



## Requirements for any FPGA/HPC Application Development Tool Flow

... if you want any reasonable fraction of the FPGA's potential performance



Tom VanCourt  
Martin C. Herbordt

Computer Architecture and Automated Design Lab

<http://www.bu.edu/caadlab>

## What is FPGA/HPC exactly?

- High performance computing  
*Computational chem.    Electromagnetics*  
*Bioinformatics        Traffic modeling*  
*Astrophysics            ...*
- Field Programmable Gate Arrays  
*App. specific processors on demand*  
*Massive fine-grained parallelism*  
*Drivers of silicon process development*

3 Feb 2006

T. VanCourt & M. C. Herbordt

2 of 10

## What's so hard about it?

- Performance computing  $\neq$  logic design  
*Standard languages hide parallelism\**  
*FPGA tools address logic designers*
- Contradictions in FPGA applications  
*Applications should be widely applicable*  
*... but finely tuned to each particular usage*  
*Require customization by application specialist*  
*... but require unfamiliar hardware constructs*  
*Demand full use of hardware resources*  
*... use is app-specific, resources are FPGA-specific*

\*Jeroen Voeten,  
ACM Trans. CAD  
6(4)533-552,  
Oct 2001

3 Feb 2006

T. VanCourt & M. C. Herbordt

3 of 10

## What's wrong with C to gates?

- "Unfortunately, and despite 40 years of parallelizing compilers for all sorts of machines, [optimization] algorithms don't work terribly well." Ian Page, 2004*
- The best you get is C code in gates  
*Good HW algorithm isn't SW algorithm*
  - C distributes algorithms in time  
*FPGAs distribute algorithms in space*  
*... and a whole industry is dedicated to reinventing the von Neumann bottleneck*

3 Feb 2006

T. VanCourt & M. C. Herbordt

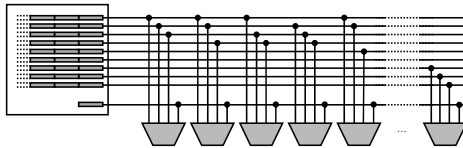
4 of 10

## Example: Size-3 subsets

- C style:

```
for i = 0 to N
  for j = 0 to i
    for k = 0 to j
      // use x[i],x[j],x[k]
```

- HW-oriented solution:



3 Feb 2006

T. VanCourt &amp; M. C. Herbordt

5 of 10

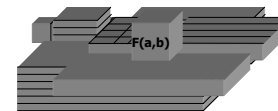
## Example: 3D Correlation

- Serial processor: Fourier transform  $\mathcal{F}$

$$A \otimes B = \mathcal{F}^{-1}(\mathcal{F}(A) \times \mathcal{F}(B))$$

- FPGA: Direct summation

RAM FIFO



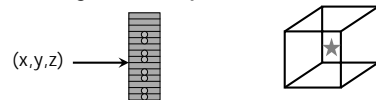
3 Feb 2006

T. VanCourt &amp; M. C. Herbordt

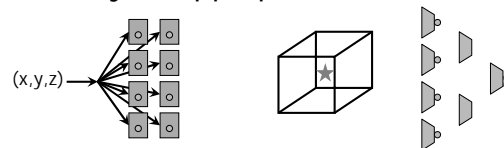
6 of 10

## Example: Trilinear Interpolation

- C style: Sequential RAM access



- HW style: App-specific interleaving



3 Feb 2006

T. VanCourt &amp; M. C. Herbordt

7 of 10

## Sizing applications to FPGAs

- Desired size of computing array:

*As big as possible – whatever that means*

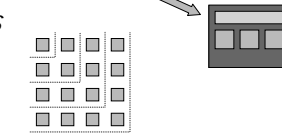
- Depends on:

*FPGA capacity*



*Application details*

*Computing array*



3 Feb 2006

T. VanCourt &amp; M. C. Herbordt

8 of 10

## C Coding Style vs. Performance

- Hardware algorithms are different
  - Require non-SW algorithms*
  - Require non-von Neumann memory*
  - Require non-obvious data paths*
  - Require careful precision analysis*
- Explicit degree of parallelism is a bug
  - No commercial tools address all factors*
- App. specialists aren't logic designers
  - Need both – efficient HW & app. details*

3 Feb 2006

T. VanCourt & M. C. Herbordt

9 of 10

## The Requirements

- Escape from the C code model
  - GPUs? Device organizes control & memory*
  - ... application is leaf calculations only*
- Support two developer groups:
  - Logic designers create efficient structures*
  - App specialists tailor it to specific usage*
- Full use of FPGA's computing resources
  - App-specific, FPGA-specific array sizes*

3 Feb 2006

T. VanCourt & M. C. Herbordt

10 of 10

# Accelerating Architectural Exploration Using Canonical Instruction Segments

Rose F. Liu and Krste Asanović

MIT Computer Science and Artificial Intelligence Laboratory  
The Stata Center, 32 Vassar Street, Cambridge, MA 02139  
{rliu, krste}@csail.mit.edu

## 1. Introduction

Detailed microarchitectural simulators are not well suited for exploring large design spaces due to their excessive simulation times. We introduce AXCIS (Architectural eXploration using Canonical Instruction Segments), a framework for fast and accurate design space exploration. AXCIS achieves fast simulation times by compressing a program’s dynamic trace into a form that can model many different machines. AXCIS performs compression and performance modeling using a new primitive called the *instruction segment* to represent the context of each dynamic instruction. As shown in Figure 1, AXCIS is divided into two stages: dynamic trace compression and performance modeling.

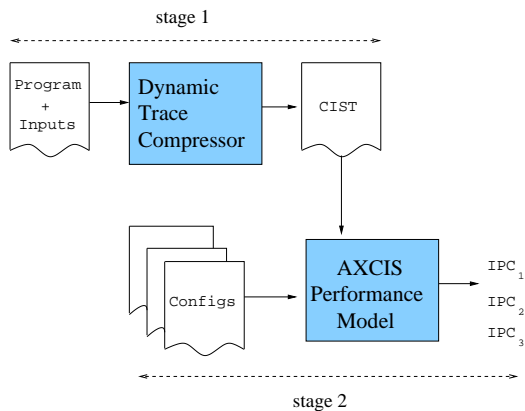


Figure 1. AXCIS Simulation Framework.

During the first stage, the Dynamic Trace Compressor (DTC) identifies all instruction segments within the dynamic trace and compresses these into a Canonical Instruction Segment Table (CIST). In order to capture locality events (ex. cache and branch behaviors) within instruction segments, partially-specified branch predictors and instruction/data caches are simulated. During this stage, only the organizations (ex. sizes, associativities)

of these structures are specified and not their latencies, allowing the generated CISTs to be used for simulating multiple machines. To perform compression, the DTC compares each dynamic segment against existing CIST entries, and either increments the count of an existing entry if a match is found or adds the new segment to the CIST if not. CISTs can be adjusted to trade simulation speed for accuracy by varying the DTC compression scheme.

In the second stage, the AXCIS Performance Model (APM) uses dynamic programming to quickly estimate performance in terms of instructions per cycle (IPC) for each design, given a CIST and a set of microarchitecture configurations. First, the APM calculates the total effective stall cycles of the CIST by summing the stall cycles of each segment weighted by their corresponding frequency counts. Then the APM computes IPC using the total effective stall cycles and the total instructions in the dynamic trace.

This paper applies AXCIS to in-order superscalar processors, although the main ideas behind AXCIS apply to out-of-order processors as well. More detailed descriptions of AXCIS can be found in [1] and [2].

## 2. Instruction Segments and CISTs

Each dynamic instruction has a corresponding instruction segment, containing its dependencies, locality events, and all preceding instructions directly affecting the stalls it experiences. A segment begins with the producer associated with the instruction’s longest dependency and ends with the instruction itself, termed the *defining instruction* of the segment. Instructions within segments are abstracted into *instruction types* (ex. integer ALU, LD hit). The left side of Figure 2 shows a sequence of dynamic instructions and their segments. Dependencies are represented by arrows, and the segment for the `st_miss` is highlighted.

A CIST is an ordered array of instruction segments and their frequency counts that also records the total in-

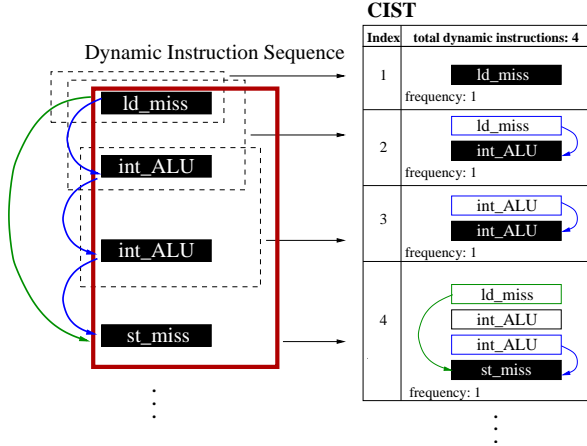


Figure 2. Instruction Segments and CIST.

structions analyzed during trace compression. The right side of Figure 2 shows the CIST corresponding to a sequence of dynamic instructions and their instruction segments.

### 3. Compression Schemes

Because program behavior repeats over time, many dynamic instructions have equivalent segments that can be compressed. Two instruction segments can be compressed, without sacrificing accuracy, if their defining instructions have the same set of stall cycles (performance) under all configurations. Since it is not possible to determine the set of all possible stalls for an instruction under all configurations, we propose three different compression schemes that approximate this idea using heuristics, while making different tradeoffs between compression and accuracy.

In the *Limit-Configurations Based Compression Scheme*, the DTC simulates two configurations (min and max) to calculate, for each instruction, a pair of stall cycles and *structural occupancies* (snapshots of microarchitectural state) to approximate its set of all possible stalls. Two segments are compressed if their defining instructions have the same (1) min/max stall cycles, (2) min/max structural occupancies, and (3) instruction types. In the *Relaxed Limit-Configurations Based Compression Scheme*, only the defining instruction types and the min/max stalls are compared for equality.

Instruction segments that look the same are more likely to have the same number of stall under all configurations. Therefore, the *Instruction Segment Characteristics Based Compression Scheme* compares segment characteristics such as segment lengths, instruction types, locality events, and dependence distances.

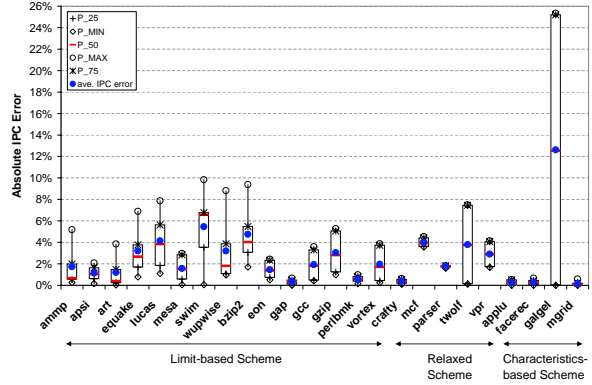


Figure 3. Distribution of absolute IPC error for each benchmark obtained under its optimal compression scheme.

### 4. Results

We evaluated AXCIS against our baseline cycle-accurate simulator, SimInOrder, for speed and accuracy (absolute IPC error between results obtained from AXCIS and SimInOrder). We simulated a wide range of microarchitectures, differing in memory latency, issue width, number of processor functional units and primary-miss tags in the nonblocking data cache using 24 SPEC CPU2000 benchmarks [3].

Using the optimal compression scheme for each benchmark (selected from the three explored), AXCIS is highly accurate and configuration independent, achieving an average IPC error of 2.6% with an average error range of 4.4% over all benchmarks and configurations. Except for `galgel` with a maximum error of 25.3%, the maximum error of all benchmarks is less than 10%. Figure 3 shows the distribution of IPC error specified in quartiles for each benchmark. Using pre-generated CISTs, AXCIS is over four orders of magnitude faster than conventional detailed simulation. While cycle-accurate simulators can take many hours to simulate billions of dynamic instructions, AXCIS can complete the same simulation on the corresponding CIST within seconds.

### References

- [1] R. F. Liu. AXCIS: Rapid architectural exploration using canonical instruction segments. Master's thesis, Massachusetts Institute of Technology, 2005.
- [2] R. F. Liu and K. Asanovic. Accelerating architectural exploration using canonical instruction segments. To appear in *International Symposium on Performance Analysis of Systems and Software*, March 2006.
- [3] SPEC CPU2000 benchmark suite. Standard Performance Evaluation Corporation. <http://www.spec.org/cpu2000/>.



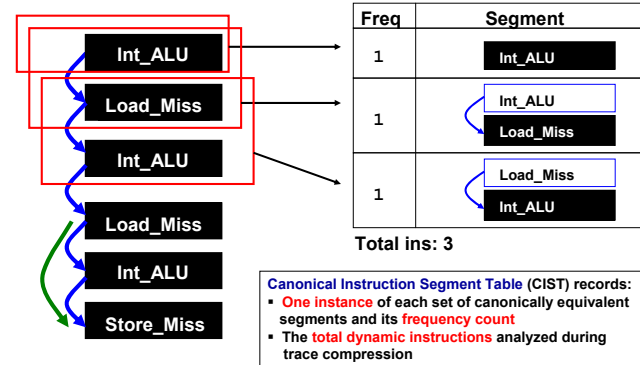
## Dynamic Trace Compression

- Repetition in program behavior such as loops, and code reuse cause instruction segments of different dynamic instructions to be **canonically equivalent**
- **Ideal Compression Scheme:** (no loss in accuracy)
  - Compress two segments if they always experience the same stall cycles regardless of the machine configuration
  - Impractical to implement within the Dynamic Trace Compressor
- **Three compression schemes that approximate this ideal scheme**
  - Each selects a different tradeoff between accuracy and speedup
  - Our simplest scheme compresses segments that look the same (i.e. have the same length, instruction types, dependence distances, branch and cache behaviors)



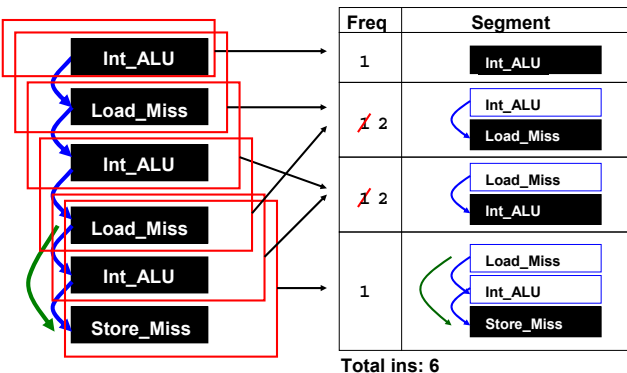
5 of 13

## Instruction Segments & CIST Example



6 of 13

## Instruction Segments & CIST Example



7 of 13

## AXCIS Performance Model

- Methodology is independent of the compression scheme used to generate the CIST
- Calculates IPC using a single linear **dynamic programming** pass over CIST entries

$$IPC = \frac{\text{Total Ins}}{\text{Total Ins} + \text{Total Effective Stalls}} = \frac{\text{Total Ins}}{\text{Total Cycles}}$$

Total Effective Stalls =

$$\sum_{i=1}^{\text{CIST Size}} \text{Freq}(i) * \text{EffectiveStalls}(\text{DefiningIns}(i))$$



8 of 13

## Dynamic Programming Example

Freq	Segment	Stalls
1	Int_ALU	
2	Int_ALU Load_Miss	
2	Load_Miss Int_ALU	
1	Load_Miss Int_ALU Store_Miss	

Total ins: 6

Look up in previous segment  
 Calculate

- Total work is proportional to the # of CIST entries
- Calculate the stalls of the defining instruction in each segment
- Look up stalls of other instructions in previous entries



9 of 13

## Experimental Setup

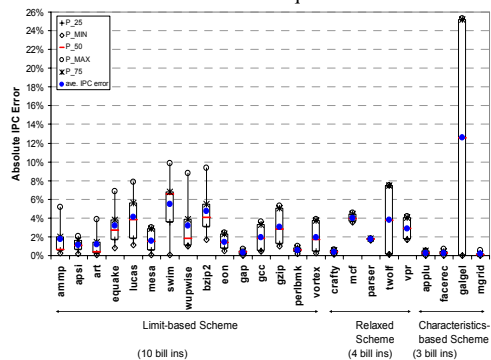
- Evaluated AX CIS against a baseline cycle accurate simulator on 24 SPEC2K benchmarks using their respective optimal compression schemes
- Evaluated AX CIS for:
  - Accuracy:
 
$$\text{Absolute IPC Error} = \frac{|\text{AXCIS} - \text{Detailed Sim}|}{\text{Detailed Sim}} \cdot 100$$
  - Speed: # of CIST entries, time in seconds
- For each benchmark, simulated many configurations that span a large design space:
  - Issue width: {1, 4, 8}, # of functional units: {1, 2, 4, 8}, Memory latency: {10, 200 cycles}, # of primary miss tags in non-blocking data cache: {1, 8}



10 of 13

## Results: Accuracy

Distribution of IPC Error in quartiles



Average Absolute IPC Error = 2.6 %

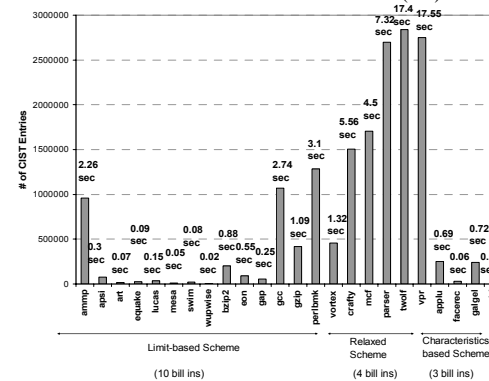
Average Error Range = 4.4%



11 of 13

## Results: Speed

# of CIST entries and simulation time (sec)



- AX CIS is over **4 orders of magnitude faster** than detailed simulation

- While detailed simulation takes hours to simulate billions of instructions, **AX CIS takes seconds**



12 of 13



## Conclusion

---

- **AXCIS is a fast, accurate, and flexible tool for design space exploration**
  
- **AXCIS**
  - Over four orders of magnitude faster than detailed simulation
  - Highly accurate across a broad range of designs
  - Predicts performance as well as buffer occupancies
  
- **Future Work**
  - More general compression schemes
  - Support out-of-order processors



# Tortola: Addressing Tomorrow’s Computing Challenges through Hardware/Software Symbiosis

Kim Hazelwood<sup>1,2</sup>

<sup>1</sup>University of Virginia, Dept. of Computer Science

<sup>2</sup>Intel Corporation, VSSAD Group

## Abstract

*Until recently, the vast majority of research efforts in optimizing computer systems have targeted a single logical “layer” in isolation: application code, operating systems, virtual machines, microarchitecture, or circuits. However, we are reaching the limits of the solutions than we can provide by targeting a single design layer in isolation. The Tortola project explores a symbiotic relationship between a virtual machine and the host microarchitecture to solve crosscutting concerns in the areas of power, reliability, security, and performance using both hardware and software extensions. We have demonstrated the effectiveness of our approach on the well-known  $dl/dt$  problem, where we successfully stabilized the voltage fluctuations of the CPU’s power supply by transforming the source code of the executing application using feedback from hardware.*

*This paper and accompanying talk will motivate our notion of symbiotic program optimization, discuss various applications, and detail our experiences in solving the  $dl/dt$  problem using a holistic approach.*

## 1 Overview

Modern computer system designers must consider many more factors than just raw performance. Thermal output, power consumption, reliability, testing, and security are quickly becoming first-order concerns. Yet, the vast majority of research efforts in optimizing computer systems have targeted a single logical “layer” in isolation: application code, operating systems, virtual machines, microarchitecture, or circuits. There are several reasons to believe we are reaching the limits of the solutions we can provide by targeting a single layer in isolation.

An important class of computing challenges exist that are better suited for more holistic approaches. Many challenges can be solved much more easily using “re-

active” techniques, whereby the hardware can detect a problem, and a virtual machine can use its global knowledge about the executing workload to correct the problem. In fact, this solution has the potential to outperform each of its constituent hardware-only or software-only solutions.

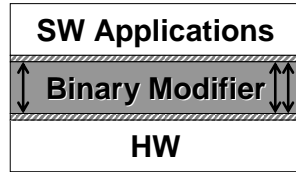
## 2 A Virtual Interface

In order to explore such solutions, the Tortola project introduces a virtual interface between the application software and the underlying machine architecture. The unique aspect of this interface is that it facilitates communication between the microprocessor and the virtual layer, which allow us to investigate combined hardware-software techniques for solving many of the future computing challenges.

In essence, this solution *virtualizes* the ISA, allowing solutions to be developed which span the hardware-software divide, as shown in Figure 1. As the figure indicates, the virtual machine can use hardware feedback to detect various machine-specific events, such as voltage fluctuations in the power supply, temperature and power problems, as well as performance-related events, such as cache misses or resource contention. The VM can then factor in its knowledge about the executing workload, such as the specific instructions selected, their instruction schedule, and the control-flow graph. Finally, the VM can develop a holistic solution to the problem at hand which accounts for both hardware and software inputs; we call this technique *symbiotic optimization*.

## 3 Our Approach

As Figure 1 indicated, our solution requires changes to the hardware in order to provide feedback to our virtual layer. Rather than building custom hardware, we began our investigations using simulation. We used SimpleScalar [1] for x86 to simulate our modified hardware. We also incorporated the Watch [2] power extensions



**Figure 1.** The Tortola architecture including a virtual layer to support HW-SW communication channels.

to SimpleScalar in order to enable power results in addition to performance results. For our virtual machine layer, we used the x86/Linux version of the Pin dynamic instrumentation system [5].

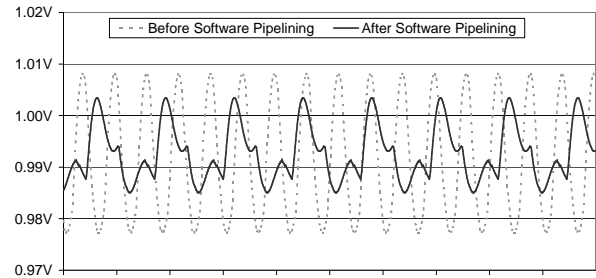
To explore symbiotic optimization, we execute our applications on top of Pin, which is running on top of SimpleScalar. Not surprisingly, a great deal of implementation effort was required in order to get SimpleScalar to the point where it could successfully emulate all of the system calls that Pin requires. The end result is a holistic simulation environment that allows us to explore collaborative solutions to existing and future computing challenges.

#### 4 A Motivating Example

Many system design problems exist that can benefit greatly from the holistic design approach we present, including issues related to power, performance, temperature, reliability, and security. As a motivating example, we present one such problem and symbiotic solution.

**A Symbiotic di/dt Solution** We have demonstrated the effectiveness of our approach on the well-known di/dt problem. The di/dt problem is a side-effect of modern techniques in low-power processor design. In order to reduce overall power consumption, idle portions of a processor are turned off. If one feature is repeatedly turned on and off, reliability problems can arise.

Hardware-based sensor/actuator mechanisms have been proposed to detect and react to these problematic current variations [4], but they do so at a performance cost to the running application. Our approach keeps these hardware solutions in tact, but simply communicates to the virtual machine in real time when the problem occurs. The virtual machine then modifies and caches the currently executing instructions in an attempt to avoid the problem in the future. The code transformations can be fairly straightforward. In fact, we were able to apply standard compiler optimizations (loop unrolling and software pipelining) to remedy future recurrences of the di/dt problem. Figure 2 shows the result



**Figure 2.** The effect of software pipelining on a voltage fluctuation stressmark.

of applying software pipelining to a hand-coded *power virus* described by Joseph et al. [4].

Our symbiotic solution was therefore able to provide the safety of the hardware-only technique with the performance improvements possible from a software-only technique. More detailed information about this particular di/dt solution is available in our ISLPED paper [3].

#### 5 Acknowledgments

Several others have contributed to this project. David Brooks provided a great deal of brainstorming effort in developing the di/dt component of this project, as well as with the Watch power extensions. Greg Humphreys helped to develop the project name and logo, and also, together with Dan Williams, provided invaluable software development of SimpleScalar/x86 to get it to the point where it could successfully run Pin. We thank Brad Calder for granting us early access to the new SimpleScalar for x86. Finally, Russ Joseph's power virus and power supply extensions to Watch have proven to be extremely useful in this research.

#### References

- [1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, pages 59–67, February 2002.
- [2] D. Brooks, V. Tiwari, and M. Martonosi. Watch: A framework for architectural-level power analysis and optimizations. In *International Symposium on Computer Architecture (ISCA-27)*, 2000.
- [3] K. Hazelwood and D. Brooks. Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization. In *International Symposium on Low-Power Electronics and Design*, pages 326–331, Newport Beach, CA, August 2004.
- [4] R. Joseph, D. Brooks, and M. Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. In *High-Performance Computer Architecture (HPCA-9)*, 2003.
- [5] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

## Tortola: Addressing Tomorrow's Computing Challenges through Hardware/Software Symbiosis



Kim Hazelwood  
University of Virginia  
Intel Corporation, VSSAD

## Modern Computing Challenges

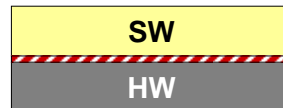
- Performance
- Power
  - Energy consumption, max instantaneous power, di/dt
- Temperature
  - Total heat output, “hot spots”
- Reliability
  - Neutron strikes, alpha particles, MTBF, design flaws
- Approaches: Circuit, microarchitecture, compiler
- Constraint: Fixed HW-SW interface (e.g., x86)



2 of 16

## Typical Approaches

- Optimize using SW or HW techniques in isolation
- Performance
  - SW: Compile-time optimizations
  - HW: Architectural improvements, VLSI technology
- Reliability: Code/data duplication (HW or SW)
- Power & Temperature
  - HW control mechanisms
  - Profile + recompile cycle



3 of 16

## Modern Design Constraints

Compilers – “Compile once, run anywhere”

- Cannot ship “MS Office for 1Q05 batch of Pentium-4 3GHz, > 1GB RAM, BrandX power supply, located in high altitudes...”

Microarchitecture – Limited window of application knowledge (past must predict the future)

VLSI – Guaranteed correctness, reliability

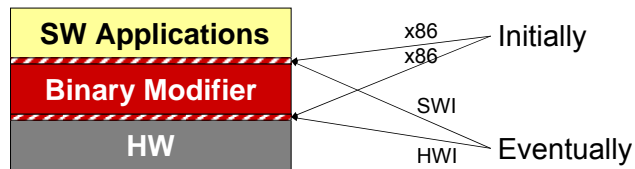
We currently must optimize for the common case  
(but must design for the worst case)



4 of 16

## The Power of Virtualization

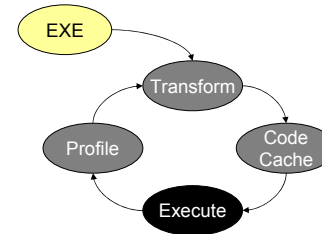
- A HW-SW interface *layer*



5 of 16

## Dynamic Binary Modification

- Creates a modified code image at run time



### Examples:

- Dynamo (HP)
- DAISY/BOA (IBM)
- CMS (Transmeta)
- Mojo (Microsoft)
- Strata (UVa)
- Pin (Intel)

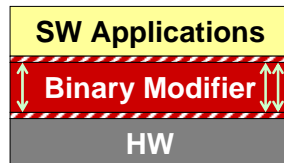
- Always triggered by software events ... until now



6 of 16

## Tortola: Symbiotic Optimization

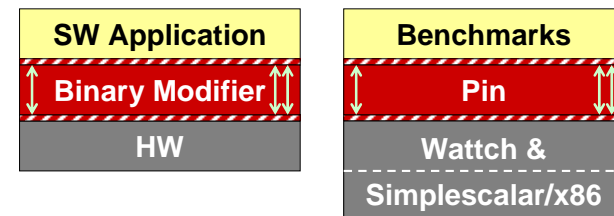
- Enable HW/SW Communication



7 of 16

## Simulation Methodology

- SimpleScalar 4.0 for x86
- Wattach 1.02 power extensions
- Pin dynamic instrumentation system (x86/Linux version)



8 of 16

## Tortola Applications

- Combine global program information with run-time feedback
  - System-specific power usage
  - Application-specific heat anomalies
  - Workload/input specific performance optimization
- Reduce hardware complexity
  - No more backwards compatibility warts
  - Fix bugs after shipment
  - Reduce time to market for new architectures
- One such application: The di/dt problem



9 of 16

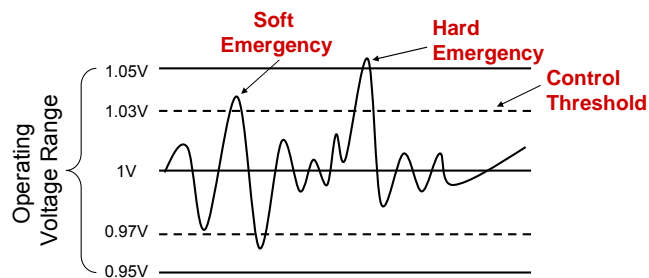
## The Di/dt Problem

- Low-power techniques have a negative side effect: current variation
- Voltage stability is important for reliability, performance
- Dips (undershoots) in supply voltage – can cause incorrect values to be calculated or stored
- Spikes (overshoots) in supply voltage – can cause reliability problems



10 of 16

## Detecting Imminent Emergencies



- Phantom firing - increases current (at the expense of power)
- Resource throttling - reduces current (at the expense of performance)



11 of 16

## A Di/dt Stressmark

```

BEGIN_LOOP:
...
ldt    $f1, ($4)
divt   $f1, $f2, $f3
divt   $f3, $f2, $f3
stt    $f3, 8($4)
ldq    $7, 8($4)
cmovne $31, $7, $3
stq    $3, $(4)
stq    $3, $(4)
stq    $3, $(4)
...
stq    $3, $(4)
...
JUMP  BEGIN_LOOP
  
```

Sequential Low Current

Parallel High Current

But...Actuator engages every loop iteration degrading performance

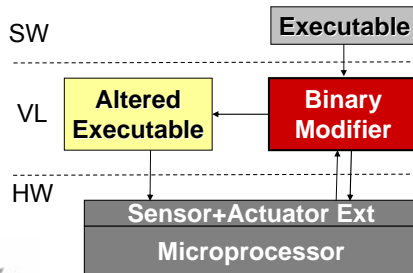
Why not correct the problem in the code?



12 of 16

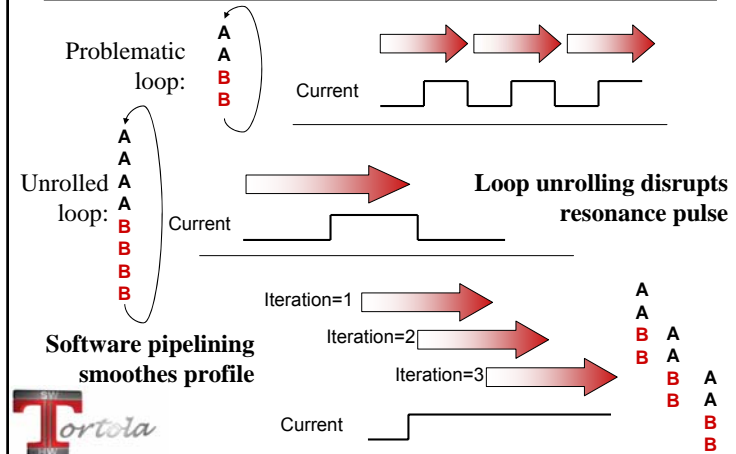
## Proposed Solution

- Leverage our additional software layer to supplement existing solutions
- Microarchitecture provides feedback to our software-based virtual layer

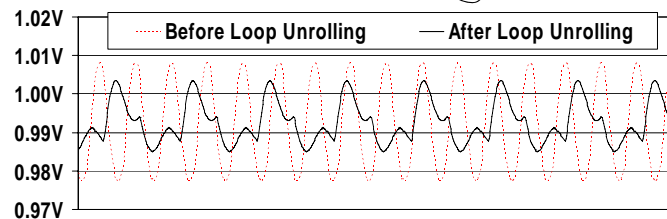
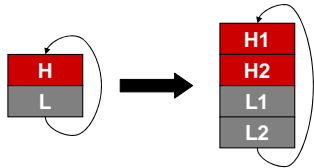


13 of 16

## Loop Unrolling & SW Pipelining



## Unrolling the Di/dt Stressmark



15 of 16

## Summary

- Symbiotic program optimization is a powerful approach
- The di/dt problem – well suited for a symbiotic solution
- The Tortola design can also target power reduction, temperature reduction, reliability, etc.

<http://www.tortolaproject.com/>



16 of 16

# Parity Replication in IP-Network Storages

Weijun Xiao, Jin Ren, and Qing Yang  
Dept of Electrical and Computer Engineering  
University of Rhode Island  
Kingston, RI 02881  
Tel: 401 874 5880  
Fax: 401 782 6422  
{qyang, wjxiao, rjin}@ele.uri.edu

## Abstract

*Distributed storage systems employ replicas or erasure code to ensure high reliability and availability of data. Such replicas create great amount of network traffic that negatively impacts storage performance, particularly for distributed storage systems that are geographically dispersed over a wide area network (WAN). This paper presents a performance study of our new data replication methodology that minimizes network traffic for data replications. The idea is to replicate the parity of a data block upon each write operation instead of the data block itself. The data block will be recomputed back at the replica storage site upon receiving the parity. We name the new methodology **PRINS** (Parity Replication in IP-Network Storages). PRINS trades off high-speed computation for communication that is costly and more likely to be the performance bottleneck for distributed storages. By leveraging the parity computation that exists in common storage systems (RAID), our PRINS does not introduce additional overhead but dramatically reduces network traffic. We have implemented PRINS using iSCSI protocol over a TCP/IP network interconnecting a cluster of PCs as storage nodes. We carried out performance measurements on Oracle database, Postgres database, MySQL database, and Ext2 file system using TPC-C, TPC-W, and Micro benchmarks. Performance measurements show up to 2 orders of magnitudes bandwidth savings of PRINS compared to traditional replicas. A queueing network model is developed to further study network performance for large networks. It is shown that PRINS reduces response time of the distributed storage systems dramatically.*

## 1. Introduction

As organizations and businesses depend more and more on digital information and networking, high reliability and high performance of data services over the Internet has become increasingly important. To guard against data loss and to provide high performance data services, data replications are generally implemented in distributed data storage systems. Examples of such systems include P2P data sharing [1,2,3], data grid [4,5] and remote data mirroring [6] that all employ replicas to ensure high data reliability with data redundancy. While replication increases data reliability, it creates additional network traffic. Depending on application characteristics in a distributed environment, such additional network traffic can be excessive and become the main bottleneck for data intensive applications and services. In addition, the cost of bandwidth over a wide area network is very high [6] making replications of large amount of data over a WAN prohibitively expensive.

In order to minimize the overhead and the cost of data replication, researchers in the P2P community have proposed techniques to reduce unnecessary network traffic for data replications. Susarla and Carter presented a new consistency model for P2P sharing of mutable data [1]. By letting applications compose consistency semantics appropriate for their sharing needs, such relaxed consistency reaps order-of-magnitude performance gains over traditional file systems. To avoid duplicated messages in the replication process, Datta, Hauswirth and Aberer proposed a hybrid push/pull rumor-spreading algorithm [2] to minimize network traffic. While these techniques can reduce unnecessary network traffic, replicated data blocks have to be multicast to replica nodes. The basic



data unit for replication ranges from 4KB to megabytes [4], creating a great amount of network traffic on replica alone. Such large network traffic will result in either poor performance of data services or excessive expenses for higher WAN bandwidth. Unfortunately, open literature lacks quantitative study of the impacts of such data replications on network performance of a distributed storage systems.

This paper presents a quantitative performance evaluation of a new data replication technique that minimizes network traffic in a P2P shared storage environment when mutable data is replicated. The new replication technique works at block level of distributed data storages and reduces dramatically amount of data that has to be transferred over the network. The main idea of the new replication technique is to replicate the parity of a changing block upon each block write instead of the data block itself, hence referred to as PRINS (Parity Replication in IP-Network Storages). Such parity is computed in RAID storage systems such as RAID 3, RAID 4 or RAID5 that are the most popular storages in use today. As a result, no additional computation is necessary at the primary storage site to obtain the parity. After the parity is replicated to the replica storage sites, the data can be computed back easily using the newly received parity, the old data and the old parity that exist at the replica sites. Extensive experiments [7,8] have shown that only 5% to 20% of a data block actually changes on a block write. Parity resulting from a block write reflects the exact data changes at bit level. Therefore, the information content and hence the size of parity is substantially smaller than the size of corresponding data block. PRINS is able to exploit the small bit stream changes to minimize network traffic and trades off inexpensive computations outside of critical data path for high cost communication.

We have implemented a PRINS software module at block device level on a cluster of PCs interconnected by a TCP/IP network, referred to as PRINS-engine. The network storage protocol that we used is the iSCSI (Internet SCSI) protocol. Our PRINS-engine runs as a software module inside the iSCSI target serving storage requests from computing nodes that have an iSCSI initiator installed. Upon each storage write request, the PRINS-engine performs parity computation and replicates the parity to a set of replica storages in the IP network. The replica storage nodes also run the PRINS-engine that receives parity, computes data back, and stores the data block in-place. The communication between PRINS-engines also uses iSCSI protocol. We have installed Oracle database, Postgres database, MySQL database, and Ext2 file system on our PRINS-engine to test its performance. TPC-C, TPC-W, and micro benchmarks are used to drive our test bed.


Measurement results show up to 2 orders of magnitudes reduction in network traffic using our PRINS-engine compared to traditional replication techniques. We have also carried out queueing analysis for large networks to show great performance benefits of our PRINS-engine.

## Acknowledgments

This research is sponsored in part by National Science Foundation under grants CCR-0073377 and CCR-0312613. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We also thank Slater Interactive Office of Rhode Island Economic Council for the generous financial support on part of this research work.


## References

- [1] S. Susarla and J. Carter, "Flexible Consistency for Wide Area Peer Replication," In *Proc. of 25th IEEE International Conference on Distributed Computing Systems (ICDCS 2005)*, Columbus, OH, June 2005, pp. 199-208.
- [2] A. Datta, M. Hauswirth, and K. Aberer, "Updates in Highly Unreliable, Replicated Peer-to-Peer Systems," In *Proc. of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS 2003)*, Providence, RI, May 2002.
- [3] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, "Pond: The OceanStore prototype," In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, Apr. 2003.
- [4] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke, "Data management and transfer in high-performance computational grid environments," *Parallel Computing*, May 2002, vol. 28, no. 5, pp. 749-771
- [5] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury and S. Tuecke, "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets," *Journal of Network and Computer Applications*, July 2000, vol. 23, no. 3, pp. 187-200.
- [6] M. Ji, A. Veitch, and J. Wilkes, "Seneca: Remote mirroring done write," *USENIX Technical Conference (USENIX'03)*, San Antonio, TX, June 2003, pp. 253-268.
- [7] Y. Hu, Q. Yang, and T. Nightingale, "RAPID-Cache --- A Reliable and Inexpensive Write Cache for Disk I/O Systems," In *the HPCA-5*, Orlando, Florida, Jan. 1999.
- [8] Y. Hu and Q. Yang, "DCD---Disk Caching Disk: A New Approach for Boosting I/O Performance," In *the ISCA-23*, Philadelphia, PA, May 1996.




## Parity Replication in IP Network Storages

Weijun Xiao, Jin Ren, and  
Qing Yang  
Dept. of ECE  
University of Rhode Island




PRINS, Dept. ECE, URI



## Motivations

- Performance
  - CPU performance: over 6 orders of magnitude change
  - Memory Performance: several orders of magnitude
  - Network performance: LAN speed: over 4 orders of magnitude
- Cost: Servers:25%; data storage 75% of IT Cost
- Reliability and Availability
  - If CPU Burned: Replace it, re-compute.
  - Memory Lost: Replace with new card, reboot
  - Network Down Fix it, rebuild, comm possible w/ other means
- What about data storage?


2 of 12



## Motivations (cont.): Real World Demand

- In 18 months (Jim Gray)
  - New Storage = sum of all old storage (ever)
- Online data storage
  - doubles every 9 months
- Cost of one hour data not available
  - up to millions \$
- IDC
  - #1 Top Challenge...“Improving Data Availability and Recovery”
  - #1 Driver of Storage ...“Data Protection and Disaster Recovery”
  - #1 Priority of storage users... “Replication”

3 of 12



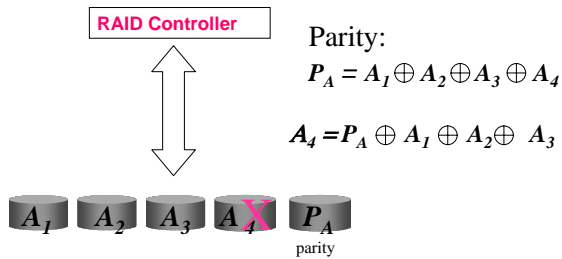
## The State-of-the-Art Technologies

- File system replication
  - LBFS, rsync, NSI, XOsft
- Block level replication
  - Synchronous vs Asynchronous
  - Delta blocks and delta set
- WAN bandwidth limitations
  - TCP optimization and data sequencing
  - Data compression before replication

4 of 12

## Our Approach

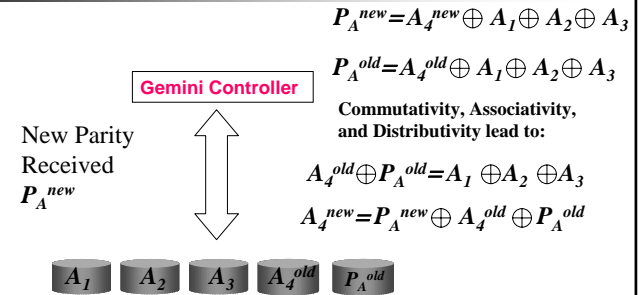
- Redundant Array of Independent disks



If data  $A_4$  is lost, it can be recovered by using parity  $P_A$ , as show above

5 of 12

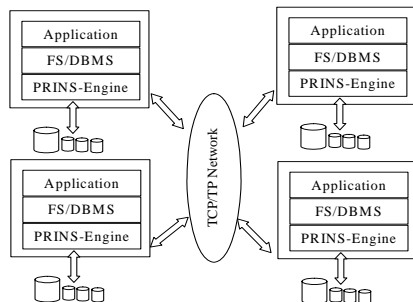
## PRINS: Parity Replication in IP-Network Storages



New data  $A_4$  can be computed using the new parity  $P_A$ , old parity, and old data already stored at the storage at mirror site

6 of 12

## PRINS Design & Implimentation

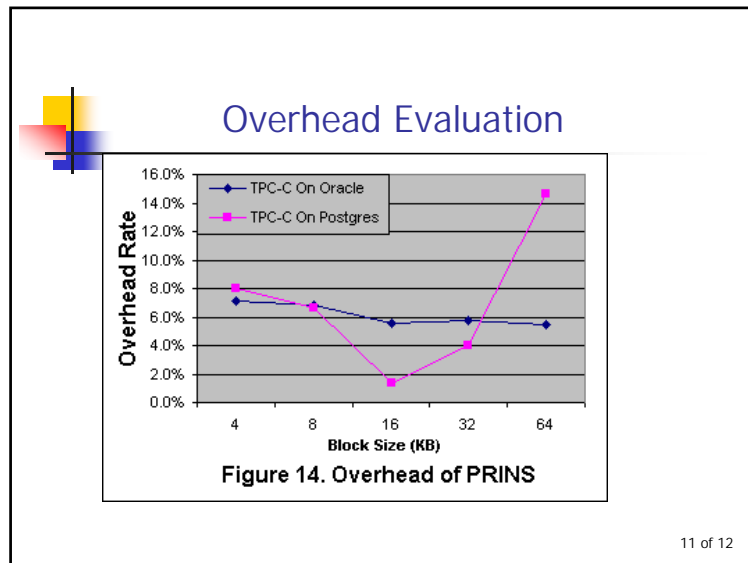
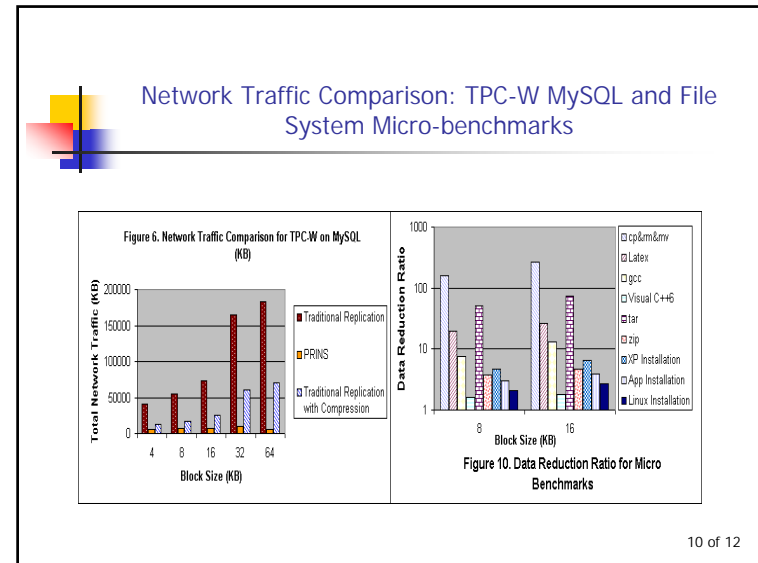
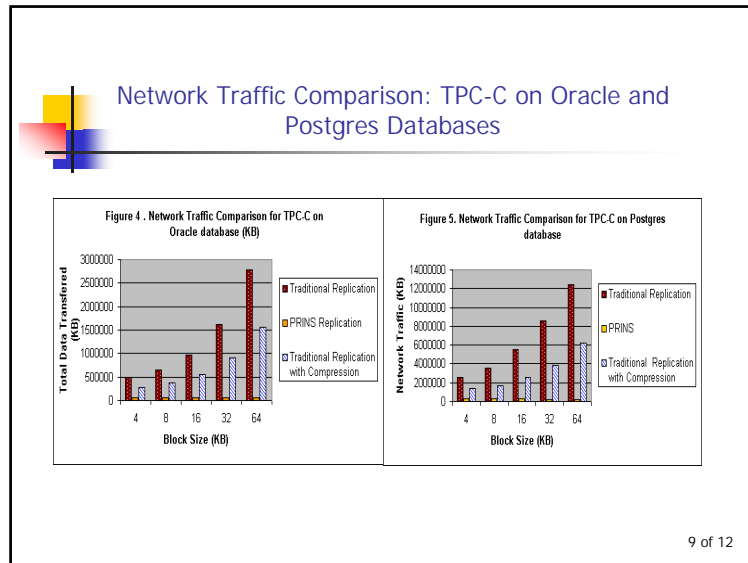


7 of 12

## Evaluation Methodology

- Measurement on Real Implementation using iSCSI protocol
- Workloads:
  - TPC-C, TPC-W, on Oracle, Postgres, MySQL Databases
  - File system micro benchmarks on MS and Linux

8 of 12



- ### Conclusions
- A New Data Replication Methodology: **PRINS**
  - Prototype Implementation
  - Measurements using real world workloads
  - 2 orders of magnitudes BW savings
- 12 of 12

## Software-based Failure Detection in Programmable Network Interfaces

Yizheng Zhou, Vijay Lakamraju, Israel Koren, C.M. Krishna  
Department of Electrical and Computer Engineering  
University of Massachusetts, Amherst, MA 01003  
E-mail: {yzhou, vlakamra, koren, krishna}@ecs.umass.edu

### Abstract

Nowadays, interfaces with a network processor and large local memory are widely used. The complexity of network interfaces has increased tremendously over the past few years. This is evident from the amount of silicon used in the core of network interface hardware. A typical dual-speed Ethernet controller uses around 10K gates whereas a more complex high-speed network processor such as the Intel IXP1200 [1] uses over 5 million transistors. This trend is being driven by the demand for greater network performance, and so communication-related processing is increasingly being offloaded to the network interface. As transistor counts increase dramatically, single bit upsets from transient faults, which arise from energetic particles, such as neutrons from cosmic rays and alpha particles from packaging material, have become a major reliability concern [2, 3], especially in harsh environments [4]. A sufficient amount of charge accumulated in transistor source and diffusion nodes may invert the state of a logic device – such as an SRAM cell, a latch, or a gate – thereby introducing a logical fault into the circuits operation. Because this type of fault does not reflect a permanent failure of the device, it is termed soft. Typically, a reset of the device or a rewriting of the memory cell results in normal device behavior thereafter. Soft-error-induced network interface failures can be quite detrimental to the reliability of a distributed system. The failure data analysis reported in [5] indicates that network-related problems contributed to approximately 40% of the system failures observed in distributed environments. The architects of programmable network interface

*should understand the impact of soft errors and select techniques to reduce this impact. Soft errors can cause the network interface to completely stop responding, function improperly, or even cause the host computer to crash/hang. Quickly detecting and recovering from such network interface failures is therefore crucial for a system requiring high reliability. We need to provide fault tolerance for not only the hardware in the network interface, but also the local memory of the network interface where the network control program (NCP) resides.*

*In this work, we propose an efficient software-based fault tolerance technique for network failures. Software-based fault tolerance approaches are highly attractive solutions, since they allow the implementation of dependable systems without incurring the high costs resulting from designing custom hardware or using massive hardware redundancy. On the other hand, software fault tolerance approaches impose some overhead in terms of reduced performance and increased code size. Since performance is critical for high-speed network interfaces, fault tolerance techniques applied to them must have a minimal performance impact.*


*Failure detection is based on a software-implemented watchdog timer to detect network processor hangs, and a software-implemented concurrent self-testing technique to detect non-interface-hang failures, such as data corruption and bandwidth reduction. The proposed self-testing scheme achieves failure detection by periodically directing the control flow to go through program paths in specific portions of the NCP in order to detect errors that affect instructions or data in the local memory as well as components of the network*

processor and other parts of the network interface hardware. The key to our technique is that the NCP is partitioned into various logical modules and only active logical modules are tested, where a logical module is defined as the collection of all basic blocks that participate in providing a service, and an active logical module is the one providing a service to a running application. When compared with testing the whole NCP, testing only active logical modules can limit significantly the impact of these tests on application performance while achieving good failure detection coverage. When a failure is detected by the watchdog timer or self-testing the host system is interrupted and informed about the failure. Then, a fault tolerance daemon is woken up to start a recovery process.

In this work, we show how the proposed failure detection techniques can be made completely transparent to the user. We demonstrate this technique in the context of Myrinet, but the approach is generic in nature, and is applicable to many modern networking technologies that have a microprocessor core and local memory.

## References


- [1] T. Halfhill. "Intel network processor targets routers," *Microprocessor Report*, vol. 13, No. 12, Sep. 1999.
- [2] J. F. Ziegler, et al., "IBM experiments in soft fails in computer electronics (1978 - 1994)," *IBM Journal of Research and Development*, vol. 40, No. 1, pp. 3 - 18, Jan. 1996.
- [3] S. S. Mukherjee, J. Emer, S. K. Reinhardt, "The Soft Error Problem: An Architectural Perspective," *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pp. 243-247, Feb. 2005.
- [4] A. V. Karapetian, R. R. Some, J. J. and Beahan, "Radiation Fault Modeling and Fault Rate Estimation for a COTS Based Space-borne Supercomputer," *IEEE Aerospace Conference Proceedings*, vol. 5, pp. 9-16, Mar. 2002.
- [5] A. Thakur, R. Iyer, "Analyze-NOW-an environment for collection of analysis of failures in a network of workstation," *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, pp. 14, Oct. 1996.



## Software-based Failure Detection in Programmable Network Interfaces

Yizheng Zhou, Vijay Lakamraju, Israel Koren, C.M. Krishna

Architecture and Real-Time Systems (ARTS) Lab  
Department of Electrical & Computer Engineering  
University of Massachusetts at Amherst



## Introduction

- Complex network interfaces
  - Typical Ethernet controller: 10 thousand gates
  - IXP1200: 5 million gates
- Transient faults: a major reliability concern
  - Neutrons from cosmic rays
  - Alpha particles from packaging material
- Software-based fault tolerance approaches
  - Pros: Less expensive than
    - Custom hardware
    - Massive hardware redundancy
  - Cons: Overhead
    - Performance degradation
    - Increased code size

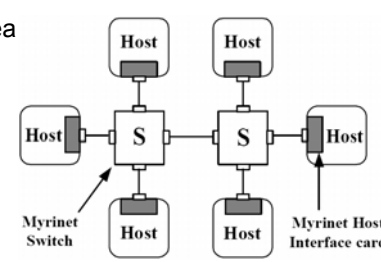
2 of 12

## Software-Based Failure Detection

- Network interface failures
  - Hardware failures
  - Software failures
    - The instruction and data of the Network Control Program (NCP) in the local memory.
- Requirements for failure detection of network interfaces
  - Limited performance impact
    - Performance is critical for high-speed network interface
  - Good failure coverage

3 of 12

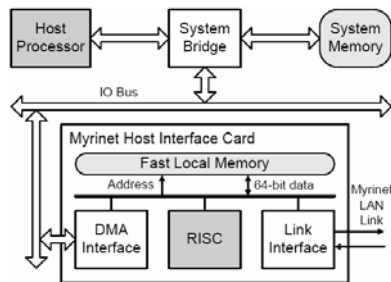
## Myrinet: An Example High-speed Network Interface

- A cost-effective local area network technology
  - High bandwidth: ~2Gb/s
  - Low latency: ~6.5μs
- Components in an example Myrinet LAN:
 

4 of 12

## Simplified Block Diagram of The Myrinet Network Interface

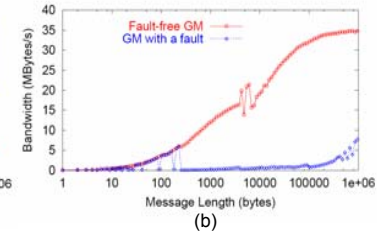
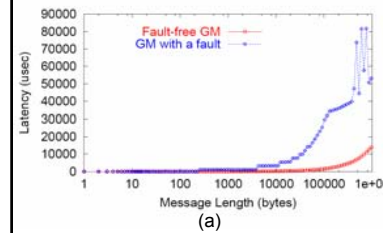
- Instruction-interpreting RISC processor
- DMA interface
- Link interface
- Fast local memory (SRAM)



5 of 12

## Network Interface Failures

- Transient faults in the form of random bit flips in the network interface
- Failures observed:
  - Network interface hangs
  - Corrupted control information
  - Send/Receive failures
  - Corrupted messages
  - DMA failures
  - Unusually long latency



6 of 12

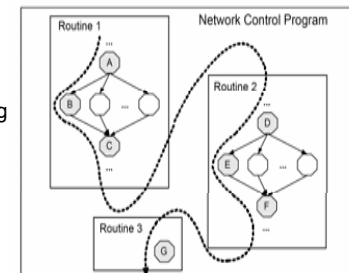
## Failure Detection Strategy

- Interface hangs
  - Software watchdog timer
- Other failures
  - A useful observation: applications generally use only a small portion of the NCP
    - *Directed Delivery*: used for tightly-coupled systems, allows direct remote memory access
    - *Normal Delivery*: used for general systems, allows reliable ordered message delivery
    - *Datagram Delivery*: delivery is not guaranteed
  - Adaptive Concurrent Self-Testing (ACST)
    - Test only part of the NCP
    - Avoids testing & signaling benign faults
    - Can detect hardware & software failures

7 of 12

## Logical modules

- Identify the “active” parts
- Logical module:
  - The collection of all basic blocks that might participate in providing a service
- To test a logical module:
  - Trigger several requests/events to direct the control flow to go through all its basic blocks



8 of 12



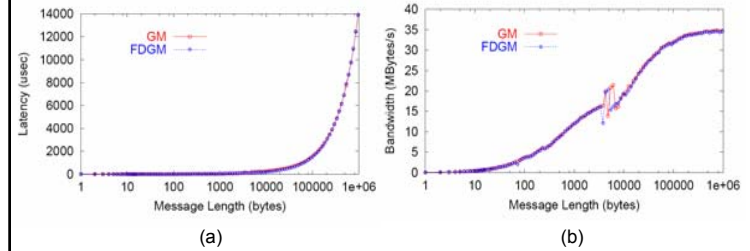
## Experimental Results: Failure Coverage

- Exhaustive fault injection into a single routine: send\_chunk
- Exhaustive fault injection into special registers
- Random fault injection into the entire code segment

	Coverage	No impact
Routine: send_chunk	99.3%	60.3%
Registers	99.2%	32.3%
Entire code segment	95.6%	93.9%

9 of 12

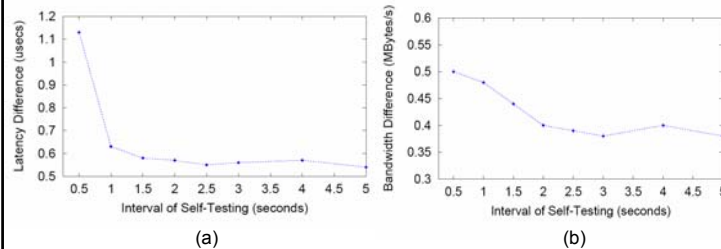
## Performance Impact



- The original Myrinet software: GM
- The modified Failure Detection GM: FDGM
- The MCP-level self-testing interval is set to 5 seconds

10 of 12

## Performance Impact For Different Self-Testing Intervals



- Message length is 2KB
- For the half-second interval
  - bandwidth is reduced by 3.4%
  - latency is increased by 1.6%

11 of 12

## Conclusion

- The proposed ACST tests only active logical modules
- Failure coverage: over 95%
- No appreciable performance degradation
- Transparent to applications
- The basic idea is generic – applicable to other fast network interfaces

12 of 12

# Software Fault Detection Using Dynamic Instrumentation

George A. Reis David I. August

Depts. of Electrical Engineering and Computer Science  
Princeton University

{gareis,august}@princeton.edu

Robert Cohn Shubhendu S. Mukherjee

FACT and VSSAD Groups  
Intel Massachusetts

{robert.s.cohn,shubu.mukherjee}@intel.com

In recent decades, microprocessor performance has been increasing exponentially, due in large part to smaller and faster transistors enabled by improved fabrication technology. While such transistors yield performance enhancements, their lower threshold voltages and tighter noise margins make them less reliable [1], rendering processors that use them more susceptible to *transient faults*. Transient faults are intermittent faults caused by external events, such as energetic particles striking the chip, that do not cause permanent damage but may result in incorrect program execution by altering signal transfers or stored values.

To detect or recover from these faults, designers typically have introduced redundant hardware. For example, storage structures such as caches and memory often include error correcting codes (ECC) or parity bits while techniques like lockstepping or Redundant Multithreading [4] have been proposed for full processor protection. Although these techniques are able to increase reliability, they all require changes to the hardware design.

Software-only approaches to reliability have been proposed and evaluated as alternatives to hardware modification [3, 5]. These techniques have shown that they can significantly improve reliability with reasonable performance overhead and no hardware modifications.

Since software-only techniques do not require any hardware support, they are far cheaper and easier to deploy. In fact, these techniques can be used for systems that have already been manufactured and now require higher reliability than the hardware alone can offer. This need can occur because of poor estimates of the severity of the transient error problem or changes in the environment, such as moving to higher altitudes.

Software-only approaches also benefit from reconfigurability after deployment. Since reliability is achieved via software, the system can dynamically configure the trade-off between reliability and performance. Software techniques can be configured to only add reliability in certain environments, for specific applications, or even for critical regions of an application, thus maximizing the reliability while minimizing the costs.

Although software-only error mitigation techniques do exist, previous proposals have been static compilation techniques that rely on alterations to the compilation process. Our proposal is the first use of software fault detection for transient errors that increases reliabil-

ity dynamically. Our proposal uses a modified version of the PIN dynamic instrumentation framework [2] to enact the reliability transformations.

Using dynamic instrumentation, rather than static compilation, to increase reliability is advantageous for a number of reasons. Since the only requirement is the program binary, it is applicable to legacy programs that no longer have readily available or easily re-compilable source code. Even if the application sources are available, users typically do not recompile libraries (such as `libc`) when recompiling an application.

While it is possible to create a binary translator that enhances reliability in some cases, our dynamic reliability technique can seamlessly handle variable-length instructions, mixed code and data, statically unknown indirect jump targets, dynamically generated code, and dynamically loaded libraries. Our technique can also attach to already running applications to increase reliability.

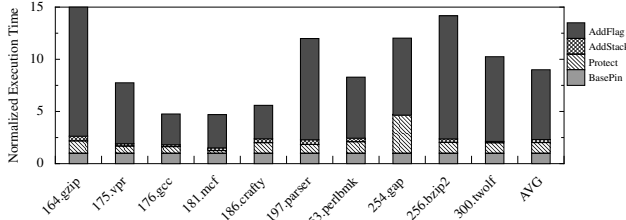
We base our fault detection implementation on the SWIFT software-only reliability technique [5], applying computational redundancy and detection to the x86 instruction set. We have implemented the dynamic translations for reliability and have evaluated some performance enhancements to our technique. Future work includes a thorough evaluation of the fault coverage as well as explore the relationship between reliability and performance when certain code regions are left unprotected.

Our technique dynamically duplicates all instructions, except for those that write to memory. Since a fault causing data corruption will only manifest itself as a program error if it changes the output, we delay validation until instructions that may affect output, such as stores. This ensures that we will not flag an error in a dynamically dead register or one whose value will be masked away. Also, our technique does not duplicate load instructions, but to maintain reliable execution, copies the loaded value into a redundant register as shown in previous work [4, 5].

Figure 1 is a simple example to illustrate the instruction duplication and verification of our technique. Instruction **1** is inserted to add redundancy to the data loaded from memory by copying the value to its duplicate virtual register. Instruction **2** is inserted to redundantly compute the subtraction and instructions **3-6** verify that both the address and value sources of the store instruction are fault-free.

<pre> mov (%edx), %eax sub %eax, %ebx mov %ebx, (%edx) </pre>	<pre> mov (%edx), %eax 1: mov %eax, %eax2 sub %eax, %ebx 2: sub %eax2, %ebx2 3: cmp %edx, %edx2 4: jne faultDetect 5: cmp %ebx, %ebx2 6: jne faultDetect mov %ebx, (%edx) </pre>
(a) Original Code	(b) Reliable Code

**Figure 1. Duplication and Validation**



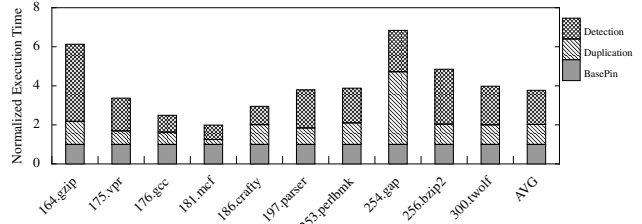
**Figure 2. Performance for duplication only, accounting for specialized registers.**

Our reliability enhancements were implemented in the PIN dynamic instrumentation framework [2]. Redundant instructions, as well as validation and copy instructions, are inserted during dynamic instrumentation. We use the existing PIN framework to register allocate the additional code, as well as perform other basic optimizations like data liveness analysis. Due to current limitations with register allocation in the PIN tool, we do not duplicate floating point or multimedia instructions, but this is part of our future work to increase reliability.

To analyze the performance of our reliability approach, we first calculated the cost of duplicating instructions without data verification. We compared the execution relative to a base PIN execution with no reliability or instrumentation tools. We ran all SPECINT2000 executions using reference inputs.

We found that the execution time of the reliable code is dominated by the duplication of the EFLAGS register. Figure 2 shows the normalized execution times of duplicating instruction with and without duplicating the stack pointer and EFLAGS registers. The average normalized executing time without the EFLAGS register is 2.31x slower than the base, but by protecting that one register, the time increases to 9.00x. Duplicating the EFLAGS register is extremely expensive due to the restricted manner in which it may be fully accessed. It can only be completely moved to and from the memory stack, whereas non-EFLAGS registers can be moved into other architectural register. In addition, moving the entire EFLAGS register is a very expensive operation.

The stack pointer is the second most expensive register to duplicate, bringing the normalized execution time from 2.02x to 2.31x. This degradation is mainly due to



**Figure 3. Performance of detection compared to duplication.**

instructions that implicitly read from or write to the stack pointer, making the allocation of the two virtual stack pointers is limited.

Instruction duplication adds the redundancy necessary for independent computation, but comparison of the independent versions is necessary for fault detection. Figure 3 shows the normalized performance for full detection, attributing the performance costs for duplication and verification. These executions use only a single version of the stack pointer and EFLAGS register.

On average, the normalized execution time for instruction duplication alone is 2.02x while duplication plus data verification is 3.77x. The per benchmark degradations vary, ranging from 254.gap with a cost of 6.84x to 181.mcf with a cost only 1.99x. Benchmarks like 181.mcf which contain many cache misses have extra instruction level parallelism to execute the redundant and detection instructions without affecting the critical path.

Our technique shows that a dynamic software-only approach to reliability is possible with acceptable performance degradation. Our future work targets ways to further increase performance of the reliable execution through smarter register allocation and scheduling. We also plan to simulate fault injections to determine the precise fault coverage, which will guide the dynamic trade-off between reliability and performance.

## References

- [1] R. C. Baumann. Soft errors in advanced semiconductor devices-part I: the three radiation sources. *IEEE Transactions on Device and Materials Reliability*, 1(1):17–22, March 2001.
- [2] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.
- [3] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. In *IEEE Transactions on Reliability*, volume 51, pages 63–75, March 2002.
- [4] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 25–36. ACM Press, 2000.
- [5] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.

## Software Fault Detection Using Dynamic Translation

George A. Reis<sup>1</sup> · David I. August<sup>1</sup>  
Robert Cohn<sup>2</sup> · Shubhendu S. Mukherjee<sup>2</sup>

<sup>1</sup>Liberty Research Group  
Princeton University

<sup>2</sup>FACT and VSSAD Groups  
Intel Massachusetts

February 3, 2006

<http://liberty.princeton.edu>

## Transient faults

- Hardware faults
  - Different than design or manufacturing faults
  - Cannot test for fault before hardware use
  - Hardware is not permanently damaged
- Caused by external energetic particle striking chip
- Randomly change one bit of state element or computation

 0x8675309  
\* 0x42  
0x32AA36852

<http://liberty.princeton.edu>

2 of 16

## Severity of transient faults

- IBM historically adds 20-30% additional logic for mainframe processors for fault tolerance [Siegel 1999]
- In 2000, Sun server systems deployed to America Online, eBay, and others crashed due to cosmic rays [Baumann 2002]
- In 2003, Fujitsu released SPARC64 with 80% of 200,000 latches covered by transient fault protection [Ando 2003]
- "it was found that a single soft fail ... was causing an entire interleaved system farm (hundreds of computers) to crash." [SER: History, Trends, and Challenges 2004]
- Los Alamos National Lab ASC Q 2048-node supercomputer was crashing regularly from soft faults due to cosmic radiation. [Michalak 2005]
- Processors are becoming more susceptible
  - lower voltage thresholds
  - increased transistor count
  - faster clock speeds

<http://liberty.princeton.edu>

3 of 16

## Goals

- Develop transparent (to user) way to increase reliability, specifically targeting soft errors, without any hardware requirements.
- This can be used to increase the reliability of currently deployed systems.

<http://liberty.princeton.edu>

4 of 16

Software Fault Detection Using Dynamic Instrumentation

## Mitigation of transient faults

- Levels to add reliability
  - Circuit, Logic, Microarchitectural, Architectural, Application
- Hardware techniques
  - Lockstepping processors [Compaq Himalaya]
  - Redundant multithreading (RMT) [Reinhardt & Mukherjee, 2000]
- Software techniques
  - NMR, TMR
  - Source-to-source [Rebaudengo et al. 2001]
  - SWIFT [Reis et al. 2005], EDDI, CFCSS [Oh et al. 2002]

<http://liberty.princeton.edu> 5 of 16

Software Fault Detection Using Dynamic Instrumentation

## Dynamic Software Translation

- Software techniques
  - Can be applied today to existing applications on existing hardware
- Binary translation
  - Can be applied without recompilation
    - legacy binaries with no source code
    - compilation of included libraries
- Dynamic binary translation
  - Can attach to running application (and later detach)
  - Can easily handle:
    - Variable-length instructions
    - Mixed code and data
    - Statically unknown indirect jump targets
    - Dynamically generated code
    - Dynamically loaded libraries

<http://liberty.princeton.edu> 6 of 16

Software Fault Detection Using Dynamic Instrumentation

## Store Protection

If a tree falls in the forest,  
but nobody is around to hear it,  
does it make a sound?

If a fault affects some data,  
but does not change the output,  
does it make an error?

Only store operations affect output,  
so validate data before stores.

<http://liberty.princeton.edu> 7 of 16

Software Fault Detection Using Dynamic Instrumentation

## Our implementation

- Create single-threaded, software-only version of RMT
  - Add redundant instruction
  - Add verification before memory accesses
  - Add duplication of loaded values
- Use PIN's dynamic instrumentation infrastructure
- Implemented for x86
  - Only 8 registers available
    - register pressure is big issue
  - Implicit register operands (PUSH, SAL)
    - add more constraints to register allocation
    - EFLAGS is frequently used

<http://liberty.princeton.edu> 8 of 16

### PIN Reliability Transform

FORMAT: OP DEST1 DEST2 = SRC1 SRC2

Compare before memory instruction

```

CMP EFLAGS'' = EAX EAX'
JNZ EIP
= 0xFAULTDETECT EIP EFLAGS''
    
```

Duplicate non-memory instruction

```

MOV (DS:EAX) = 0x00000000
INC EDX EFLAGS = EDX
INC EDX' EFLAGS' = EDX'
ADD EAX EFLAGS = EAX 0x04
ADD EAX' EFLAGS' = EAX' 0x04
CMP EFLAGS = EDX 0x4a
CMP EFLAGS' = EDX' 0x4a
JBE EIP
= 0xABCDABCD EIP EFLAGS
    
```

<http://liberty.princeton.edu> 9 of 16

### PIN Reliability Transform - Loads

Compare before memory instruction

```

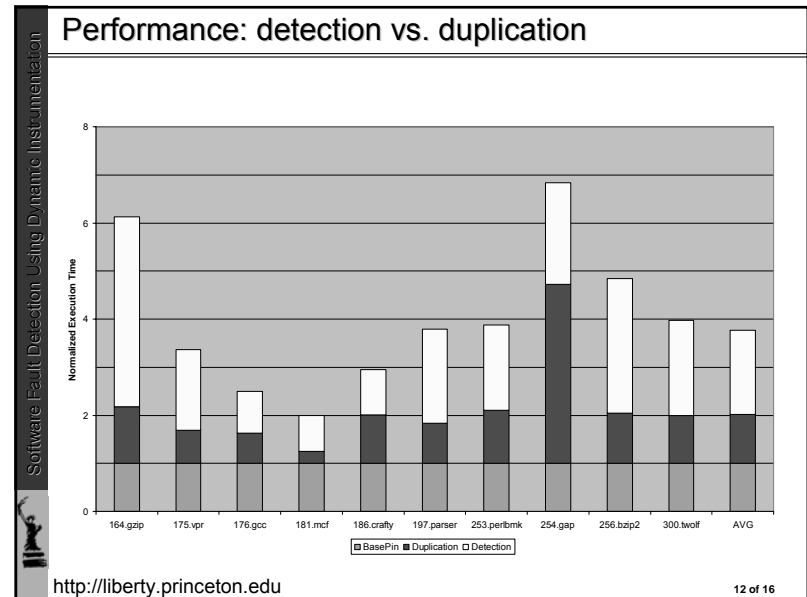
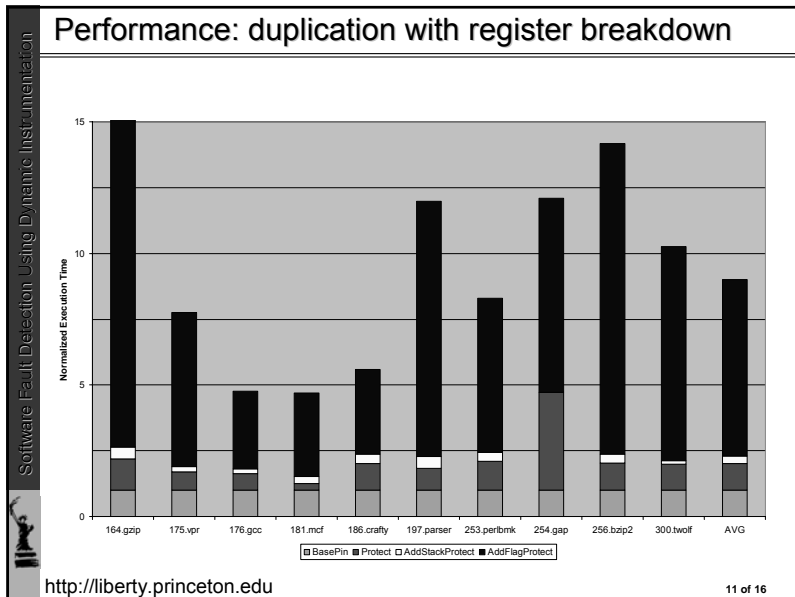
CMP EFLAGS'' = ESP ESP'
JNZ EIP
= 0xFAULTDETECT EIP EFLAGS''
    
```

Duplicate loaded value

```

MOV EAX = (ESP, 0xffffd64)
MOV EAX' = EAX
. . .
    
```

<http://liberty.princeton.edu> 10 of 16



Software Fault Detection Using Dynamic Instrumentation

### Just the beginning...

- Register allocation
  - Can greatly reduce overhead via more sophisticated algorithm
  - Increase reliability by protecting MMX, FP registers
- Persistence Pin [Janapa Reddi WBIA-2005]
  - Cache (on disk) the instrumented code
  - Eliminate most of dynamic translation cost
- Running on x86-64
  - More available registers will decrease overhead due to spill/fill
- Fault injection to determine error coverage

<http://liberty.princeton.edu>

13 of 16

Software Fault Detection Using Dynamic Instrumentation

### Just the beginning... Error coverage

preliminary results

Benchmark	Configuration	Correct (%)	Fault Detected (%)	Faulty Output (%)
164.gzip	No FT	78	22	0
	Reliable	78	12	10
181.mcf	No FT	78	22	0
	Reliable	75	15	10
183.equake	No FT	70	30	0
	Reliable	48	42	10
300.twolf	No FT	78	22	0
	Reliable	75	15	10

<http://liberty.princeton.edu>

14 of 16

Software Fault Detection Using Dynamic Instrumentation

### Just the beginning... Software-modulated Fault Tolerance

- Dynamic translation can make different decisions for different code regions, and can change over time
  - Programs
  - Functions
  - Individual store dependence chains
- Programs have varying level of importance
- Programs have varying level of natural fault resistance
- Output corrupting faults have varying severity

original jpegenc output    faulty jpegenc output    faulty? jpegenc output

<http://liberty.princeton.edu>

15 of 16

Software Fault Detection Using Dynamic Instrumentation

### Just the beginning... Software-modulated Fault Tolerance

- Software flexibility allows tradeoff between performance and reliability
- Tune redundancy based on function reliability and performance response
- Example: changes in reliability and execution time for different function of 124.m88ksim

<http://liberty.princeton.edu>

16 of 16

# Self-healing Nanoscale Architectures on 2-D Nano-fabrics

Teng Wang, Mahmoud Ben Naser, Yao Guo, Csaba Andras Moritz  
Department of Electrical and Computer Engineering  
University of Massachusetts, Amherst, MA 01003

## 1 Introduction

One of the most promising underlying nanodevice technologies today for nanoscale integrated circuits is semiconductor nanowires (NWs) and arrays of crossed NWs. Researchers have already built FETs and diodes out of NWs [3]. Complementary depletion-mode FETs in the same material have been demonstrated with Germanium and Silicon. There has also been a lot of progress made on assembling arrays with such devices with both unconventional lithographic techniques and bottom-up self-assembly. The rapid progress on devices is driving researchers to explore possible circuits/architectures out of them. Examples of proposed architectures include [1, 4, 2, 6].

The fabric architecture we proposed is called NASIC (Nanoscale Application-Specific IC) [6]. WISP-0 (Wire Streaming Processor) is a simple but complete stream processor that exercises many of different NASIC circuit styles and optimizations. These previous efforts focused on circuit level optimizations that set apart our work from the other nanoscale proposals. In this paper we focus on another key distinguishing aspect: our strategy for fault tolerance. As discussed by several researchers, fault tolerance is expected to be a key issue in nanoscale designs.

Our solution for fault tolerance of NASICs is based on built-in circuit-level redundancy which makes the circuits in our designs self-healing.

## 2 Overview of NASIC Designs and WISP-0 Processor

NASIC designs use FETs on 2-D semiconductor NWs to implement logic functions and various optimizations to work around layout and manufacturing constraints as well as defects. While still based on 2-level AND-OR logic style, our designs are optimized according to specific applications to achieve higher density and self-healing. NASIC circuits are based on a new

type of dynamic circuitry [5].

WISP-0 is a stream processor (based on self-healing NASIC circuits) that implements a 5-stage pipeline architecture including *fetch*, *decode*, *register file*, *execute* and *write back*. WISP-0 consists of five nanotiles, as shown in Figure 1. In WISP designs, in order to preserve the density advantages of nanodevices, data is streamed through the fabric with minimal control/feedback paths. With the help of dynamic Nano-latches [5], intermediate values during processing are often stored on the wire without requiring explicit latching. The compiler is responsible of generating code such that data hazards are avoided.

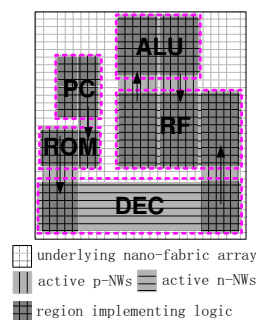


Figure 1. The floorplan of WISP-0.

## 3 Fault Tolerance Strategy

Nanoscale computing systems face challenges not encountered in the world of traditional microelectronic devices. Researchers have pointed out that the defect levels in nano-fabrics tend to be quite high: we have to build enough fault-tolerance to sustain functionality when a substantial fraction of circuits are faulty.

There are basically two main approaches to deal with faults. First, if reconfigurable devices are available, we could possibly devise techniques to work around faults. One key challenge in such solutions is, however, accessing crosspoints in the fabric. That requires a special in-



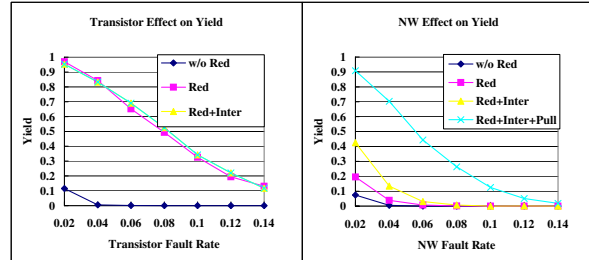
interface between the micro and the nanodevices and such an interface is not only presenting a high area overhead but it is also very difficult to do due to the required alignment between the nano and the micro wires. No proposals with exception of perhaps CMOL [4] (that has other challenges) address this issue in a credible way.

Alternatively, as proposed by this work, we can make the circuits and the architectures self-healing. This of course requires some sort of added redundancy. The redundancy is achieved by replicating NWs and transistors in our designs. Most faults can be automatically masked by the AND-OR logic either in the current stage or the next one. For example, a faulty “0” can be automatically masked by the following OR plane and faulty “1” can be masked by the following AND plane. Interleaving replicated NWs further improves the efficiency of tolerating faults. There are cases that AND-OR logic can not handle directly. To solve this problem, we propose to insert *weak* pull-up/down NWs between AND and OR logic planes. With pull-up/down NWs, we improve the fault-tolerance of our self-healing circuits considerably with the cost of some speed reduction.

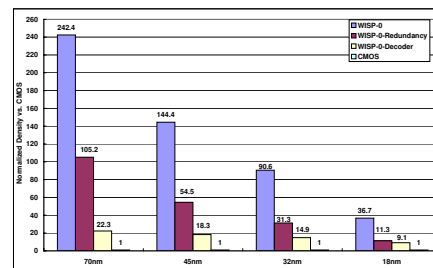
## 4 Results

We base our evaluation on self-healing WISP-0. We apply several of our fault tolerance techniques and explore both the impact of faulty transistors and broken NWs. As shown in Figure 2, our built-in redundancy works very well and the techniques combined improve yield considerably. Without our self-healing mechanism, the yield goes down rapidly to 0 for even a very small defect rate. As such, with the exception of compensating faults, without fault tolerance the presence of faults cause incorrect execution. For self-healing NASICs, however, even if the defect rate of transistors or NWs is 10%, the yield remains over 10%. Of course, if combined with system-level approaches, we can likely improve the yield further.

We have also estimated the impact of our self-healing techniques on density. As shown in Figure 3, the redundancy techniques cause around 2X area overhead for WISP-0 (see “WISP-0” and “WISP-0-Redundancy” in Figure 3). However, our solution eliminates the need of accessing each crosspoint for reconfiguration. The density of WISP-0 with redundancy is in fact better than the density without redundancy but with a micro-nano decoder. Even at 18-nm CMOS, available in 12 years according to ITRS 2005, our self-healing design would still be over 10X denser than equivalent processor design in 18-nm CMOS.



**Figure 2.** The yield achieved under different configurations (*Red* means WISP-0 with redundancy. *Inter* means interleaving of NWs. *Pull* means applying weak pull-up/down NWs). The left figure shows the impact of faulty transistors and the right one shows the impact of broken NWs.



**Figure 3.** Normalized density of WISP-0 under different configurations compared with equivalent designs of 70, 45, 32, 18-nm CMOS technology.



## References

- [1] A. DeHon. Array-based architecture for FET-based, nanoscale electronics. *IEEE Transactions on Nanotechnology*, 92(1), 2003.
- [2] S. C. Goldstein and M. Budiu. Nanofabrics: Spatial computing using molecular electronics. In *The 28th Annual International Symposium on Computer Architecture, ISCA'01*, 2001.
- [3] Y. Huang, X. Duan, Y. Cui, L. Lauhon, K.-Y. Kim, and C. Lieber. Logic gates and computation from assembled nanowire building blocks. *Science*, 1313(294), 2001.
- [4] K. K. Likharev. CMOL: Devices, circuits, and architectures. *Introducing Molecular Electronics*, 2004.
- [5] C. A. Moritz and T. Wang. Latching on the wire and pipelining in nanoscale designs. In *Non-Silicon Computing Workshop, NSC-3*, 2004.
- [6] T. Wang, M. Bennaser, Y. Guo, and C. A. Moritz. Wire-streaming processors on 2-D nanowire fabrics. In *Nanotech 2005*. Nano Science and Technology Institute, May 2005.

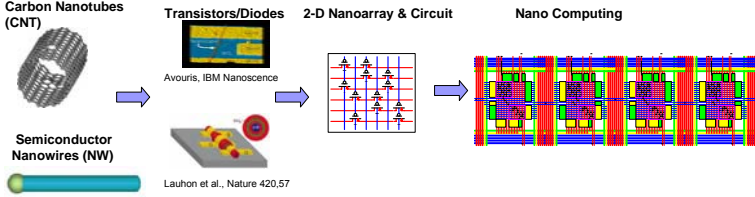
# Self-Healing Nanoscale Architectures on 2-D Nano-fabrics

Teng Wang and Csaba Andras Moritz  
University of Massachusetts, Amherst  
[twang@ecs.umass.edu](mailto:twang@ecs.umass.edu)

February 3, 2006


## From Devices to Nano Computing



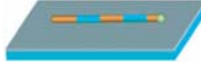
- We are trying to answer questions like
  - What are the challenges when building nanoscale circuits and architectures?
  - Can the density advantages of nanodevices be preserved at system level?
  - What would be the capabilities of such systems compared to CMOS?
  - Influence device/manufacturing research

Copyright - Teng Wang, ECE, UMass Amherst 2 of 12


## Self Assembly of FETs and Metallic Interconnects on Nanoarray



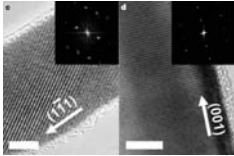
Metalize NWs



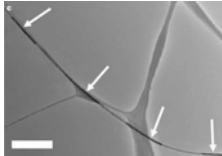
Form NiSi segments



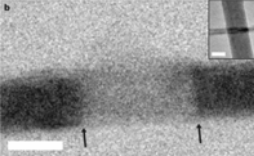
NWs as masks



Scale bar: 5nm



Scale bar: 1um

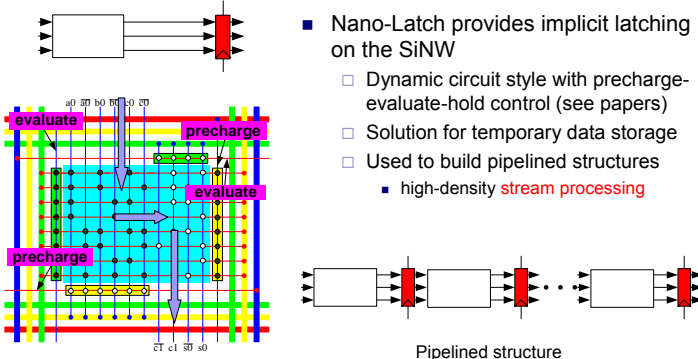


Scale bar: 10nm

Wu et al., Nature Vol. 430, pp. 61, 2004

Copyright - Teng Wang, ECE, UMass Amherst 3 of 12

## Dynamic NASIC Tile and Pipeline

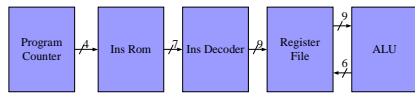
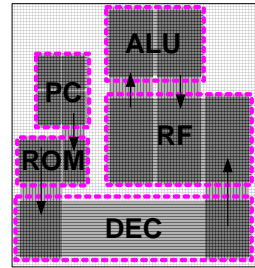


- Nano-Latch provides implicit latching on the SiNW
  - Dynamic circuit style with precharge-evaluate-hold control (see papers)
  - Solution for temporary data storage
  - Used to build pipelined structures
    - high-density **stream processing**

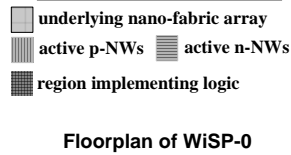
Copyright - Teng Wang, ECE, UMass Amherst 4 of 12

# Architecture of WiSP-0

- WiSP-0 is the initial version of WiSP.
  - Supports simple ISA: nop, movi, mov, add, mul
  - Hazards exposed to compiler
  - Implements 5-stage pipeline on 5 NASIC nanotiles

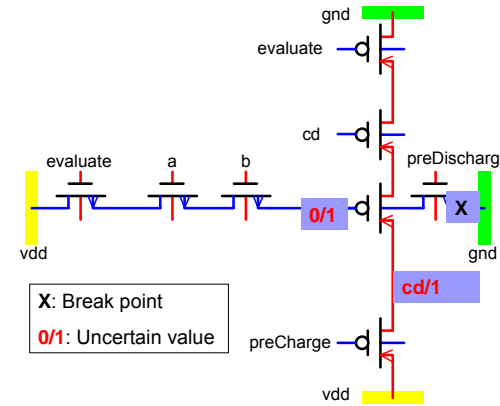


Schematic of WiSP-0



Floorplan of WiSP-0

# NASICs without Fault Tolerance



Logic:  $f=ab+cd$

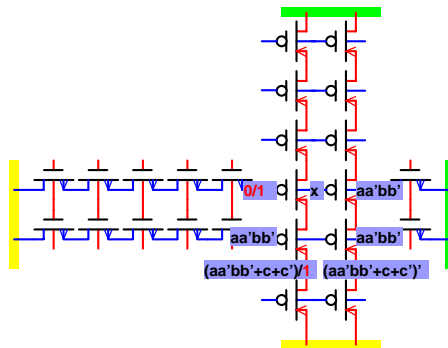
Without fault tolerance:

- Any fault can make the whole nanotile faulty
- We explored several approaches
- Built-in redundancy

X: Break point  
0/1: Uncertain value

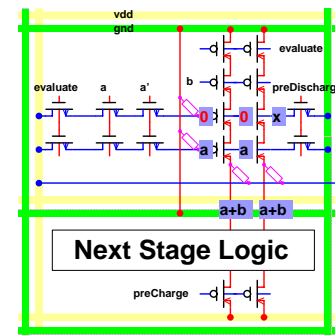
# 2-level Redundancy – Example

With duplicated rows:



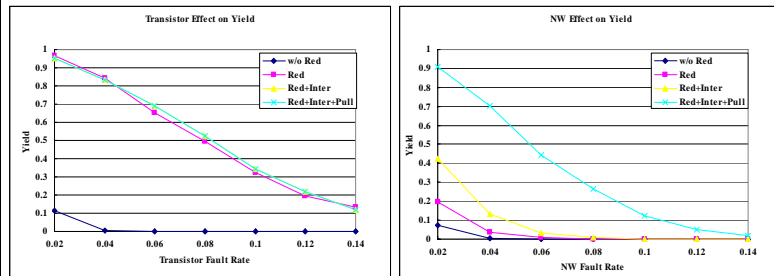
- Breaks between duplicated columns are masked by AND plane in the next stage
- Similar for breaks on the left from columns

# Pull-up/down NW for Fault Tolerance



- Weak pull-up/down NWs for the case that 2-level redundancy can not handle
- Tradeoff – better fault tolerance with lower speed

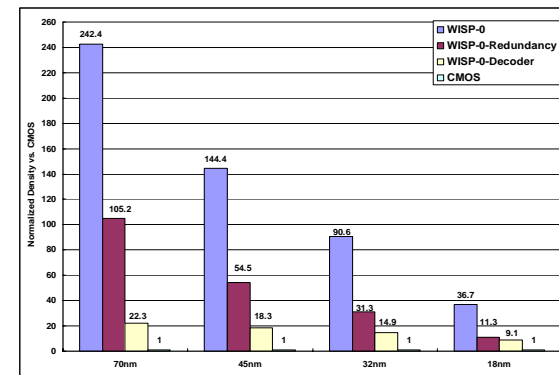
## Defect Effect on Yield



Copyright - Teng Wang, ECE, UMass Amherst

9 of 12

## Comparison with CMOS



Copyright - Teng Wang, ECE, UMass Amherst

10 of 12

## Conclusions

- Self-healing technique improves the yields of NASICs considerably.
- Self-healing technique eliminates the needs of decoder for reconfiguration, defect map extraction, and micro-nano alignment
- Self-healing NASICs have great density advantage over deep sub-micron CMOS technology.

Copyright - Teng Wang, ECE, UMass Amherst

11 of 12

# End

Copyright - Teng Wang, ECE, UMass Amherst

12 of 12

# Functional Programming in Embedded System Design

Al Strelzoff  
E-TrolZ, Inc.  
Lawrence, Mass.  
al.strelzoff@e-trolz.com

A concept for Embedded System Design is presented based on an adaptation of functional programming to hard real time systems. The goals are the development of a highly dependable embedded computing environment that will be free from many common programming errors, and possibly be capable of applying formal checking methods. An integrated approach to both language and operating system is included. An additional goal is to remove the distinction between hardware and software so that a program can be run on any mix of hardware and software without re-compilation. That is, the program design is to be independent of the underlying computing fabric. This leaves a "back-end" problem of mapping the computation to a specific fabric which in the case of multi-processing can be a hard problem. This paper deals with running a "soft" program on a small, simple processor that can execute the software portion of a program as a soft core within an FPGA. The hardware would be realized directly within the remaining gates. The language is made to look as much like C, C++, Java, and Verilog as possible to facilitate adoption.

We will refer to the language presented in this paper as V, for verifiable[1]. The starting principle is that V should have cycle based, repetitive execution semantics. This is sometimes referred to in the literature as Synchronous execution[2][3], and can be related to the sampling theorem. The language then carries with it a model of its own operating system which is very simple. A program is compiled, run in simulation, and then downloaded to a target for instantiation. The program can then be initialized after which it runs in a continuous loop. There is no garbage collection, because no garbage is created. The idea is to create a program that is statically analyzable in order to reduce run-time errors to a minimum.

The program is executed repeatedly and consists of a net-list of functions. The outermost

function is the program itself, which contains a net-list of inner functions. This "fractal" picture continues down to a set of primitive functions that can not be further de-composed. It could be depicted in the form of a synchronous dataflow graph. Synchronous dataflow can be statically scheduled at design time, whereas Asynchronous dataflow[4] is scheduled at run time.

Synchronous dataflow closely resembles the way hardware works and it can be translated to synthesizable Verilog[5] and then hardware.

The execution semantics are as follows. On each cycle of the program, it accepts a set of inputs, and produces a set of outputs. The data flows through the program progressing a step at a time. Intermediate values are held in memory (registers) inserted into some of the functions.

On each cycle, all inputs and data stored in registers are presented to the inputs of all functions. Then these functions execute. Finally, all outputs are made and intermediate results stored away. All registers (which store intermediate results) can be read as many times in the program as desired, but can be written only once. This is a "single assignment" rule[6]. An assignment does not update a register until the end of each execution cycle. Thus all registers are read in a "coherent" fashion.

The syntax can look much like pointer-less C or Java. A compiler converts the mixture of infix operator and prefix function expressions to postfix. This compiled intermediate code is executed on a stack machine that is very similar to that of a Forth machine. The difference is that the user does not see the postfix stack instructions, and in addition, V is strongly typed. The type of a variable as well as its value is kept at run time. The stack machine is very simple compared with conventional register based processors and appears a likely target for a small FPGA soft core.

The V language is functional in that it is very similar to Haskell and pH[6]. Hudak[7] and co-workers have applied Haskell to real time systems. But unlike Haskell, V is not executed

with lazy evaluation. And unlike pH, the memory locations in V are non-blocking because V is synchronous. V is also inspired by Sisal[8], an early attempt at a real time functional language. But in most respects, V is like Haskell. For example, full recursion is supported. The stack machine is rather efficient at this. It may also be possible to convert some kinds of recursion to loops in the compiler.

To generate hardware, the postfix expression can be unwound by a special kind of "interpreter" which instead of executing the program generates structural Verilog.

The V type system is similar to Haskell and supports user defined types and a restricted form of object oriented design. The user can define Type Classes which consist of a list of member Types. A Type Class is abstract and can not itself be instantiated. It contains a list of members and a list of those functions which its members are required to supply (a bit like an interface), but it is up to the members to supply them. This leads to a form of inheritance where the Type Classes can have parents and children. A form of object oriented design based on this kind of a type system has been described by Jackson and co-workers[9]. V implements a subset of the type system described there.

Type Classes can have Types as members which can be instantiated. The Types must supply all the required functions and fields as required by their parents. They are not overwriting virtual methods, they are supplying required methods. A Type must also supply a constructor which contains a list of all their properties that are going to be instantiated. Types can only be instantiated during the initialization portion of a program, never during the main loop of the program itself.

Types can be further sub-typed themselves, but only by providing relations among their properties, not by providing new properties. There is no over writing of functions and properties. Run-time polymorphism is supported; the arguments of a function can be that of a Type Class while at run-time, a specific member of that class can be actually passed as an argument. There are also no class casts. It remains to be seen if this limited version of object oriented design will be powerful enough to support the design of useful embedded system applications.

The existence of persistent storage that holds data between execution "ticks" supports finite state object modeling. The easy creation of Finite State Machine models is a useful feature

of this system. Not all portions of the program have to "tick" at the same rate, it is possible to have multiple sub-programs each "ticking" at a different rate. The different sub-programs are then connected by what amounts to rate adapting filters. So it will be possible to support a complex system model of concurrent Finite State Machines.

Dependability of systems designed in this manner appears likely as there is no possibility to get null pointers, class cast exceptions, and so forth. However, arithmetic exceptions still remain and have to be dealt with carefully. A next step for this project is to determine the degree to which formal checking can be applied to the verification of such a system before it is even run.

## References

- [1] A. Strelzoff, "Functional Programming of Reconfigurable Computing Arrays for Real-Time Embedded Systems", Proceedings of the Reconfigurable Array Workshop, Inderscience press, 2004.
- [2] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," Proceedings of the IEEE 79, 1270-1282, 1991.
- [3] J.N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, "The Synchronous Programming Language LUSTRE", Proceedings of the IEEE 79, 1305-1320, 1991.
- [4] W. Böhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar, "Mapping a Single Assignment Programming Language to Reconfigurable Systems," Supercomputing, 21:117-130, 2002.
- [4] W. Ackerman, J. B. Dennis. VAL - A Value-Oriented Algorithmic Language. MIT/LCS/TR-218, June 1979.
- [5] Lionel Bening and Harry Foster, "Principles of Verifiable RTL Design, Second Edition", Kluwer, 2001.
- [6] R.S. Nikhil and Arvind, "Implicit Parallel Programming in pH", Morgan Kaufmann, 2001.
- [7] Walid Taha, Paul Hudak, Wan Zahnyong, "Functional Programming of Real Time Applications", Yale University Report, 2001.
- [8] Jean-Luc Gaudiot, et. al, "The Sisal Project - Real World Functional Programming", Lawrence Livermore report, 1997.
- [9] Jonathan Edwards, Daniel Jackson and Emina Torlak, "A Type System for Object Models", MIT Report, 2004.

## Functional Programming in Embedded System Design

Al Strelzoff  
CTO, E-TrolZ  
Lawrence, Mass.  
al.strelzoff@e-trolz.com

1/28/2006

Copyright E-TrolZ, Inc.

1 of 20

## Hard Real Time Embedded Systems MUST:

- Be highly Dependable. Zero failures.
- Do what they're supposed to do.
- Not ever crash.
- Integrate complex hardware and software.
- Ensure behavior by some means before they are run.

1/28/2006

Copyright E-TrolZ, Inc.

2 of 20

## What Designers Want

- No run time exceptions:
  - No null pointers.
  - No out of range arrays.
  - No class casts.
  - No arithmetic exceptions - most difficult.
- Well specified execution semantics.
- No distinction between hardware and software.
- Implicit parallelism.
- Compatibility with existing languages if possible.

1/28/2006

Copyright E-TrolZ, Inc.

3 of 20

## Look to the Basics of Computer Science

- Functional Programming. Examples: Haskell and pH.
- Build execution semantics into the language.
  - Cycle based execution from sampling theory and synchronous computing.
  - Set the cycle rate based on input data streams.
  - Leads to synthesizable Verilog subset and synchronous dataflow graphs.
- Must be a net-list language so that we can map a program to one or more processors or to hardware.

1/28/2006

Copyright E-TrolZ, Inc.

4 of 20

## Take Out the Garbage

- All instantiation at initialization.
- After instantiation
  - Software: the program runs in a cyclical loop at a fixed rate (or set of rates) on one or more processors.
  - Hardware: the design is mapped to RTL(Verilog).
- The design is statically analyzable before being run.
- Analyzable side effects.

1/28/2006

Copyright E-TrolZ, Inc.

5 of 20

## Why Not Just Haskell?

- Layout syntax is not industrial strength.
- No clear treatment of memory and state.
- Input/Output?
- Object oriented design?
  - Inheritance.
  - Types and Type Classes.

1/28/2006

Copyright E-TrolZ, Inc.

6 of 20

## Introduce a Special Memory Function

- A “register” sources and/or sinks data each cycle.
- Single assignment rules:
  - Write once in a cycle. New value updated at the end of cycle.
  - Multiple reads during a cycle(old value only).
- Synchronous, unblocked reads and writes.
- Registers are instantiated only at initialization.
- Input and Output are memory mapped to Registers.

1/28/2006

Copyright E-TrolZ, Inc.

7 of 20

## Execution Cycle

- Fetch data from each input and from registers and place it at the input to all functions.
- Execute all functions.
- Not lazy evaluation.
- Store the produced values away in the appropriate registers.
- Conventional “drivers” must then fill and empty these registers outside the functional program.

1/28/2006

Copyright E-TrolZ, Inc.

8 of 20



### What if the Input Data rates vary widely?

- No interrupts.
- Multiple “Tasks” each cycle at a different rate.
- Inter-task communication by rate adapting filters.
- System model is locally synchronous and globally asynchronous.

1/28/2006

Copyright E-TrolZ, Inc.

9 of 20

### System Modeling with Finite State Machines

- Each Task can be represented as a Finite State Machine.
- The State is contained in the registers of each task.
- State changes only at an execution cycle boundary.
- Leads to a design model of concurrent FSM's.

1/28/2006

Copyright E-TrolZ, Inc.

10 of 20

### Software Implementation on a Stack

- The compiler converts C-like expressions to postfix.
- Execute the postfix directly on a simple stack machine.
- Is this Forth? Almost!
  - No user written postfix.
  - Strongly typed (like Haskell).
- Stack machines are small, efficient and a target for soft cores in an FPGA.

1/28/2006

Copyright E-TrolZ, Inc.

11 of 20

### Hardware Implementation

- The front end of the compiler is the same!
- Only difference is that for hardware, the “interpreter” unwinds the postfix code into structural Verilog.
- Can we do without synthesis?
- For parallel processing, a complex mapping problem remains.

1/28/2006

Copyright E-TrolZ, Inc.

12 of 20

## Types and Type Classes

- Type Classes:
  - Members can be sub TypeClasses or Types
  - Abstract, not instantiated.
  - Lists methods that must be supplied by its Types.
- Types:
  - Are Instantiated.
  - Supply constructor and methods required by its Type Class.
- Sub-types differentiate with relations on property values of types.

1/28/2006

Copyright E-TrolZ, Inc.

13 of 20

## Polymorphism and Inheritance

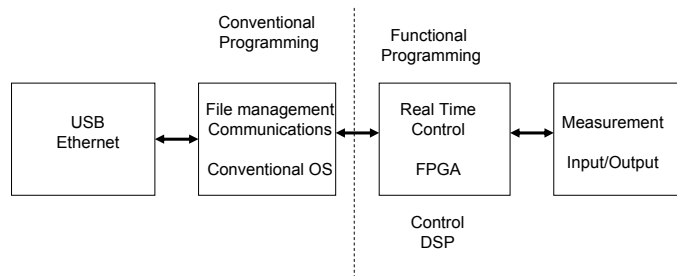
- Simple tree like single inheritance.
- Parametric polymorphism.
- No class casts.
- Simplified form of Object Oriented Design.

1/28/2006

Copyright E-TrolZ, Inc.

14 of 20

## Partitioning an Embedded Architecture



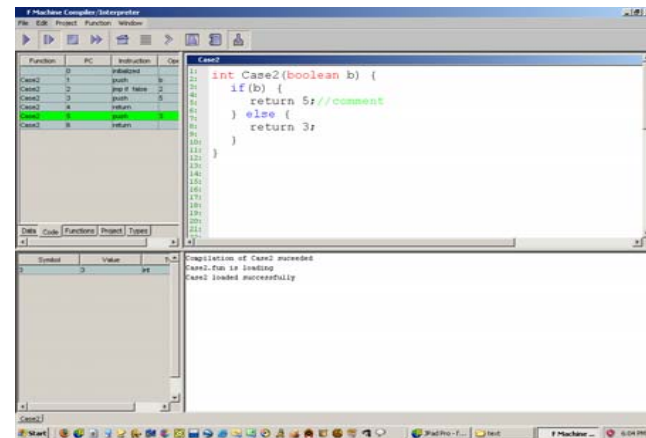
Separate Soft from Hard Real Time Measurement and Control

This Slide Courtesy of E-TrolZ, Inc., [www.e-trolz.com](http://www.e-trolz.com).

1/28/2006

Copyright E-TrolZ, Inc.

15 of 20



A Simple if – else branch executing. The return value is on the stack.

1/28/2006

Copyright E-TrolZ, Inc.

16 of 20

A "for" loop looks just like C.

1/28/2006 Copyright E-TrolZ, Inc. 17 of 20

A Recursive Factorial Function.

1/28/2006 Copyright E-TrolZ, Inc. 18 of 20

A Shape Type Class. Each Type provides its own Area implementation. Sub-Types provide relations on the properties of their parent Types.

1/28/2006 Copyright E-TrolZ, Inc. 19 of 20

### Summary

- Start with Functional programming (Haskell).
- Add a "memory" function; a non-blocking register.
- Instantiation followed by cyclical execution determined by the sampling rate.
- A type system whose instantiated objects have "state".
- Simplified object and inheritance model.
- C like syntax. But not C or Java compatible.
- Execute on simple stack machine(s) or translate into hardware.

1/28/2006 Copyright E-TrolZ, Inc. 20 of 20

# The Fresh Breeze Memory Hierarchy

Jack B. Dennis

MIT Computer Science and Artificial Intelligence Laboratory  
Cambridge, MA 02139

## Extended Abstract

The Fresh Breeze project concerns the design of a multi-core chip, guided by principles of modular software construction, that may be used to build scalable parallel computing systems that achieve high performance with low power consumption. The project addresses the challenge of making parallel computers more programmable. In particular the functional programming style is supported so that parallelism is readily exploited in programs written in an appealing and familiar form.

In the Fresh Breeze system architecture, there are three significant departures from conventional multi-processor architecture. One is the use of simultaneous multithreading processors. The benefit of this choice is greater latency tolerance for memory references, and better utilization of function units. Another feature is the use of a large, global, shared address space. This reduces the complexity of thread switching and communication, and eliminates the need for a distinction between memory and files. A more radical choice is the use of fixed-size chunks for memory allocation. Memory chunks are created and shared among computing activities, but are never updated. This choice avoids cache consistency problems, and allows reference count garbage collection by preventing creation of cycles of chunk references.

A Fresh Breeze chip has eight simultaneous multithreading processors (MTPs) together with blocks of on-chip memory that hold active code and data chunks. The memory blocks, known as Instruction Access Units (IAUs) and Data Access Units (DAUs) are accessible from all MTPs through a set of crossbar interconnects.

An code chunk can hold 32 32-bit instructions; a data chunk can hold up to 32 32-bit data words or up to 16 64-bit longs. Any of the 64-bit items in a chunk may be the UID of another chunk, hence the collection of all data chunks and their pointers to one another forms a directed graph called the heap. In view of the no-update rule, it is impossible to create cycles in the heap, and this fact facilitates continuous, parallel garbage collection using the reference count method.

Execution of a computation job on a Fresh Breeze system has the form of a tree of method (function) activations. Each method activation may have one or more active threads or activities, and has an associated code segment from which instructions are fetched, and a local data segment for variables local to the method. Both the code segment and the local data segment are implemented as a one-level tree of chunks, a master chunk containing pointers to up to sixteen code or data chunks. Pointers (UIDs) may be present as components of chunks, values in a code segment, or may be held in registers.

The top of the memory hierarchy of a Fresh Breeze system is the Register File of each processor, which holds 32 words or 16 longs for each of four activities. The second level consists of the IAUs and DAUs that hold code and data chunks for use by the MTPs on one Fresh Breeze chip. The third level is the Shared Memory System, not yet proposed in detail, that acts as a repository for code and data chunks.

The Register File of each MTP is organized to provide high performance at relatively low cost, in exchange for an occasional added cycle in instruction execution. It is divided into four independent sections, each having two banks that can operate concurrently. Each bank has three read ports and one write port. One read port is reserved to support write transfers from registers to the DAUs; the other two ports provide (for two banks and four sections) 16 simultaneous word accesses, enough to support two two-operand, double-precision operations on each cycle, plus extra bandwidth for integer and control operations. Each function unit has a result buffer that holds results until they can be written back to the Register File. While in the Result Buffer, values are available for use as bypass operands.

Instructions are provided to request multi-word transfers between the DAUs and the Register File to make best use of the crossbar connectivity.

The IAUs and DAUs serve as cache memories for code and data chunks. Analogous to the tag lines of cache memories, a Fresh Breeze chip has two content-addressable memories (I-CAM and D-CAM) that

implement directories of code and data chunks held in the IAUs and DAUs. When the on-chip location of a chunk is needed, an MTP requests translation of its UID from the I-CAM or D-CAM over an arbitrated bus. If there is no entry for the chunk, the complete chunk is retrieved from the Shared Memory System. During such retrievals, an on-chip thread scheduler keeps a record of suspended threads and reactivates them when retrieval is complete and processor resources are available. In the meantime the scheduler will run other threads awaiting service. In this way, demand “paging” of chunks is implemented. The LRU replacement method is implemented by arranging for each MTP to mark the chunk locations touched in program execution.

Because the shared bus would be overwhelmed if the I-CAM and D-CAM were consulted for every instruction fetch or data reference, short-cuts are provided so that only the first reference to a chunk by an MTP causes a CAM search. To this end each MTP has

two Map Tables, one for code segments and one for data segments. These are read on each code or local data reference to find the location if the chunk has already been mapped for an earlier access. To provide the same benefit for access to heap chunks, each word pair (that can hold a UID) of the Register File has an associated location tag that provides the corresponding chunk location if it is known.

## Conclusion

The Fresh Breeze project combines a lot of ideas, old and new, in a way that we believe addresses the challenges of today’s technological and usage environment. Many design choices have been made with little ability to anticipate their effects on performance and programmability, so we look forward with excitement and trepidation to the results of simulation trials and programming experiments and. It should be fun.

## References

- [1] Jack B. Dennis. Fresh Breeze: A multiprocessor chip architecture guided by modular programming principles. *ACM Sigarch News*, March 2003.
- [2] Jack B. Dennis. A parallel program execution model supporting modular software construction. In *Third Working Conference on Massively Parallel Programming Models*. IEEE Computer Society, 1998, pp 50-60.
- [3] Jack B. Dennis. General parallel computation can be performed with cycle-free heap. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compiling Techniques*. IEEE Computer Society, 1998, pp 96-103.

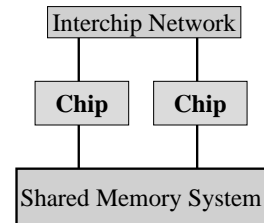
## The Fresh Breeze Memory Hierarchy

Jack Dennis  
MIT Computer Science  
and  
Artificial Intelligence Laboratory

3 February 2006

1 of 15

## A Fresh Breeze System



- A scalable multiprocessor system.
- Intended to support the functional programming style for implicit parallelism and programmability.
- Intended to achieve high performance with low power consumption.

3 February 2006

2 of 15

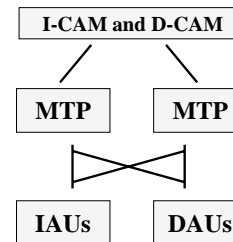
## Ideas

- **Simultaneous Multithreading:** Improves latency tolerance and function unit utilization.
- **Global Shared Memory:** Reduces complexity of thread switching and communication; eliminates need for distinction between memory and files.
- **Fixed-Size Chunks:** Simplifies memory management.
- **No Update:** Chunks are created, accessed, and released; no multiprocessor consistency problem.
- **Cycle-Free Heap:** Parallel reference count garbage collection of chunks may be used.

3 February 2006

3 of 15

## Fresh Breeze Multiprocessor Chip



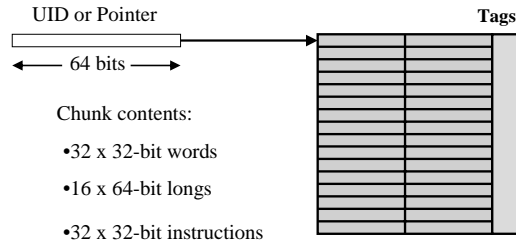
- Active chunks are held in on-chip IAUs and DAUs. Chunks are analogous to I-cache and D-cache data lines.
- MTPs: Multithreaded Processors.
- I-CAM and D-CAM are associative directories of chunks held in IAUs and DAUs; analogous to cache address lines and TLB, but shared by all MTPs.

3 February 2006

4 of 15

## Memory Chunks and UIDs

- **Chunk:** A fixed-size unit of memory allocation. 1024 bits of data;

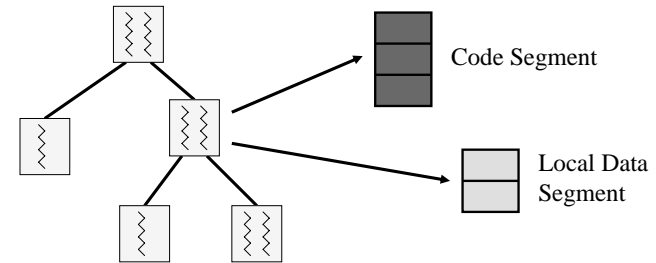


3 February 2006

5 of 15

## Program Execution

- A program in execution is a tree of method activations (similar to Monsoon).



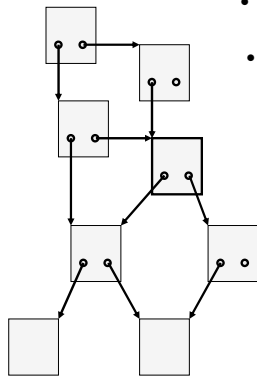
Several threads may be active in each method activation.

3 February 2006

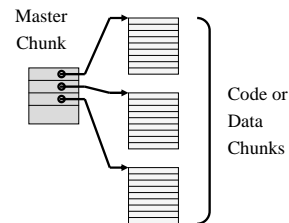
6 of 15

## Data Structures

- Cycle-Free Heap



- Fan-out as large as 16
- **Code Segment or Local Data Segment**



- Arrays: Three levels yields 4096 elements (longs)

3 February 2006

7 of 15

## Memory Hierarchy

- **Register File:**
  - 32 x 32-bit words per Activity
- **Code and Data Access Units:**
  - 1024-bit Code and Data Chunks shared by all MTPs on chip. About  $2^{14}$  chunks. Like cache lines.
- **Shared Memory System:**
  - Repository for code and data chunks. Up to  $2^{64}$  chunks.

3 February 2006

8 of 15

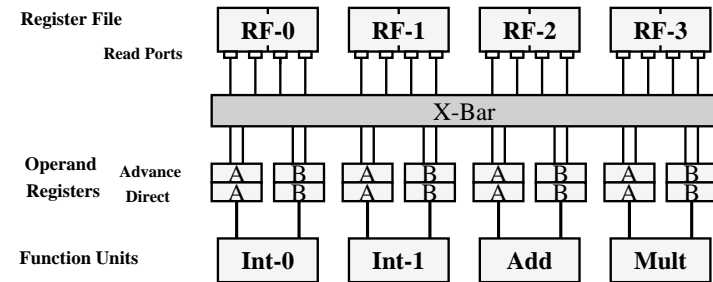
## Register File

- 32 x 32-bit words per Activity
- Words are paired for longs or UIDs
- Four sections, each having two banks for left and right (even/odd) words.
- Total of 16 words per bank (four per activity).
- Two read ports, one write port per bank.
- Total bandwidth is  $4 \times 2 \times 2 = 16$  reads (32-bit) per cycle; 8 (32-bit) writes per cycle.

3 February 2006

9 of 15

## Superscalar Operation I Operand Access

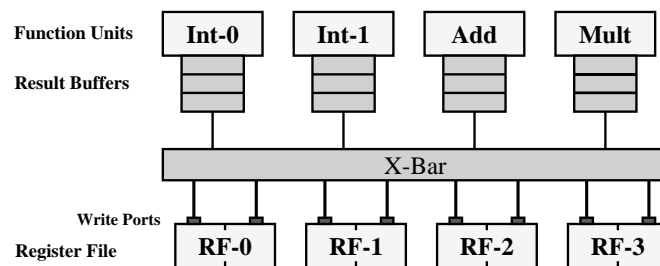


- Port 0 supplies Operand A; Port 1 supplies Operand B.
- Port 3 is used to read registers for store transfers.

3 February 2006

10 of 15

## Superscalar Operation II Result Writeback



- Bypass Operands may be selected from the result buffers.
- Load transfers share the write ports of all Register File sections and have priority over writebacks

3 February 2006

11 of 15

## Data Movement

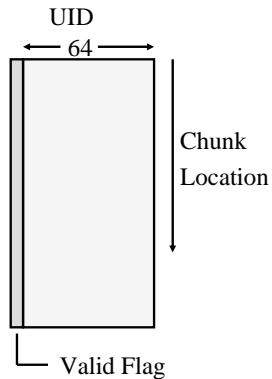
- Between Registers and Access Units
  - Programmed as multi-word loads and stores
  - X-Bar switch permits one transfer per MTP simultaneously, if no conflicts
- Between Access Units and the Shared Memory System
  - Complete Chunk is the unit of transfer.
  - Demand “Paging” using built-in LRU replacement of chunks in Access Units.
  - Supported by on-chip Activity (Thread) Scheduler.

3 February 2006

12 of 15



## Chunk Directories I-CAM and D-CAM

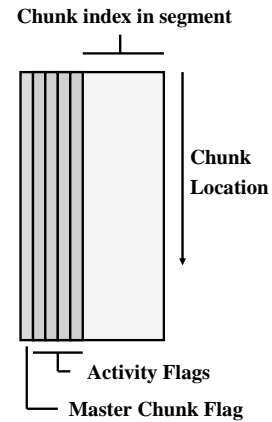


3 February 2006

13 of 15

- Content-Addressed memory;  $2^{14}$  entries.
- Shared by all MTPs on Chip using an arbitrated bus.
- Short-cuts needed to minimize number of accesses and decrease average latency of **access**.

## Mapping Short-Cuts



3 February 2006

14 of 15

- A Mapping table is maintained separately by each MTP for the chunks of the code segment and local data segment of each Activity.
- The Map is consulted before requesting service from the I-CAM or D-CAM.
- Each UID entry in the Register File has an associated location auxiliary field.
- After the first reference using a UID from a register, subsequent references are made without consulting the D-CAM.

## Conclusion

- The Fresh Breeze project combines of a lot of ideas, old and new.
- We believe it addresses the challenges of today's technological and usage environment.
- Many design choices have been made with little ability to anticipate their effects on performance and programmability.
- We look forward with excitement and trepidation to the results of simulation trials programming experiments.

**It should be fun!**

3 February 2006

15 of 15

# Ideal and Resistive Nanowire Decoders

## General models for nanowire addressing

Eric Rachlin and John E. Savage  
Department of Computer Science  
Brown University

Recent research in nanoscale computing offers multiple techniques for producing large numbers of parallel nanowires (NWs). These wires can be assembled into crossbars, two orthogonal sets of parallel NWs separated by a layer of molecular devices. In a crossbar, pairs of orthogonal NWs provides control over the molecules at their crosspoints. Hysteretic molecules act as programmable diodes, allowing crossbars to function as both memories and circuits (a PLA for example). Either application requires that NWs be interfaced with existing CMOS technology.

The technology for controlling a large number of NWs with a much smaller number of lithographically produced mesoscale wires (MWs) is called a decoder. A number of methods for producing decoders have been proposed and studied separately. These decoders can all be modeled as embedding resistive switches in NWs, where each switch is controlled by a MW. In this unifying approach, the sequence of switches embedded in a particular NW is termed its “codeword”.

All proposed techniques for decoder production involve a significant degree of uncertainty. Nanoscale features cannot be placed precisely. As a result, codewords are assigned randomly to NWs. We believe that stochastic assembly will remain a defining characteristic of nanoscale computing technology. NW decoders provide a highly practical starting point for the more general study of stochastically assembled devices.

We begin this talk by briefly reviewing existing NW and decoder technologies, then present our general model for NW decoders. We define a “simple NW decoder”, which uses a pair of ohmic contacts and a set of MWs to control a set of parallel NWs. We also define a “composite NW decoder”, which combines multiple simple NW decoders to control a large number of NWs efficiently.

We pay particular attention to how these decoders are used in the context of a crossbar.

To understand what qualifies as a properly functioning decoder, we must first describe how MWs control NWs. We provide two models of MW control. In the ideal model, each MW completely turns off some subset of NWs. In the more general resistive model, each MW merely increases each NW’s resistance by some positive amount. The ideal model uses binary codewords, while the resistive model uses real-valued codewords.

Binary codewords are a convenient way of describing decoders. Using binary codewords, one can concisely state the conditions a decoder must satisfy. Real-valued codewords, by contrast, are cumbersome to work with. We describe how real-valued codewords can be mapped to binary codewords by adding the notion of “errors” to our ideal model.

In our discussion of errors, we explain how decoders can be made robust. We define the notion of “balanced hamming distance”, which accurately captures the criteria a decoder must meet to tolerate permanent defects and certain transient errors. Using this concept, we can bound the number of MWs required to make different types of fault-tolerant decoders. This in turn bounds the size of these decoders.

Given more time, we would also address the problem of codeword discovery. Since decoders are assembled stochastically, testing is required to discover which codewords are present. This information must be stored to produce a properly functioning decoder. We consider efficient codeword discovery algorithms which do not require nanoscale measurements or specialized testing circuitry. In contrast, our previous discovery algorithms required the use of hysteretic molecules on which read/write operations could be performed.

# Ideal and Resistive Nanowire Decoders

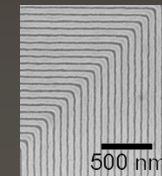
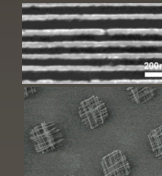
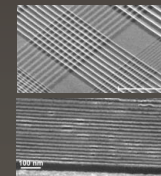
General Models for Nanowire Addressing

Eric Rachlin and John E. Savage  
Brown University CS Department  
February 03, 2006

## The Nanowire

- Sets of parallel NWs have been produced.
- Devices will reside at NW intersections.
- We must gain control over individual NWs.

SNAP NWs (Heath, Caltech)    CVD NWs (Lieber, Harvard)    Directed Growth (Stoykovich, UW)



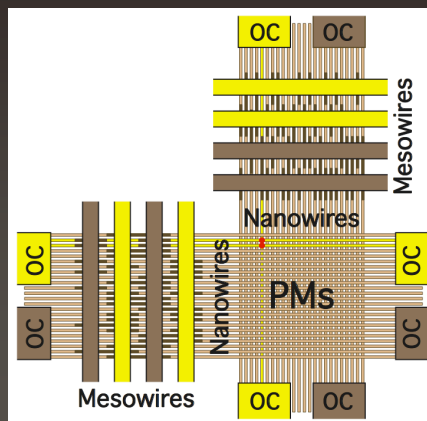
2 of 16

## The Crossbar

The crossbar is currently the most feasible nano-scale architecture.

By addressing individual NWs, we can control programmable molecules at NW crosspoints.

Crossbars are a basis for memories and circuits.



3 of 16

## Nanowire Control

- Mesoscale contacts apply a potential along the lengths of NWs.
- Mesoscale wires (MWs) apply fields to across NWs, some of which form FETs.
- NW/MW junctions can form FETs using a variety of technologies:
  - ⇒ Modulation-doping
  - ⇒ Random Particle deposition
  - ⇒ Masking NWs with dielectric material

4 of 16

## Simple NW Decoders

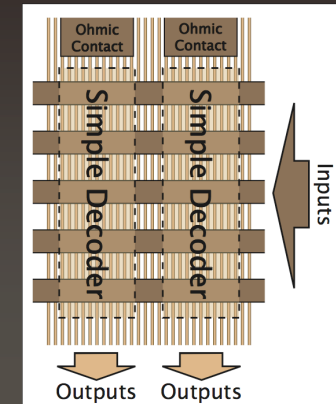
- A potential is applied along the NWs.
- $M$  MW inputs control  $N$  NW outputs. Each MW controls a subset of NWs.
- When a MW produces a field, the current in each NW it controls is greatly reduced.
- Each MW “subtracts” out subsets of NWs. This permits  $M \ll N$ .
- Decoders are assembled stochastically and are difficult to produce if  $N$  large.



5 of 16

## Composite Decoders

- A composite decoder uses multiple simple decoders to control many NWs.
- The simple decoders share MW inputs.
- This space savings allows for mesoscale inputs.



6 of 16

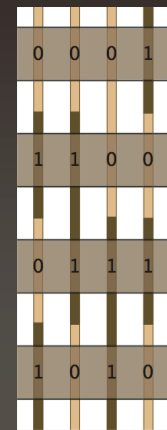
## Ideal Decoders

- To analyze a decoder, we must model how MWs control NWs.
- In an **ideal decoder**, a MW's electric field completely turns off the NWs it controls. Other NWs are unaffected.
- This model is accurate if the FETs formed from MW/NW junctions have high on/off ratios.

7 of 16

## Binary Codewords

- In an ideal decoder, we associate an  $M$ -bit codeword,  $c_i$ , with each NW,  $n_i$ .
- The  $j^{\text{th}}$  MW controls the  $i^{\text{th}}$  NW if and only if the  $j^{\text{th}}$  bit of  $c_i$ ,  $c_{ij}$ , is 1.
- The  $M$ -bit decoder input,  $A$ , causes  $n_i$  to carry a current if and only if  $A \cdot c_i = 0$ .
- Codeword assignment is stochastic.
- Control over codewords is a key way to compare decoding technologies.



8 of 16

# Codeword Interaction

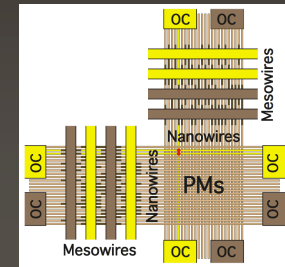
- If  $c_{bj} = 1$  where  $c_{aj} = 1$ ,  $c_a$  **implies**  $c_b$ . Inputs that turn on  $n_a$  turn off  $n_b$ .
- A set of codewords,  $S$ , is **addressable** if some input turns off all NWs not in  $S$ .
- $S = \{c_i\}$  is addressable if and only if no codeword implies  $c_i$ .  $S$  is addressed with input  $A = c_i$ .



9 of 16

# Decoders for Memories

- A  $B$ -bit memory maps  $B$  addresses to  $B$  disjoint sets of storage devices.
- A  $D$ -address memory decoder addresses  $D$  disjoint subsets of NWs.
- Equivalently, the decoder contains  $D$  addressable codewords.



10 of 16

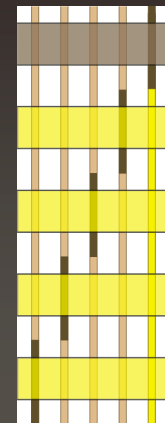
# Resistive Decoders

- Decoders that rely on FETs are not ideal.
- MWs carrying a field increase each NW's resistance by some amount.
- In a **resistive decoder**, codewords are real-valued. In real-valued codeword  $r_i$ ,  $r_{ij}$  is the resistance induced in  $n_i$  by the  $j^{\text{th}}$  MW.
- On input  $A$ ,  $n_i$ 's resistance is  $r_{base} + A \cdot r_i$ .

11 of 16

# Ideal vs. Resistive

- In a resistive memory decoder the addressed NWs must output more current than the other NWs.
- Consider 1-hot codewords:
  - $\Rightarrow$  The addressed wire has resistance  $< r_{base} + M r_{low}$
  - $\Rightarrow$  Remaining wires have resistance  $> (r_{base} + r_{high})/N$
- We require that  $r_{high} \gg M N r_{low}$  and  $N r_{base}$ 
  - If  $r_{ij} \leq r_{low}$ ,  $c_{ij} = 0$ .
  - If  $r_{ij} \geq r_{high}$ ,  $c_{ij} = 1$ .
  - If  $r_{low} < r_{ij} < r_{high}$ ,  $c_{ij}$  is an error.



12 of 16

## Ideal Decoders with Errors

- To apply the ideal model to resistive decoders, consider binary codewords with random **errors**.
- If  $c_{ij} = e$ , the  $j^{\text{th}}$  MW increases  $n_i$ 's resistance by an unknown amount.
- Consider input  $A$  such that the  $j^{\text{th}}$  MW carries a field.  $A$  functions reliably if a MW for which  $c_{jk} = 1$  carries a field.

0	0	0	1
1	1	0	0
0	1	e	1
1	0	1	0

13 of 16

## Balanced Hamming Distance

1	0	1	0	1	0	1	0
0	1	1	0	0	1	1	0
0	0	0	0	0	0	0	0

- Consider two error-free codewords,  $c_a$  and  $c_b$ . Let  $|c_a - c_b|$  denote the number of inputs for which  $c_{aj} = 1$  and  $c_{bj} = 0$ .
- The balanced Hamming distance (BHD) between  $c_a$  and  $c_b$  is  $2 \cdot \min(|c_a - c_b|, |c_b - c_a|)$ .
- If  $c_a$  and  $c_b$  have a BHD of  $2d + 2$  they can collectively tolerate up to  $d$  errors.

14 of 16

## Fault-Tolerant Random Particle Decoders

- In a particle deposition decoder,  $c_{ij} = 1$  with some fixed probability,  $p$ .
- If each pair of codeword has a BHD of at least  $2d + 2$ , the decoder can tolerate  $d$  errors per pair.
- This holds with probability  $> 1 - f$  when

$$M > \frac{(d + (d^2 + 4 \ln(N^2/f))^{1/2})^2}{4p(1 - p)}$$

15 of 16

## Conclusion

- Any nanoscale architecture will require control over individual NWs.
- Stochastically assembled decoders can provide reliable control even if errors occur.
- Our decoder model applies to many viable technologies and provides conditions that decoders must meet.

16 of 16