Workshop on Complexity−Effective Design

Held in conjunction with the 32nd International Symposium on Computer Architecture

Madison, Wisconsin

June 5, 2005

Workshop Organizers:

Dave Albonesi, Cornell University

Pradip Bose, IBM Corporation

Prabhakar Kudva, IBM Corporation

Diana Marculescu, Carnegie−Mellon University

WCED Program

8:30-9:30AM Session I

- Effects of Pipeline Complexity on SMT/CMP Power-Performance Efficiency
  B. Lee and D. Brooks (Harvard University)

- Early Performance Prediction
  P. Kudva, B. Curran, S.K. Karandikar, M. Mayo, S. Carey (IBM), and S.S. Sapatnekar (University of Minnesota)

- Wire Management for Coherence Traffic in Chip Multiprocessors
  L. Cheng, N, Muralimanohar, K. Ramani, R. Balasubramonian, and J. Carter (University of Utah)

9:30-9:50AM Break
9:50-10:30AM Session II

- Reducing the Power and Complexity of Path-Based Neural Branch Prediction
  G.H. Loh (Georgia Institute of Technology) and D.J. Jiménez (Rutgers University)

- A Break-Even Formulation For Evaluating Branch Predictor Energy Efficiency
  M. Co, D.A.B Weikle, and K. Skadron (University of Virginia)

10:30-11:00AM Break
11:00-11:40AM Session III

- Heuristics for Complexity-Effective Verification of a Cache Coherence Protocol Implementation
  D. Abts (Cray), Y. Chen, and D.J. Lilja (University of Minnesota)

- The Design Complexity of Program Undo Support in a General-Purpose Processor
  R. Teodorescu and J. Torrellas (University of Illinois at Urbana-Champaign)

11:40AM-12:30PM Metrics discussion led by J. Torrellas (University of Illinois at Urbana-Champaign)

Introduction

The quest for higher performance via deep pipelining (for high clock rate) and speculative, out-of-order execution (for high IPC) has yielded processors with greater performance, but at the expense of much greater design complexity. The costs of higher complexity are many-fold, including increased verification time, higher power dissipation, and reduced scalability with process shrinks/variations. The Workshop on Complexity-Effective Design (WCED) was founded with the intention of bringing together microarchitects, circuit designers, performance modelers, compiler developers, verification experts, and system designers to discuss and explore hardware/software techniques and tools for creating future designs that are more complexity-effective.

A complexity-effective design feature or tool either (a) yields a significant performance and/or power efficiency improvement relative to the increase in hardware/software complexity incurred; or (b) significantly reduces complexity (design time and/or verification time and/or improved scalability) with a tolerable performance/power impact. The papers in this year's WCED program address both of these themes.

We wish to thank Mikko Lipasti, the Workshops Chair, and the other ISCA organizers that allowed us to offer the workshop, the WCED Program Committee (Todd Austin, R. Iris Bahar, David Brooks, Alper Buyuktosunoglu, George Cai, Babak Falsafi, Keith Farkas, Antonio Gonzalez, Peter Hofstee, Gokhan Memik, Chuck Moore, and Subbarao Palacharla), all those who reviewed papers, and all workshop authors and presenters. We welcome any and all feedback that will help us improve WCED in future years.

Dave Albonesi
Pradip Bose
Prabhakar Kudva
Diana Marculescu

WCED Co-Chairs

Effects of Pipeline Complexity on SMT/CMP Power-Performance Efficiency

Benjamin Lee and David Brooks
Harvard University
Division of Engineering and Applied Sciences
Cambridge, Massachusetts, USA
{bclee, dbrooks}@eecs.harvard.edu

Abstract

We consider processor core complexity and its implications for the power-performance efficiency of SMT and CMP architectures, exploring fundamental trade-offs between the efficiency of multi-core architectures and the complexity of their cores from a power-performance perspective. Taking pipeline depth and width as proxies for core complexity, we conduct power-performance simulations of several SMT and CMP architectures employing cores of varying complexity. Our analyses identify efficient pipeline dimensions and outline the implications of using a power-performance efficiency metric for core complexity.

Collectively, our results suggest SMT architectures enable efficient increases in pipeline dimensions and core complexity. Furthermore, reducing pipeline dimensions in CMP cores is inefficient, assuming ideal power-performance scaling from voltage/frequency scaling and circuit re-tuning. Given these conclusions, we formulate guidelines for complexity effective design.

1. Introduction

We present an analysis of processor core complexity, quantified by pipeline depth and width, and its effects on the power-performance efficiency of simultaneous multi-threading (SMT) and chip multi-processing (CMP). We define efficiency in terms of $BIPS^3/W$, a voltage invariant power-performance metric that captures the cubic relationship between power and performance. Research in efficient computer architectures has been motivated by significant increases in power dissipation on high-performance systems. Increasing power dissipation also complicates thermal management and current/voltage stability in a system. SMT and CMP architectures are of particular interest as the microprocessor industry moves towards such systems to meet performance targets in mainstream computing [1, 2, 3].

SMT architectures amortize the cost of microarchitectural structures over a greater number of instructions per cycle drawn from multiple threads. Similarly, CMP architectures increase thread-level parallelism by constructing multiple processor cores on a single die. Prior work has examined the power-performance efficiency of such architectures for a particular core design [4, 5] or a particular class of applications [6]. In contrast, we examine SMT/CMP efficiency as a function of core complexity by taking pipeline depth and width as a proxy for complexity.

We specify pipeline depth by the number of FO4 inverter delays per pipeline stage[1] and pipeline width by the maximum number of instructions decoded per cycle. Identifying optimal pipeline parameters for SMT and CMP cores effectively optimizes the balance between instruction-level and thread-level parallelism to maximize efficiency.

Conducting simulations of several SMT and CMP system configurations with varying pipeline parameters (Section 2), we identify power-performance efficient pipeline dimensions and outline the implications of using an efficiency metric for core complexity (Section 3, Section 4). Overall, we draw the following conclusions:

1. SMT architectures enable power-performance efficient increases in pipeline dimensions toward deeper, wider pipelines (Section 3.3, Section 4.3).
2. Reducing pipeline dimensions in CMP architectures is power-performance inefficient (Section 3.4, Section 4.4) relative to alternatives from hardware tuning.

Collectively, these results support the conventional wisdom that SMT architectures are more effective with deeper, wider pipelines and refute the belief that CMP core complexity should decrease relative to uni-processor complexity. We employ these conclusions to formulate guidelines for complexity effective SMT and CMP design (Section 5).

1 Fan-out-of-four (FO4) delay is defined as the delay of one inverter driving four copies of an equally sized inverter. When logic and latch overhead per pipeline stage is measured in terms of FO4 delay, deeper pipelines have smaller FO4 delays.

2. Experimental Methodology

2.1. Performance Modeling

We employ Turandot, a cycle-based microprocessor performance simulator [7, 8], to obtain data for varying pipeline designs in out-of-order superscalar processors. We evaluate the performance of a design in terms of its achieved instruction throughput measured in billions of instructions per second (BIPS). We compute throughput for n threads with Equation (1), where $Inst_i$ is the number of instructions committed by thread i, $max(Cy_i)$ is the number of cycles required to complete the execution of all threads, and f is the clock frequency. We also evaluate performance in terms of effective delay or inverse throughput ($BIPS^{-1}$).

$$BIPS = \frac{\sum_{i=1}^{n} Inst_i}{max(Cy_i)} \times \frac{f}{10^6} \quad (1)$$

2.2. Power Modeling

We employ PowerTimer, a Turandot based microarchitectural simulator with power modeling extensions, to examine the power implications of varying pipeline designs [9, 10]. The PowerTimer energy models are based on circuit-level power analyses performed on microarchitectural structures in a modern, high-performance PowerPC [11]. Each structure or subunit is comprised of multiple individually analyzed macros. The circuit-level power analyses determine each structure's power dissipation as the sum of hold power and switching power where switching power is a function of the structure's input switching factor. The unconstrained hold and switching power for each macro are combined to generate linear equations for each macro's power. A linear combination of power equations for the macros within a particular subunit produces the subunit's unconstrained power model (Figure 1). PowerTimer uses microarchitectural activity information from Turandot to scale down each subunit's unconstrained hold and switching power under a variety of clock gating assumptions on a per-cycle basis to get the final estimate of power.

Power dissipated by a processor consists of dynamic and leakage components, $P = P_{dynamic} + P_{leakage}$. The dynamic component may be expressed as

$$P_{dynamic} = CV^2 f(\alpha + \beta) \times CGF \quad (2)$$

where α is the true switching factor required for logic functionality, β is the glitching factor that accounts for spurious transitions resulting from differing delays between logic paths. Thus, $(\alpha + \beta)$ is the average number of actual transitions. The clock gating factor, CGF, is the fraction of cycles during which microarchitectural structures are not clock gated. The remaining variables are the effective switching capacitance (C), the supply voltage (V), and the clock frequency (f).

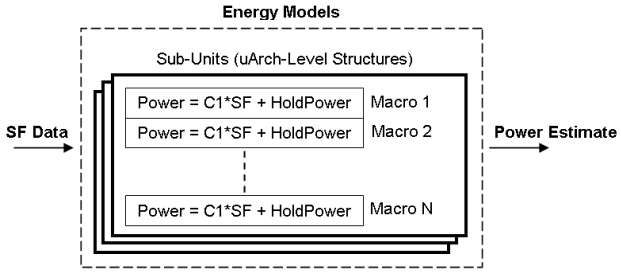


Figure 1. PowerTimer Energy Models.

2.3. Power-Performance Efficiency

We use $\frac{BIPS^3}{W}$ as our power-performance efficiency metric for comparing different pipeline designs. This metric is derived from the cubic relationship between power dissipation and the supply voltage, given a fixed logic/circuit design. Since $P_{dynamic} \approx CV^2 f\alpha \times CGF$ and V is roughly proportional to f, $P_{dynamic} \approx CV^3\alpha \times CGF$. Thus, power is roughly proportional to V^3. This suggests $P \times D^3$, where P is power and D is delay, is an appropriate voltage invariant power-performance metric for server-class microprocessors [9]. Lastly, note that $\frac{BIPS^3}{W} \equiv (P \times D^3)^{-1}$.

2.4. Microarchitectural Modeling

The baseline configuration is a single-threaded pipeline with 19 FO4 delays per stage capable of decoding four non-branch instructions per cycle (Table 1). We also analyze several SMT and CMP architectures (Table 2) with extended versions of Turandot and PowerTimer[5]. The SMT cores are modeled by increasing the sizes of shared and critical resources (*e.g.* register file and issue queues) by 50 percent and implementing round-robin thread scheduling. CMP architectures are simulated as separate cores, tracking conflicts in the shared L2 cache and cache bus. Inter-thread synchronization is not currently supported.

We refer to a particular design point as [*arch*,*depth*,*width*], where *arch* is an architecture from Table 2 and *depth*, *width* refer to the pipeline dimensions. Depth is quantified in terms of FO4 delays per pipeline stage and width is quantified in terms of the number of non-branch instructions decoded per cycle. For example, the baseline is denoted as [ST,19,4].

Processor Core	
Decode Rate	4 non-branch instructions per cycle
Dispatch Rate	9 instructions per cycle
Reservation Stations	FXU(40),FPU(10),LSU(36),BR(12)
Functional Units	2 FXU, 2 FPU, 2 LSU, 2 BR
Physical Registers	80 GPR, 72 FPR
Branch Predictor	16k 1-bit entry BHT
Memory Hierarchy	
L1 DCache Size	32KB, 2-way, 128B blocks, 1-cy latency
L1 ICache Size	32KB, 1-way, 128B blocks, 1-cy latency
L2 Cache Size	2MB, 4-way, 128B blocks, 9-cy latency
Memory	77-cy latency
Pipeline Dimensions	
Pipeline Depth	19 FO4 delays per stage
Pipeline Width	4-decode

Table 1. Baseline Microarchitectural Parameters.

Architecture	Description
ST	Single-threaded baseline (Table 1)
SMT1c1t	SMT-expanded core running 1 thread
SMT1c2t	SMT-expanded core running 2 threads
CMP2c1t	CMP with 2 SMT-expanded cores, each running 1 thread
CMP2c2t	CMP with 2 SMT-expanded cores, each running 2 threads

Table 2. SMT,CMP Architectures.

2.5. Benchmarks

We report experimental results from a suite of 21 traces from the SPEC2000 benchmarks in Table 3. The traces were generated with the tracing facility *Aria* [8] using the full reference input set. However, sampling reduced the total trace length to 100 million instructions per benchmark program. The sampled traces were validated against the full traces before finalizing the traces [12].

We present power and performance data as an average of the SPEC2000 benchmarks. The single-threaded core configurations (*i.e.*, ST and SMT1c1t) were run with every benchmark in the suite. The multi-threaded core configurations (*i.e.*, SMT1c2t and CMP*c*t) were run with multiple copies of the same benchmark for every benchmark in the suite. For example, we simulated CMP2c2t running four copies of *ammp*. Identifying an interesting subset of heterogeneous benchmarks for parallel execution is future work.

3. Pipeline Depth Analysis

3.1. Depth: Performance Scaling

Models for architectures with varying pipeline depths are derived from the reference 19 FO4 design by treating the total number of logic levels as constant independent of the number of pipeline stages. This is an abstraction for the purpose of our analyses; increasing the pipeline depth could require logic design changes. The baseline latencies (Table 4) are scaled to account for pipeline changes according to

$$Latency_{target} = \left\lfloor Latency_{base} \times \frac{FO4_{base}}{FO4_{target}} + 0.5 \right\rfloor \quad (3)$$

SPEC2000 Benchmarks			
ammp	applu	apsi	art
bzip2	crafty	equake	facerec
gap	gcc	gzip	lucas
mcf	mesa	mgrid	perl
sixtrack	swim	twolf	vpr
wupwise			

Table 3. SPEC2000 Benchmark Suite.

Fetch		Decode	
NFA Predictor	1	Multiple Decode	2
L2 I-Cache	11	Millicode Decode	2
L3 I-Load	8	Expand String	2
I-TLB Miss	10	Mispredict Cycles	3
L2 I-TLB Miss	50	Register Read	1
Execution		**Memory**	
Fix Execute	1	L1 D-Load	3
Float Execute	4	L2 D-Load	9
Branch Execute	1	L3 D-Load	77
Float Divide	12	Float Load	2
Integer Multiply	7	D-TLB Miss	7
Integer Divide	35	L2 D-TLB Miss	50
Retire Delay	2	StoreQ Forward	4

Table 4. [ST,19,4] Latencies (cy).

All latencies have a minimum of one cycle. This is consistent with prior work in pipeline depth simulation and analysis for a single-threaded core [14].

3.2. Depth: Power Scaling

Each factor in Equation (2) scales with pipeline depth. The clock frequency f increases linearly with depth as the delay for each pipeline stage decreases. The clock gating factor CGF decreases by a workload dependent factor as pipeline depth increases due to the increased number of cycles in which the shorter pipeline stages are stalled. As the true switching factor α is independent of the pipeline depth and the glitching factor β decreases with pipeline depth due to shorter distances between latches, switching power dissipation decreases with pipeline depth. The latch count, and consequently hold power dissipation, increases linearly with pipeline depth. We take leakage power as 30 percent of dynamic power dissipation. We refer the reader to prior work for a detailed treatment of these scaling models.

3.3. Depth: Power-Performance Evaluation

Figure 2 presents the power and performance trends as the pipeline depth increases for [SMT1c1t,*,4] as an average of the 21 benchmarks in our suite (Table 3).[2] Note that pipeline depth is quantified in FO4 delays per stage and smaller FO4 delays are equivalent to deeper pipelines.

The performance maximizing pipeline depth is the 10 FO4 design point, where performance is measured in BIPS.

2 [ST,*,4] and [SMT1c1t,*,4] have similar trends despite differences in resource sizing. The CMP architectures, [CMP2c*t,*,4], effectively double the power and throughput of their SMT counterparts and are not evident in relative trends.

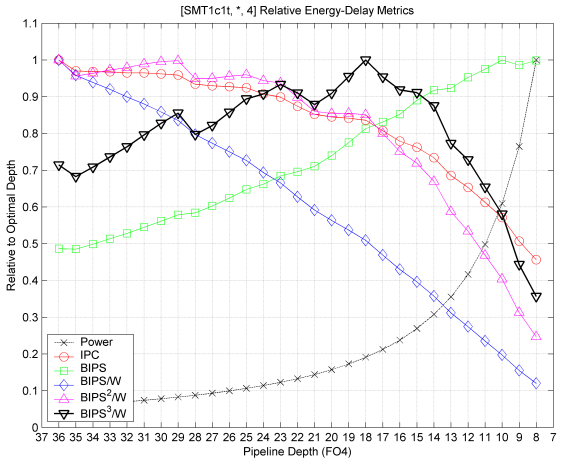


Figure 2. [SMT1c1t,*,4] Power-Performance Metrics.

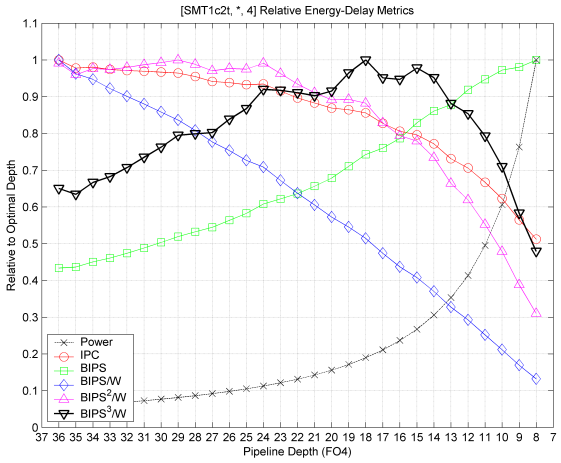


Figure 3. [SMT1c2t,*,4] Power-Performance Metrics.

Since power increases superlinearly and performance increases sublinearly with pipeline depth, the performance to power ratio $\frac{BIPS}{W}$ decreases with depth. In contrast, the 18 FO4 design point is power-performance efficient, maximizing the $\frac{BIPS^3}{W}$ metric. Note $\frac{BIPS^3}{W}$ decreases significantly as the number of pipeline stages increases beyond this design point. In particular, the performance maximizing 10 FO4 design point is 40 percent less efficient than the power-performance optimal 18 FO4 design point.

Figure 3 performs a similar power-performance analysis for the same architecture simultaneously executing two threads, [SMT1c2t,*,4]. Relative to the [SMT1c1t,*,4] results from Figure 2, we find executing a second thread mitigates the efficiency penalties associated with deeper pipelines. Although SMT1c1t and SMT1c2t share the same optimal pipeline depth of 18 FO4 delays per stage, deeper SMT1c2t pipelines up to the 14 FO4 design point achieve $\frac{BIPS^3}{W}$ within 5 percent of the optimal. In contrast, increasing SMT1c1t pipeline depth to 14 FO4 delays per stage reduces $\frac{BIPS^3}{W}$ by more than 10 percent. If performance were the only objective and the 8 FO4 design point were chosen, the 52 percent loss in efficiency for SMT1c2t is significantly smaller than the 64 percent loss for SMT1c1t.

Overall, these results are consistent with the conventional wisdom that SMT architectures perform better in deeper pipelines. The larger number of pipeline stages allows interleaving multiple simultaneous threads at a finer granularity than those afforded by shallower pipelines. Although SMT enables power-performance efficient increases in pipeline depth, voltage/frequency scaling and circuit retuning may provide many of the same benefits for significantly less engineering effort.

3.4. Depth: Complexity Implications

Figure 4 depicts *microarchitectural tuning curves* for each architecture[14]. The average effective delay (or inverse throughput[3]) of the traces in our benchmark suite is plotted on the x-axis in units relative to the baseline [ST,19,4]. The average power dissipation is similarly plotted on the y-axis. The scatter plots indicate the location of each design point in this power-performance space, capturing trends in power-performance trade-offs at each point for a fixed supply voltage. The deeper pipelines are positioned in the high-power, low-delay region of the tuning space.

The downward sloping plots are *hardware tuning curves* representing the power-performance trade-offs achievable from varying the supply voltage and tuning the circuits to meet the frequency target. These curves are an abstraction of the techniques employed and design decisions made at various stages in the microprocessor design (*e.g.* voltage/frequency scaling, transistor sizing, and circuit styles). As the delay budget tightens, greater gate-level parallelism and transistor sizes are needed and more power is dissipated. Hardware intensity, η, quantifies these power and performance trade-offs by specifying the percentage change in energy required to achieve a 1 percent improvement in the critical path delay by restructing the logic and re-tuning the circuits for a given power supply [13]. Mathematically, $\eta = -\frac{\%E}{\%D}$. Assuming a typical hardware intensity of two, these curves represent a 1 percent reduction in effective delay by changing the power supply and tuning the hardware, at a cost of 3 percent in power. This is consistent with the cubic relationship between power and performance.

The efficient 18 FO4 design point from Section 3.3 is also the point on the microarchitectural tuning curve where a 1 percent performance gain from increasing the pipeline

3 Zyuban et al. originally tracked delay on this axis, but inverse throughput ($BIPS^{-1}$) is a more accurate description of this metric when considering multiple thread contexts.

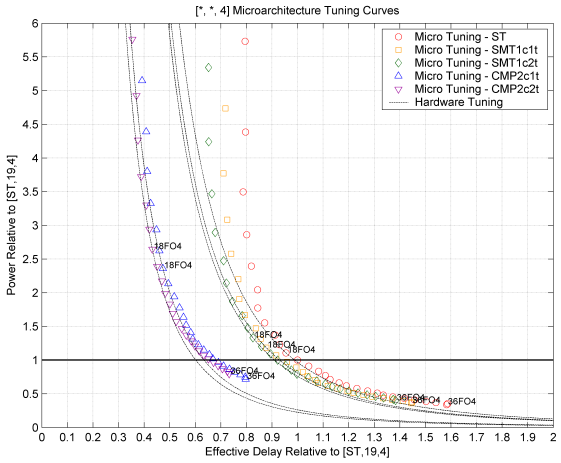


Figure 4. [*,*,4] Micro Tuning Curves.

depth requires a 3 percent increase in power dissipation. Graphically, this implies the point of tangency between the microarchitectural and hardware tuning curves is the efficient design point that delivers a given level of performance while minimizing power.

Let the power budget be the relative power dissipation of the original design [ST,19,4]. Alternative designs that meet this power budget include [SMT1c1t,20,4], [SMT1c2t,21,4], [CMP2c1t,30,4] and [CMP2c2t,31,4]. Note that each of these designs incorporate fewer pipeline stages while reducing effective delay. In particular, choosing [CMP2c2t,31,4] over [ST,19,4] reduces the number of pipeline stages by approximately two-thirds and decreases effective delay by approximately 35 percent.

[CMP2c2t,31,4] is inefficient because it is possible to achieve a greater reduction in delay for the same power dissipated, as illustrated in Figure 4. First, increase the pipeline depth to the 18 FO4 design point, the power-performance optimal for this architecture. This corresponds to moving up the CMP2c2t tuning curve from [CMP2c2t,31,4] to [CMP2c2t,18,4]. Second, tune the hardware until the power budget is met. This corresponds to moving down the hardware tuning curve until relative power dissipation is one. At this new, efficient design point, the power budget is met and effective delay is reduced by approximately 40 percent relative to [ST,19,4] and 8 percent relative to [CMP2c2t,31,4]. Thus, reducing core complexity in multi-core architectures are not power-performance efficient since hardware tuning more complex cores yields greater performance for the same power budget.

	8D	4D	2D	1D
Functional Units				
FXU	4	2	1	1
MEM	4	2	1	1
FXU	4	2	1	1
BR	4	2	1	1
CR	2	1	1	1
Pipeline Stage Widths				
FETCH	16	8	4	2
DECODE	8	4	2	1
RENAME	8	4	2	1
DISPATCH	8	4	2	1
RETIRE	8	4	2	1

Table 5. [ST,19,*] Width Resource Scaling.

Structure	Energy Growth Factor
Register Rename	1.1
Instruction Issue	1.9
Memory Unit	1.5
Multi-ported Register File	1.8
Data Bypass	1.6
Functional Units	1.0

Table 6. Energy Growth Parameters.

4. Pipeline Width Analysis

4.1. Width: Performance Scaling

Performance data for architectures with varying pipeline widths are obtained from the reference 4-decode (4D) design by a linear scaling of the number of functional units and the number of non-branch instructions fetched, decoded, renamed, dispatched, and retired per cycle (Table 5). All pipelines have at least one instance of each functional unit. As pipeline width decreases, the number of instances of each functional unit is quickly minimized to one. Thus, the decode width becomes the constraining parameter for instruction throughput for the narrower pipelines we consider (*i.e.* 2D and 1D).

4.2. Width: Power Scaling

A relatively optimistic power scaling technique assumes unconstrained hold and switching power increases linearly with the number of functional units, access ports, and any other parameter that must change as width varies. We expect linear power scaling to effectively estimate changes in power dissipation for functional units since we employ a clustered architecture. Superlinear power scaling effectively reduces to an approximate linear scaling for clustered structures[15, 16, 17]. Furthermore, cache port scaling by replicating a 1-read, 1-write port cache to obtain a 2-read, 1-write port cache for the Power-4 architecture, modeled by [ST,19,4], also suggests linear power scaling is applicable for this microarchitectural structure [18, 19].

In certain cases, however, linear power scaling is an optimistic first-order approximation and, for example, does not capture non-linear relationships between power and the number of register file access ports since it does not ac-

count for the additional circuitry required in a multi-ported SRAM cell. For this reason, we formulate a relatively pessimistic estimate of power dissipation trends as pipeline width varies by applying superlinear power scaling with exponents (Table 6) drawn from Zyuban's work in estimating energy growth parameters [15]. These parameters form a pessimistic power estimate since the values are experimentally derived from non-clustered architectures and tend to overestimate energy growth for clustered architectures.

4.3. Width: Power-Performance Evaluation

As in the pipeline analysis, Figures 5–6 depict microarchitectural tuning curves with linear and superlinear power scaling, respectively. These figures demonstrate the effects of tuning pipeline width for each architecture given a fixed pipeline depth. Note that pipeline width is quantified by the number of non-branch instructions decoded per cycle and faster decode rates correspond to wider pipelines.

Hardware tuning curves are drawn through power-performance efficient design points. Under linear power scaling, a pipeline width of 8D is found to maximize $\frac{BIPS^3}{W}$ for all architectures except the ST baseline; the efficient ST width is 4D. As in the depth analysis, the optimality of these design points can be demonstrated by considering an alternative design point and showing it cannot achieve lower delay for the same power budget.

Comparing the architectures executing one thread per core (*i.e.* ST, SMT1c1t, CMP2c1t) and those executing multiple simultaneous threads per core (*i.e.* SMT1c2t, CMP2c2t) with linear power scaling, we find executing a second thread motivates increasing pipeline width to improve efficiency. In particular, we find the 4D and 8D design points offer comparable efficiency when executing one thread, but choosing the wider 8D pipeline for architectures executing multiple threads can reduce effective delay by approximately 6 to 8 percent after hardware tuning to meet power constraints imposed by their respective original 4D design points. This observation may be drawn from Figure 5 by noting that microarchitectural tuning curves for single-threaded workloads track the hardware tuning curves more closely than their multi-threaded counterparts around the 4D and 8D design points.

As expected, superlinear power scaling decreases the power-performance optimal width to 4D for architectures executing a single thread per core and reduces the benefits of using the wider 8D design for architectures executing multiple threads per core. The hardware tuning curves in Figure 6 track the microarchitectural tuning curves more closely from 4D to 8D compared to those in Figure 5. This implies smaller efficiency penalties for choosing the 4D design over the 8D design.

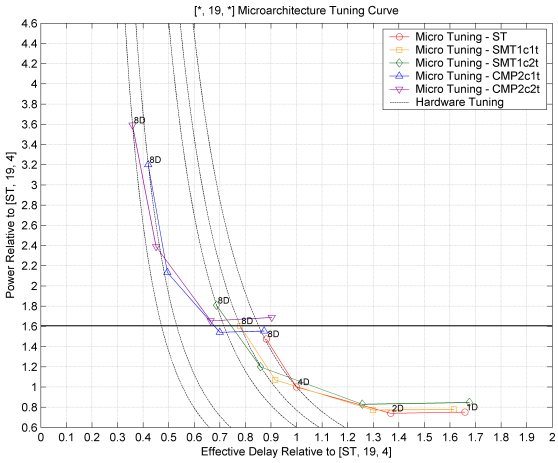


Figure 5. [*,19,*] Micro Tuning Curves - Linear.

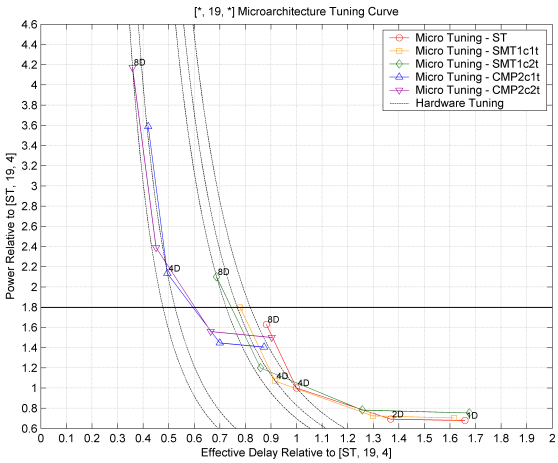


Figure 6. [*,19,*] Micro Tuning Curves - Superlinear.

Overall, these results are consistent with the conventional wisdom that SMT architectures perform better in wider pipelines. Conceptually, the larger number of instructions capable of entering a wider pipeline is better exploited when multiple thread contexts are executing simultaneously.

4.4. Width: Complexity Implications

Since no other design point improves performance while meeting the power constraints imposed by [ST,19,4] with linear power scaling in Figure 5, we instead consider a power budget of 1.7 defined by the relative power dissipation of [SMT1c1t,19,8]. Alternative designs that meet this power budget are [SMT1c2t,19,$4 < w < 8$] and [CMP2c1t,19,$2 < w < 4$].[4] Note each of these designs em-

ploy cores with narrower pipelines while reducing delay.

As in the pipeline analysis, these alternative designs with less complex cores may not be power-performance efficient. [CMP2c1t,19,$2 < w < 4$] is inefficient since it is possible to achieve higher performance for the same power dissipated by increasing the pipeline width to 4D or 8D and tuning the hardware to meet power constraints. This corresponds to moving up the CMP2c1t tuning curve from [CMP2c1t,19,$2 < w < 4$] to [CMP2c1t,19,4] or [CMP2c1t,19,8], then moving down the hardware tuning curve until the relative power dissipation is 1.7. At this new, efficient design point, the power constraints are satisfied and delay is reduced by approximately 33 percent relative to [SMT1c1t,19,8], compared to an approximately 12 percent reduction achieved from [CMP2c1t,19,$2 < w < 4$].

With the relatively pessimistic superlinear power scaling in Figure 6, however, the efficiency differences between the 8D and 4D designs are negligible for both SMT and CMP architectures. In these cases, the 4D design is preferrable since it is likely much less complex than the 8D design point.

Note the analysis thus far has emphasized the relative efficiency of the 8D and 4D design points. In no case, however, does the data support moving to cores narrower than the baseline 4D design. The [CMP2c2t,19,8] design point reduces effective delay by 45 percent over the baseline [ST,19,4] after hardware tuning with linear power scaling. In contrast, the [CMP2c2t,19,2] design point only achieved a 30 percent reduction in delay without hardware tuning. Tuning the hardware for the 2D design would only further reduce performance gains. Thus, these analyses suggest reducing core complexity in multi-core architectures by employing narrower pipelines are not power-performance efficient. As with pipeline depth, hardware tuning more complex cores with wider pipelines to meet power budgets may be more efficient.

5. Complexity Effective Design

The preceding analyses suggest limited opportunities for efficiently reducing core complexity in CMP architectures. In particular, we draw the following conclusions:

1. Employing cores with shallower or narrower pipelines is power-performance inefficient since hardware tuning an efficient design achieves higher performance for the same power dissipation.
2. In the cases where multiple design points are efficient, designers are able to choose a complexity effective design among these efficient alternatives.
3. Given a need to reduce complexity, the efficiency penalties for shallower pipelines are less than those for narrower pipelines.

The analyses from Section 3 and Section 4 find simply reducing pipeline depth or width is inefficient. Superior alternatives, from the perspective of power and performance, employ an efficient core in a multi-core architecture and tune the hardware to meet power constraints. Thus, designers should not rely on trends toward a larger number of less complex cores to meet power or performance targets, because such trends are likely to be inefficient. Overall, this conclusion implies microprocessor core design continues to play a significant role in CMP architectural development.

Multiple efficient design points exist in both depth and width analyses. In particular, designs ranging from 14 to 24 FO4 track the 3 percent, 1 percent power-performance trade-off of the hardware tuning curves, suggesting these designs are approximately equivalent from an efficiency perspective. Similarly, the 4D and 8D designs offer approximately the same efficiency. Choosing less complex designs from these efficient choices, enables designers to manage complexity among efficient alternatives.

The analyses in this paper seeks to maximize performance for a given power budget or minimize power for a given performance target. In the case where complexity constraints must also be met at the expense of power-performance efficiency, decreasing pipeline depth incurs less of an efficiency penalty relative to penalties incurred from decreasing pipeline width. For example, [CMP2c2t,36,4] incurs a 12 percent performance penalty relative to the hardware tuned [CMP2c2t,18,4]. In contrast, [CMP2c2t,19,2] incurs a 25 percent performance penalty relative to the hardware tuned [CMP2c2t,19,4]. Thus, designers seeking to reduce complexity should focus on pipeline depth to minimize the impact on efficiency.

Although we only consider two cores per chip, the complexity, power, and area of interconnect between cores in a CMP architecture becomes increasingly relevant as the number of cores increase [16, 28]. This suggests a fundamental trade-off between core and interconnect complexity. Employing many low complexity cores necessarily implies a more complex interconnect network. Conversely, as core complexity increases, fewer cores per chip are needed to achieve the same performance targets and a less complex interconnect network is required. Thus, complexity is effectively transferred from the interconnect to the CMP cores. Understanding this complexity trade-off is future work.

This work also neglects area effects which may become increasingly significant as the number of cores increase. Reducing core complexity while increasing the number of cores per chip may produce a net increase in throughput per

4 We employ this interval notation due to limited granularity in our design exploration space.

unit area, a metric not considered in this paper. Accounting for these effects is future work.

6. Related Work

The experimental work in this paper combines prior research in optimizing pipeline depths and power-performance analyses for SMT and CMP architectures.

6.1. Optimizing Pipeline Depth

Zyuban, *et al.*, [14] found 18 FO4 delays to be the power-performance optimal pipeline design point for a single-threaded microprocessor. The authors also introduced the microarchitectural tuning curves for graphically analyzing a design's relative position in the power-performance space.

Most prior work in optimizing pipeline depth focused exclusively on improving performance. Kunkel, *et al.*, [20] demonstrated that vector code performance is optimized on deeper pipelines while scalar codes perform better on shallower pipelines. Dubey, *et al.*, [21] developed a more general analytical pipeline model to show that the optimal pipeline depth decreases with increasing overhead from partitioning logic between pipeline stages.

More recent research includes finding optimal pipeline designs from simulation. In particular, Hartstein, *et al.*, [22] performed detailed simulations of a four-way superscalar, out-of-order microprocessor with a memory execute pipeline to identify a 10.7 FO4 performance optimal pipeline design for the SPEC2000 benchmarks. Similarly, Hrishkesh, *et al.*, [23] performed simulations for an Alpha 21264-like machine to identify 8 FO4 as a performance optimal design running the SPEC2000 benchmarks.

6.2. SMT, CMP Power-Performance Analyses

Li, *et al.*, [5] performed a comparative performance, power, and temperature analysis on SMT and CMP architectures. The authors found CMP architectures to be more power-performance efficient for CPU bound benchmarks and SMT architectures to be more efficient for memory bound benchmarks. The latter conclusion follows from the fact that SMT architectures are able to have larger L2 caches given a fixed area budget.

Other related work has examined the power-performance efficiency of SMT architectures. Li, *et al.*, [4] studied the efficiency of a POWER4-like architecture while Seng, *et al.*, [24] studied power optimizations for a multi-threaded Alpha processor. Sasanka, *et al.*, [6] and Kaxiras, *et al.*, [25] compare the relative efficiencies of SMT and CMP architectures for multimedia and signal processing workloads, respectively. Similarly, work by Kumar, *et al.*, [26] with heterogeneous CMP cores demonstrates these architectures produce a net increase in efficiency.

Prior studies have also considered hybrids of SMT and CMP designs (*e.g.* two CMP cores, each supporting two-way SMT), concluding that hybrid organizations with N thread contexts are generally inferior to pure CMP architectures with N full cores [6, 26, 27]. This conclusion is also supported in our work (SMT1c2t versus CMP2c1t).

The complexity, power dissipation, and area of interconnect between cores in a CMP architecture become increasing relevant as the number of cores increase [16, 28]. These considerations do not significantly impact the work presented in this paper since we consider CMP architectures with only two cores.

7. Conclusions and Future Directions

SMT architectures offer opportunities to efficiently increase pipeline dimensions and, consequently, core complexity. In contrast, reducing pipeline dimensions in CMP cores is potentially power-performance inefficient, assuming ideal power-performance scaling from hardware tuning.

We will continue to develop power scaling techniques for pipeline analysis. Preliminary work in dividing microarchitectural structures into primitive building blocks and performing circuit-level power analyses on these blocks may improve existing analytical power models. We expect this hierarchical modeling scheme will enable faster and more accurate characterization of power scaling trends.

We also intend to consider heterogeneous trace pairs for simultaneous execution in our continuing work. Pairing CPU bound traces with memory bound traces might better utilize architectural resources and demonstrate higher power-performance efficiency.

We take power and performance to be the primary metrics in this study, but area and interconnect effects will become significant as we continue our work in CMP architectures for a larger number of cores. Accounting for these other design parameters and metrics is future work.

We intend to integrate the pipeline depth and width analyses for a more comprehensive understanding of the design space. This is a first step towards developing statistical regression models that will enable architectural designers and researchers to interpolate the power-performance effects of varying a pipeline design parameter without a large number of simulations.

References

[1] R. Kalla, B. Sinharoy, J. Tendler. Power5: IBM's Next Generation Power Microprocessor. In *Proc. 15th Hot Chips Symposium*, Aug 2003.

[2] D.T. Mar, F. Bins, D.L. Hill, G. Hinton, D.A. Koufaty, J.A. Miller, M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1), Feb 2002.

[3] K. Krewell. UltraSPARC IV Mirrors Predecessor: Sun Builds Dual-Core Chip in 130nm. *Microprocessor Report*, Nov 2003.

[4] Y. Li, D. Brooks, Z. Hu, K. Skadron, P. Bose. Understanding the Energy Efficiency of Simultaneous Multithreading. In *Proc. ISLPED*, Aug 2004.

[5] Y. Li, D. Brooks, Z. Hu, K. Skadron. Performance, Energy, and Thermal Considerations for SMT and CMP Architectures. In *Proc. HPCA*, Feb 2005.

[6] R. Sasanka, S.V. Adve, Y.K. Chen, E. Debes. The Energy Efficiency of CMP vs SMT for Multimedia Workloads. In *Proc. ICCD*, Sep 2000.

[7] M. Moudgill, P. Bose, J. Moreno. Validation of Turandot, a Fast Processor Model for Microarchitecture Exploration. In *Proc. IEEE International Performance, Computing, and Communications Conference*, Feb 1999.

[8] M. Moudgill, J. Wellman, J. Moreno. Environment for PowerPC Microarchitecture Exploration. *IEEE Micro*, May/Jun 1999.

[9] D. Brooks, P. Bose, S. Schuster, H. Jacobson, P. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, P. Cook. Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors. *IEEE Micro*, Nov/Dec 2000.

[10] D. Brooks, P. Bose, V. Srinivasan, M. Gschwind, P. Emma, M. Rosenfield. Microarchitecture-Level Power-Performance Analysis: The PowerTimer Approach. *IBM J. Research and Development*, vol. 47, nos. 5/6, 2003.

[11] J.S. Neely, H.H Chen, S.G. Walker, J. Venuto, T. Bucelot. CPAM: A Common Power Analysis Methodology for High-Performance VLSI Design. In *Proc. Ninth Topical Meeting Electrical Performance of Electronic Packaging*, 2000.

[12] V. Iyengar, L.H. Trevillyan, P. Bose. Representative Traces for Processor Models with Infinite Cache. In *Proc. HPCA-2*, Feb 1996.

[13] V. Zyuban, P. Strenski. Balancing Hardware Intensity in Microprocessor Pipelines. *IBM J. Research and Development*, vol. 47, nos. 5/6, 2003.

[14] V. Zyuban, D. Brooks, V. Srinivasan, M. Gschwind, P. Bose, P. Strenski, P. Emma. Integrated Analysis of Power and Performance for Pipelined Microprocessors. *IEEE Transactions on Computers*, Aug 2004.

[15] V. Zyuban. Inherently Lower-Power High-Perfomrance Superscalar Architectures. Ph.D. Thesis, University of Notre Dame, Mar 2000.

[16] K. Ramani, N. Muralimanohar, R. Balasubramonian. Microarchitectural Techniques to Reduce Interconnect Power in Clustered Processors. *5th Workshop on Complexity-Effective Design (WCED), in conjunction with ISCA-31*, Jun 2004.

[17] P. Chaparro, J. Gonzlez and A. Gonzlez. Thermal-Aware Clustered Microarchitectures. *Proc. of the IEEE International Conference on Computer Design*, Oct 2004.

[18] J.M. Tendler, J.S. Dodson, J.S. Fields,Jr., H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM J. of Research and Development*, vol. 46, no. 1, 2002.

[19] J.D. Warnock, J.M. Keaty, J. Petrovick, J.G. Clabes, C.J. Kircher, B.L. Krauter, P.J. Restle, B.A. Zoric, and C.J. Anderson. newblock The Circuit and Physical Design of the POWER4 Microprocessor. newblock *IBM J. of Research and Development*, vol. 46, no. 1, 2002.

[20] S.R. Kunkel, J.E. Smith. Optimal Pipelining in Supercomputers. In *Proc. ISCA-13*, Jun 1986.

[21] P. Dubey, M. Flynn. Optimal Pipelining. In *J. Parallel and Distributed Computing*, 1990.

[22] A. Hartstein, T.R. Puzak. The Optimum Pipeline Depth for a Microprocessor. In *Proc. ISCA-29*, May 2002.

[23] M.S. Hrishikesh, K. Farkas, N.P. Jouppi, D.C. Burger, S.W. Keckler, P. Sivakumar. The Optimal Logic Depth per Pipeline Stage is 6 to 8 FO4 Inverter Delays. In *Proc. ISCA-29*, May 2002.

[24] J. Seng, D. Tullsen, G. Cai. Power-Sensitive Multi-threaded Architecture. In *Proc. ICCD 2000*, 2000.

[25] S. Kaxiras, G. Narlikar, A.D. Berenbaum, Z. Hu. Comparing Power Consumption of SMT DSPs and CMP DSPS for Mobile Phone Workloads. In *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, Nov 2001.

[26] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, D.M. Tullsen. Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction. In *Proc. ISCA-31*, Jun 2004.

[27] J. Burns, J.L. Gaudiot. Area and System Clock Effects on SMT/CMP Processors. In *Proc. PACT 2001*, Sep 2001.

[28] R. Kumar, V. Zyuban, D. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In *Proc. ISCA-32*, Jun 2005.

Early Performance Prediction

P. Kudva
IBM T.J.Watson Research Center
Yorktown Heights, NY

B. Curran
IBM Server and Technology Group
Poughkeepsie, NY

S. K. Karandikar
IBM Austin Research Lab
Austin, TX

M. Mayo
IBM Server and Technology Group
Poughkeepsie, NY

S. Carey
IBM Server and Technology Group
Poughkeepsie, NY

S. S. Sapatnekar
University of Minnesota
Minneapolis, MN

ABSTRACT

Critical paths and delays of high performance circuits can be accurately identified only in later stages of the design cycle. If the specified clock period is not achievable, changes at the RTL-level may be necessary, leading to re-synthesis and re-validation of the entire design. This, in turn, can cause schedule overruns in the time to market and hence competitiveness of the design. Designers therefore need performance predictors that can identify potential problem areas early in the design flow. In this work, we identify the main characteristics of a design that affect the clock period, and propose models for estimating it at early stages of the design process. Techniques similar to the ones presented in this paper have been evaluated and incorporated into a microprocessor design methodology.

1. INTRODUCTION

Microprocessor designs are complex enough to require a few years from specification to tape-out. Market pressures dictate tight schedules, and overruns in these schedules can be costly not only in monetary terms, but also in the competitiveness of the design. A frequent cause of schedule breakdowns is due to final implementation parameters such as area, power and clock period (and hence frequency) not meeting the original specifications.

In order to evaluate the performance of a design in the early stages of the flow, designers typically use system and microarchitectural models. These can be enhanced to include physical information, such as the effects of long wires between microarchitectural blocks [1, 2]. However, these improvements do not capture the effects of the logic functionality of the design. Arguably, the complexity of the logic being implemented has a big effect on the performance of the design – very complex architectures increase design complexity, thereby adding to cycle time pressures, power consumption as well as verification and validation challenges. Additionally, changes to complex microarchitecture require significant redesign effort, and can cause schedule slips.

It is therefore imperative to have a methodology to evaluate the hardware complexity of early microarchitectures, to allow for adequate exploration of the system space. Further, as the microarchitecture is implemented in an HDL, these early estimations should become more accurate. Determining complexity from microarchitectural models themselves is an objective of ongoing and future research. In this paper, we focus on early RTL models derived in the process of converting microarchitectural models to RTL specifications. In the early stages of this step, RTL specifications are crude and incomplete. Additionally, technology rules for gates/wires may not be available, and have to be extrapolated from past technologies. As implementation progresses, the RTL is further refined and more technology specific information is available. Such incomplete specifications cannot be synthesized, and it is therefore difficult to use well known synthesis-based early estimation techniques [3–5, 7] to estimate design complexity. Estimating the performance of such incomplete designs is still necessary to provide early feedback to microarchitects.

In this scenario, if designers were to be provided a metric for estimating design performance *early* in the design flow, they would be able to better identify potential problem areas, and would be able to use this information to guide their design process. In this work, we address the *clock period* of the design, which is possibly the most visible parameter of the final product.

In order to avoid costly iterations of the design cycle, we propose models that allow a designer to quickly determine which signal paths are critical, and focus his or her efforts on these areas. Feedback is also provided to microarchitects regarding extremely challenging paths in order to consider changes. Since this information is made available early in the design process, we hope to avoid expensive redesign, resynthesis and revalidation. The models provide information that both the logic designers and microarchitects can use to make intelligent choices between different available implementations. Additionally, we can also identify paths that are non-critical; these can then be targeted for aggressive pipeline rebalancing, area or power optimization.

We have analyzed a number of macros in the various units of a latest-generation microprocessor. In the following sections we present the results of this analysis, and propose a number of models that can be used at different stages of the design flow to estimate the delay of an implementation. As the design is successively refined, more data becomes available, and our proposed models can use this data to provide more accurate data. We have tried to abstract out technology dependent parameters from our models, so that they can be used for different technology generations. Rather than absolute accuracy, our goal is to be able to predict which parts of the circuit could prove to be problematic, and therefore need more attention from designers.

We first put our contributions in context by describing the typical microprocessor design flow. We then present the data accumulated and our analysis of this data, followed by various predictive models. We conclude with a discussion of additional approaches and future research directions

2. DESIGN FLOW

Figure 1 shows a typical, albeit simplified design flow of a high performance, complex microprocessor. At the initial stages, only system models are available. These define stages in the pipeline, the flow of data and interactions between these stages, along with

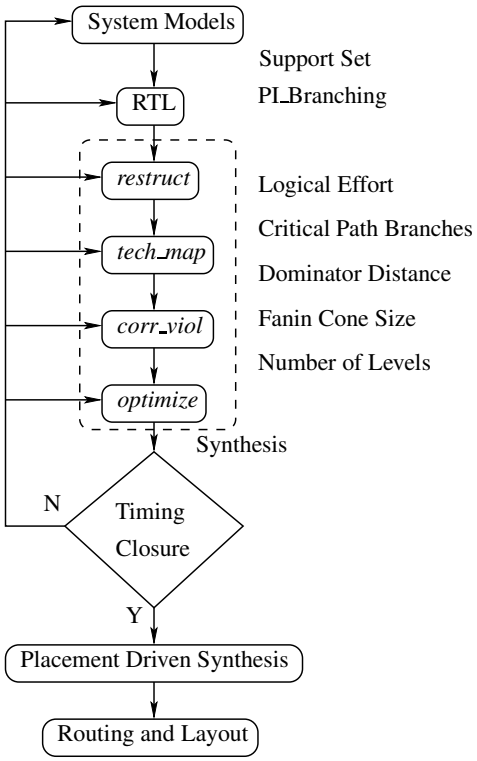


Figure 1: Typical Design Flow

cycle accurate models. Logic designers use these models to implement control logic in a HDL. This is the stage where the design starts assuming a concrete form. Signals are initially defined in terms of boolean equations, and as RTL is developed and synthesized, we obtain a logic gate level netlist. At each stage of synthesis, the design is successively refined. Initial technology independent optimizations such as kernel factoring, literal minimization, etc. (performed at the step labelled *restruct*), are followed by technology mapping (*tech_map*), where the design is mapped to a specific technology library. This is followed by a number of technology dependent transformations for correcting violations (*corr_viol*) and further design optimizations (*optimize*), which use gate and wire sizing, buffer insertion, etc. Not surprisingly, the delay of the design can vary wildly during synthesis. As an example of how much the design changes, Figure 2 presents the delays of critical paths in a number of industrial circuits at different points during synthesis, normalized to the delay at the RTL stage. Not only do the values of critical path delays change, but the actual critical paths themselves can vary quite widely from pre-RTL to the final circuit. In some cases, the final delays after synthesis are close to the initial delays, while in others they are twice as large. As is obvious, these delay values fluctuate in a wide range during the synthesis transforms, being smaller than the initial values in some cases, and larger in others.

Figure 1 also shows a few parameters that are defined and analyzed in the following section. We point out here that these parameters are available at different stages of the design flow; intially only structural parameters such as the support set and PI_Branching (which will be defined shortly) can be calculated, while parameters such as logical effort, size of fanin cone, etc. are determined as the design is implemented and synthesized.

Once a mapped, optimized circuit is obtained, its delay can be

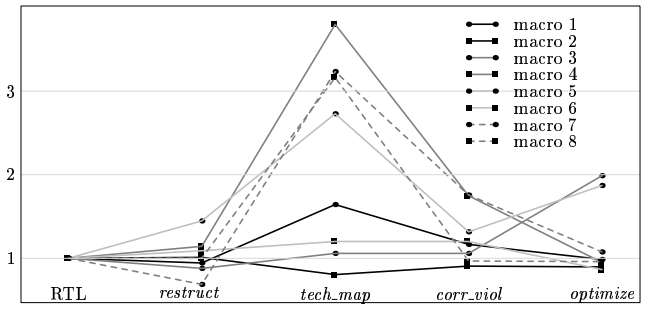


Figure 2: Delay Variation During Synthesis

accurately measured. If the implemented circuit meets the original specifications, the design continues to the next stages of placement, layout and routing, and can go on to silicon fabrication. However, if the delay of the circuit at this stage does not meet the original specification, the designer has to return to preceding stages in the design process and attempt to fix these violations. Depending on how much difference there is in the implementation versus the specification, changes have to be made at different stages of the design flow. All changes require re-synthesis, which can be expensive. In addition, validation, which usually runs concurrent with synthesis, and which has its own cost, has to be repeated. Even small changes, if made at the RTL (or earlier) stages can be expensive, if they affect a large portion of the design. Thus, any loops in the design flow caused by timing closure issues can cause schedule slips, which in turn affect the time that a design can be brought to market.

Prediction, at an early stage, need not be very accurate – even separating the circuit into high, medium and low delay critical portions provides more information than currently available. There is greater freedom to make changes in the design at early stages, and the effect of changes made early in the design flow is greater. Hence, the earlier a designer has an estimate of the performance of the design, the greater the benefit of any predictor. We develop models that can estimate delays as the designer is starting to implement RTL. The accuracy of these models is limited due to the limited design properties available at this stage. As the design matures, our predictors provide successively accurate delay models.

3. PARAMETERS AFFECTING DELAY

3.1 Preliminaries

A combinational path is a path without any latches. The clock period of a design is defined by the length(delay) of the longest combinational path. This critical path may be from primary inputs or latches to primary outputs or latches. We only consider paths between latches, since a path containing a primary input (output) of a macro may be part of a larger combinational path that crosses macro boundaries.

Referring once again to Figure 1, different design properties are available at different stages. The first stage in the design development is the RTL description of the circuit. At this point, the only information available for each primary output is the size of its support set. After *restruct*, which applies technology independent optimizations to the circuit, coarse information on the number of levels and branching is available, but these can change significantly in the following steps. The following step, labelled *tech_map* is the technology mapping step, where cells are mapped to actual library elements. At this stage, we can take all the data available into consideration. However, we note once again that this is sub-

ject to further refinement by following optimizations. The last step in the physical synthesis process is *optimize*, where optimizations such as gate sizing are applied. At this point, the entire spectrum of data is available, since this is at a point very close to the final circuit.

This availability of design properties is also presented in Figure 1. Parameters such as support set and PI branching are available early in the design process. These have some bearing on the delay of a design. Parameters that have a much larger effect on the delay, such as logical effort and branching, on the other hand, are available only after the design is completed. However, a designer can make an educated guess as to the values of these parameters early in the design cycle, and we can potentially use these values when building predictors. Note that even after a design has been finalized and is passed on the the next steps, placement and routing can introduce long wires into the design. At today's circuit geometries, wire delays are of the same order as device delays and cannot be ignored. Long wires have to be buffered, and this can significantly change the number of levels.

3.2 Discussion of Parameters

We now present the design properties under consideration and hypothesize as to their effect on actual delay. Section 4 describes how this data is used in building prediction models that can be used in microprocessor design methodologies.

1. **Support Set** (Figure 3): The support set is the number of primary inputs driving the fanin cone of a primary output. It is an invariant that does not change during synthesis. A larger support set implies more complicated logic, and hence greater delay. Even if the logic is simple, combining a large support set requires that the signals pass through more levels of logic, hence increasing the delay. The logarithm of the support set can give us a lower bound on the number of levels of logic in the design, and we hypothesize that the delay of the design depends directly on this value.

   Figure 3 is a plot of the delay (after synthesis) versus the support set of a number of combinational logic paths. As indicated by the upper right region, outputs with large support sets have large delays. For outputs with small support sets (the points to the left of the plot), we observe a larger variation in delay. This effect is due to the synthesis process – since these paths are not critical, delay is traded off for area. However, a clearly defined lower bound on the delay, as a function of the size of the support set can be observed. This indicates that a design with a large support set will have large delays, but when the size of the support set is smaller, other parameters have a greater influence on the delay. Thus, a good delay estimator will have to include a combination of different parameters.

2. **PI_Branching** (Figure 4): We count the number of primary outputs driven by each primary input, and then for a primary output, we calculate the accumulation (the sum or the product) of the number of primary outputs driven by all inputs in the support set of this primary output. This parameter is called the PI_Branching of an output. This parameter indicates how "self-contained" the fan-in cone of an output is. A low value corresponds to an independent cone of logic, while a larger value implies divided loyalties – the logic is driving multiple outputs. A primary input driving more than one primary output will have divergent paths *somewhere* in the circuit. This divergence implies that a larger electrical load that has to be driven, which directly leads to a larger delay. PI_Branching captures this effect at the output of a cone of logic.

   In Figure 4, the first plot presents the delay versus *product* and the second figure plots the delay versus the *sum* of the number of primary outputs driven by each input in the support set. These plots seems to indicate that there is a weak correlation between the delay and PI_Branching, but our models show a surprisingly strong effect on the estimating function, as shown in the following section.

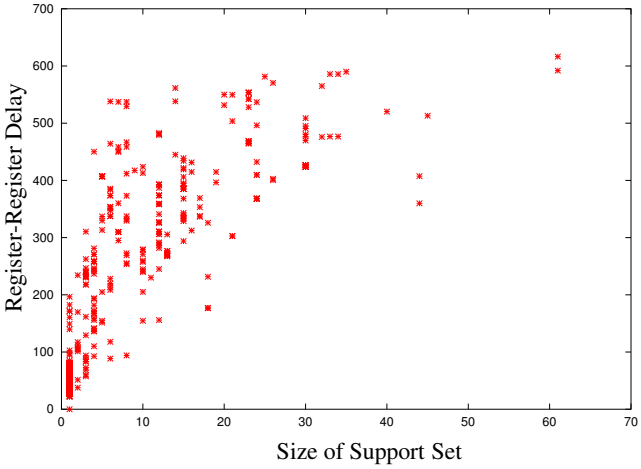


Figure 3: Delay Vs. Size of Support Set

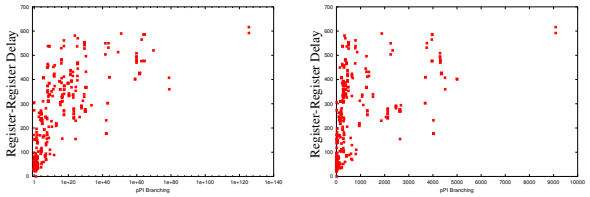


Figure 4: Delay Vs. PI_Branching

3. **Size of Fan-in Cone** (Figure 5): The size of the fanin-in cone is the number of gates in the fan-in cone of the primary output under consideration. This naturally changes as synthesis proceeds, and can only be determined at later stages of the design. Intuitively, the more logic driving a given output, the slower it will be. This can be seen by the plot.

   Note that this parameter has to be used with care, since the same functionality can be implemented with either one complex gate or a number of simple gates. The choice of which solution is selected depends on a number of factors, but the effect on the size of the fan-in cone can be at odds with the effect on delay.

4. **Branching** (Figure 6): We calculate the accumulated product of the fanouts of each gate on the path under consideration. In general, the larger the branching, the greater the load that has to be driven, which increases the delay of that path. We present the branching on the critical path as well as the average branching over all paths, normalized by the path length.

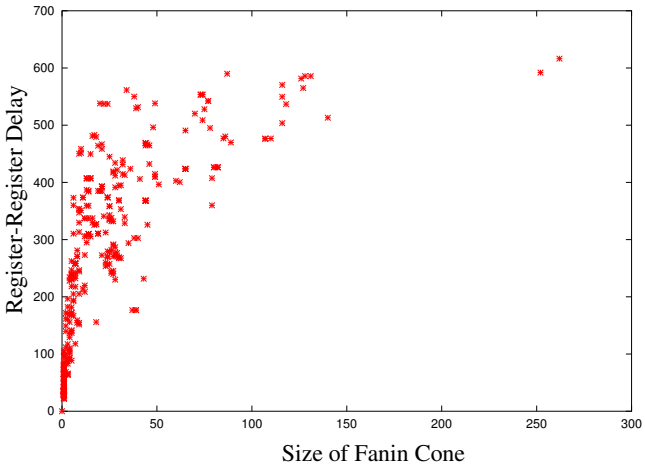


Figure 5: Delay Vs.Size of Fan-in Cone

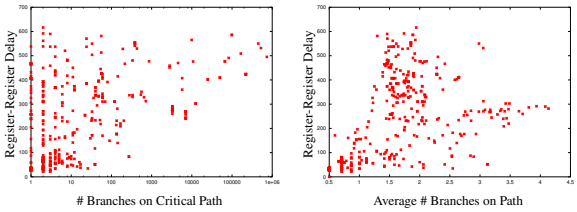


Figure 6: Delay Vs. Path Branching(Critical Path and Average)

5. **Logical Effort** (Figure 7): Logical effort measures the complexity of the gates that implement the required functionality. The logical effort along a path is the *product* over the gates on the path. Figure 7(b) is the average logical effort in the fanin cone of the output under consideration. Naturally, these values are available only after after some synthesis effort such as restructuring, and while they can be used at this stage, they can and do change during subsequent optimizations.

6. **Number of Levels** (Figure 8): This is simply the number of levels of logic that a signal traverses from input to output. Intuitively, delay is directly proportionally to this value. However, this depends on the load being driven: it can be shown that deeper paths can better drive large loads [6]. The number of levels converges after technology mapping and buffering, and can be reliably used only at the final stage.

4. MODELS FOR DELAY PREDICTION

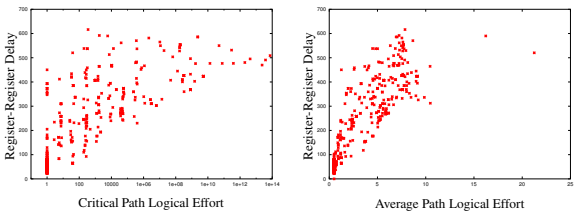


Figure 7: Delay Vs. Path Logical Effort (Critical Path and Average)

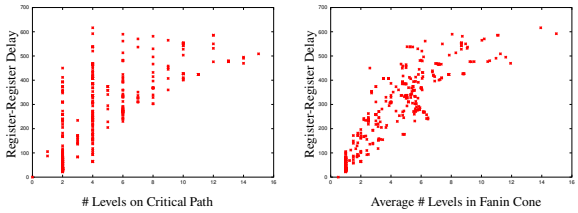


Figure 8: Delay Vs. Number of Levels (Critical Path and Average)

A direct correlation between delay and the parameters presented in the previous section seems logical – the larger the support set, number of levels and fan-in cone size, the greater the expected delay. However, this is not true for smaller values of these parameters. As mentioned before, this can be due to a number of different factors, such as synthesis optimizing for area or power (and therefore increasing delay) in non-critical portions of the design. Sometimes the discrepancies are due to the nature of the parameters themselves – the logical effort of a long chain of buffers is low, but the delay of such a chain can be large.

The above data therefore provides us with a *lower bound* on the delay; given a value of support set, say, we can be relatively sure of the minimum achievable delay. The data presented in the previous section shows this general trend to varying degrees. With this data in hand, we now attempt to determine *which* parameters can be used to predict delay, *how* these parameters can be combined, and reason *why* the parameters play the role that they do.

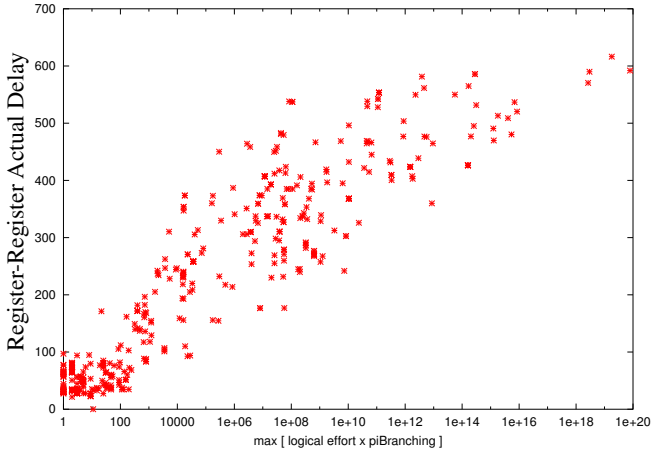


Figure 9: Delay Vs. max (Logical Effort $\times$ PI_Branching)

Consider the previous example of a long chain of buffers, which have a low value of logical effort, but high delay. The single parameter of logical effort, by itself, does not predict the large delay. However, a parameter such as the number of levels would indicate that the delay of such a chain will be high. Thus, while we cannot estimate delay as a function of a single parameter, combinations of different parameters can potentially serve as better delay estimators. The reasoning is that each parameter captures one aspect of the design, and the correct combination of these will be able to capture the overall nature of the design.

As another example of the correlation between a combination of the above parameters and delay, refer to Figure 9. For every cone of logic, we obtain the maximum value of the product of the logi-

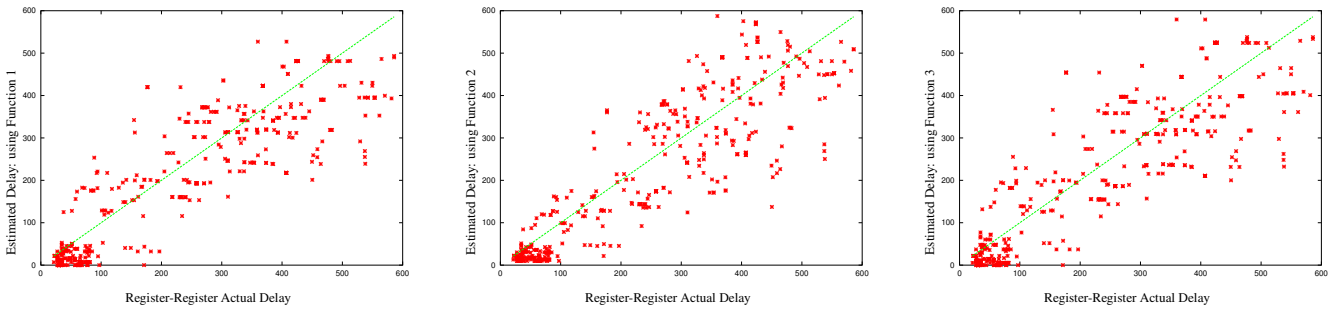


Figure 10: Estimated Vs. Actual Delay (Functions 1-3)

cal effort and the PI_Branching, analogous to the product of logical effort and electrical effort when calculating the gain of individual gates in the method of logical effort [6]. The improvement as compared to previous data is immediately apparent – the lower bound seen before is maintained, while a rough upper bound can be discerned.

An interesting point to note is that all the parameters that we have presented are dimensionless. In order to determine the best models for predicting delay, we are therefore free to determine the delay dependency – be it a linear, quadratic, exponential or some other function of a selected parameter. In the rest of this section, we hypothesize on the relation between delay and different parameters, and test these hypotheses by comparing the difference in actual and estimated delay.

The parameters are used in the order they are available to the designer, as shown in Figure 1. We start with the support set and PI_Branching, and then expand the models to include the remaining parameters.

Function 1

At the initial stages of the design, the only parameters available are the support set (*ss*) and PI_Branching. Our first attempt is based on the assumption that delay is possibly a linear function of the log of two parameters, i.e. $delay \propto \log ss + \log PI\_Branching$. The intuition behind this assumption is as follows. We have observed in the previous section that the support set gives a reasonable lower bound on the delay (Figure 3). However, for logic that has small support sets but higher delay, this parameter by itself is insufficient. The second parameter available to us at this stage is PI_Branching, which is included to account for these circuits. However, this model had significantly large errors, which can be reduced by introducing a combination of support set and PI_Branching:

$$\begin{aligned} delay &= k_1 \cdot \log ss + k_2 \cdot \log PI\_Branching \\ &\quad + k_3 \cdot \log ss \times \log PI\_Branching \end{aligned} \tag{1}$$

We then perform regression analysis using Equation 1 and available data in order to determine the the values of coefficients k_1, k_2 and k_3 so that the difference between the actual delay and the function in Equation 1 is minimized. The values of these coefficients also indicate the relative importance of the corresponding term.

The average error is 23%, with a standard deviation of 57 *ps*. Figure 10(a) plots the estimated delay using this function versus the actual delay. As mentioned previously, these coefficients are specific to the technology that this circuit will be implemented in. However, they do indicate the relative importance of each parameter, and therefore such a weighted function of support set and PI_Branching can be used in other technology generations as well.

Function 2

We next add quadratic functions of support set and PI_Branching,

$$\begin{aligned} delay &= k_1 \cdot \log ss \\ &\quad + k_2 \cdot \log PI\_Branching \\ &\quad + k_3 \cdot (\log ss)^2 \\ &\quad + k_4 \cdot (\log PI\_Branching)^2 \\ &\quad + k_5 \cdot \log ss \times \log PI\_Branching \end{aligned} \tag{2}$$

obtaining an average error of 25% with a standard deviation of 58 *ps*. Figure 10(b) plots the estimated delay using this function Vs the actual delay.

As in the previous model, the relative weights of the *ss* and PI_Branching parameters are similar multiples of the combined *ss* and PI_Branching parameter. The coefficients of the quadratic versions of these parameters are attenuated, in fact these do not contribute to accurate delay estimation at all, only increasing the error. This indicates that the quadratic terms are not needed when estimating delay.

Function 3

In this model, we include the average logical effort of the cone of logic under consideration. Though the exact value is not available at early stages, designers can make an educated guess based on the complexity of the logic to be implemented.

$$\begin{aligned} delay &= k_1 \cdot \log ss \\ &\quad + k_2 \cdot \log PI\_Branching \\ &\quad + k_3 \cdot (\log ss)^2 \\ &\quad + k_4 \cdot (\log PI\_Branching)^2 \\ &\quad + k_5 \cdot \log ss \times \log PI\_Branching \\ &\quad + k_6 \cdot leAvg \\ &\quad + k_7 \cdot \log ss \times leAvg \\ &\quad + k_8 \cdot \log PI\_Branching \times leAvg \end{aligned} \tag{3}$$

The estimated delay using this function, plotted against the actual delay in Figure 10(c) has an average error of 26% and a standard deviation of 56 *ps*. Note that the regressions also

indicate the relative effect of each term on the delay, by proportionately increasing / decreasing the coefficients. This can guide us towards better models. For example, the dominant term in this model is the average logical effort. We will possibly get better results by removing the quadratic terms, this is currently under investigation.

Function 4

Finally, we use the product of PI_Branching and logical effort. The intuition for the delay dependence on these parameters in this model is as follows. The first term, $\log ss$, accounts for the inherent delay of an implementation – no matter what the actual logic being implemented, all the signals in the support set have to be combined at the output. The second term – PI_Branching $\times$ LE is analogous to an *RC* delay – the logical effort estimates the complexity of the logic, while PI_Branching accounts for the different loads being driven by that logic. This is the simplest and most accurate model:

$$\begin{aligned} delay &= k_1 \times \log ss \\ &\quad + k_2 \times \log(\max(PI\_Branching \times LE)) \end{aligned} \quad (4)$$

This has an the best average error and standard deviation among all the functions, of 20% and 47 *ps* respectively.

We build our models based on the assumption that the above parameters capture information that can be used to predict the delay of a cone of logic. Combinations of parameters obviously provide a better estimate than parameters taken singly. Unfortunately, there are an exponentially large number of ways in which these parameters can be combined to generate delay estimates. The functions presented above are a first cut, and only examine a fraction of all possibilities.

5. APPLICATION METHODOLOGY

The estimators presented in this paper were evaluated within a microprocessor design methodology in a number of different scenarios. Firstly, based on the maturity of the design, the appropriate function is selected. At early stages of the design process, since only limited data is available, Function 1 can be used to estimate delay. During intermediate stages, one of the other functions are selected. As the designer gets a better estimate of the complexity of the design being implemented, Function 4, which uses parameters that were unknown before, can be used to estimate the delay of different portions of the circuit. Thus, as the design progresses, more parameters are determined, which are subsequently used in increasingly accurate estimating functions.

As a high level design is evaluated, the architectural performance models are also converted into early VHDL representations for analysis purposes. At this stage, the VHDL models are not intended for synthesis but for quick estimation to evaluate complexity of the designs. Cycle time delays are estimated for different parts of the processor pipeline datapath and controller and a determination is made regarding whether the high level design is able to meet cycle time. If the early estimations show that architectural assumptions were impractical from a logic design point of view, this information is fed back to the architect for a change in the high level architectural design.

If on the other hand, the timing is challenging but not impractical, then the logic design for that partition is identified and additional resources would be assigned to them. In addition, they are meant to guide the logic designer in writing the VHDL description that will improve the timing critical sections. Note, once again, that these models are not intended to estimate the actual delays, rather they help the designer divide the circuit into parts that have high delay, and hence are critical, and the remainder, which are not, thus helping them focus their attention where required. At each stage, optimizations can be applied only when required, and the target of these optimizations (delay, area, power) can be determined based on how critical that circuit is. Aggressive optimizers take more time to run, and the above models can be used to determine the required degree of optimization.

Our estimators are also used to determine the effect of floorplanning changes, which are made all throughout the design process. In early stages of the design cycle, it is necessary to know the effect of floorplan changes on the timing of the design. On the other hand, at these stages, the RTL descriptions are not complete or only partially available. When critical paths cross partition/macro boundaries, it is necessary to evaluate whether the floorplan can meet the timing requirement. Fast estimation of timing during the floorplanning process is essential. Estimates for each partition are recalculated throughout as the design floorplan begins to converge as well as when the contents of the partitions begin to stabiliize. The timing estimates for the macros are combined with timing estimation at the chip level including interconnect estimation to derive the overall cycle time, thus allowing the designers to monitor the cycle time as the floorplan changes and the design evolves.

The results of such early analysis methodology is fed back as well as forward to microarchitects, logic designers, physical designers and floorplanners. These estimates prove invaluable in making early changes to the design point from all these different design considerations.

6. CONCLUSION AND FUTURE WORK

In this paper, we present a methodology at predicting cycle time of a proposed design *early* in a microprocessor design flow. Though the error in the predicted values may seem significant in absolute terms, it is important to note that these predictions are based on extremely coarse data and provide more information to designers than currently available. Also, we have a high level of fidelity with actual delays, which, at early stages of the design is more important than accuracy. An extended version of the work presented in this paper has been implemented in a microprocessor design methodology. In the methodology, the estimates are used both to provide feedback to the architects in selecting optimal high level designs as well as to the logic designers, floorplanners and physical designers to aid them in meeting challenging cycle time objectives.

The estimation approach presented in this paper continues to be refined to be able to capture methodology issues much earlier in the design process [8]. Methods for generating estimates even earlier in the design flow can be implemented by directly taking into account microarchitectural parameters (pipeline depth, branch prediction algorithm complexity, renaming algorithms etc) along with logic design parameters (wiring, cycle time, area, power). These models are currently being evaluated as part of current and future directions for this research.

7. REFERENCES

[1] CONG, J., JAGANNATHAN, A., REINMAN, G., AND ROMESIS, M. Microarchitecture evaluation with physical planning. In *Proc. ACM/IEEE Design Automation Conference* (2003), pp. 32–36.

[2] EMER, J., BINKERT, N., ESPASA, R., JUAN, T., AHUJA, P., BORCH, E., KLAUSER, A., LUK, C.-K., MANNE, S., MUKHERJEE, S. S., PATIL, H., AND WALLACE, S. Asim: A

performance model framework. *IEEE Computer 35*, 2 (2002), 68–76.

[3] NEMANI, M., AND NAJM, F. Delay estimation of VLSI circuits from a high level view. In *Proc. ACM/IEEE Design Automation Conference* (1998), pp. 591–594.

[4] OHM, S. Y., KURDAHI, F. J., DUTT, N., AND XU, M. A comprehensive estimation technique for high level synthesis. In *International Symposium on System Synthesis* (1995), pp. 122–127.

[5] RAGHUNATHAN, A., DEY, S., AND JHA, N. High level macro-modeling and estimation techniques for switching activity and power consumption. *IEEE Transactions on VLSI Systems 11*, 4 (2003), 538–557.

[6] SUTHERLAND, I., HARRIS, D., AND SPROULL, R. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kauffman, San Fransisco, CA, 1999.

[7] WALLACE, D., AND CHANDRASHEKAR, M. High level delay estimation for technology independent logic equations. In *Proc. International Conf. Computer-Aided Design* (1990), pp. 188–191.

[8] ZYUBAN, V., AND STRENSKI, P. N. Balancing hardware intensity in microprocessor pipelines. *IBM Journal of Research and Development 47*, 5 (2003), 585–598.

Wire Management for Coherence Traffic in Chip Multiprocessors

Liqun Cheng, Naveen Muralimanohar, Karthik Ramani, Rajeev Balasubramonian, John Carter
School of Computing, University of Utah *

Abstract

Improvements in semiconductor technology have made it possible to include multiple processor cores on a single die. Chip Multi-Processors (CMP) are an attractive choice for future billion transistor architectures due to their low design complexity, high clock frequency, and high throughput. In a typical CMP architecture, the L2 cache is shared by multiple cores and data coherence is maintained among private L1s. Coherence operations entail frequent communication over global on-chip wires. In future technologies, communication between different L1s will have a significant impact on overall processor performance and power consumption.

On-chip wires can be designed to have different latency, bandwidth, and energy properties. Likewise, coherence protocol messages have different latency and bandwidth needs. We propose an interconnect comprised of wires with varying latency, bandwidth, and energy characteristics, and advocate intelligently mapping coherence operations to the appropriate wires. In this paper, we present a comprehensive list of techniques that allow coherence protocols to exploit a heterogeneous interconnect and present preliminary data that indicates the potential of these techniques to significantly improve performance and reduce power consumption. We further demonstrate that most of these techniques can be implemented at a minimum complexity overhead.

1. Introduction

Advances in process technology have led to the emergence of new bottlenecks in future microprocessors. One of the chief bottlenecks to performance is the high cost of on-chip communication through global wires [19]. Power consumption has also emerged as a first order design metric and wires contribute up to 50% of total chip power in some processors [28]. Future microprocessors are likely to exploit huge transistor budgets by employing a chip multi-processor (CMP) architecture [30, 32]. Multi-threaded workloads that execute on such processors will experience high on-chip communication latencies and will dissipate significant power in interconnects. In the past, the design of interconnects was primarily left up to VLSI and circuit designers. However, with *communication* emerging as a larger power and performance constraint than *computation*, architects may wish to consider different wire implementations and identify creative ways to exploit them [6]. This paper presents a number of creative ways in which coherence communication in a CMP can be mapped to different wire implementations with minor increases in complexity. We present preliminary results that demonstrate that such an approach can both improve performance and reduce power dissipation.

In a typical CMP, the L2 cache and lower levels of the memory hierarchy are shared by multiple cores [22, 32]. Sharing the L2 cache allows high cache utilization and avoids duplicating cache hardware resources. L1 caches are typically not shared as such an organization entails high communication latencies for every load and store. Maintaining coherence between the individual L1s is a challenge in CMP systems. There are two major mechanisms used to ensure coherence among L1s in a chip multiprocessor. The first option employs a bus connecting all of the L1s and a snoopy bus-based coherence protocol. In this design, every L1 cache miss results in a coherence message being broadcast on the global coherence bus. Individual L1 caches perform coherence operations on their local data in accordance with these coherence

*This work was supported in part by NSF grant CCF-0430063 and by Silicon Graphics Inc.

messages. The second approach employs a centralized directory in the L2 cache that tracks sharing information for all cache lines in the L2 and implements a directory-based coherence protocol. In this design, every L1 cache miss is sent to the L2 cache, where further actions are taken based on directory state. Numerous studies [1, 10, 20, 23, 27] have characterized the high frequency of cache misses in parallel workloads and the high impact these misses have on total execution time. On a cache miss, a variety of protocol actions are initiated, such as request messages, invalidation messages, intervention messages, data block writebacks, data block transfers, etc. Each of these messages involves on-chip communication with latencies that are projected to grow to tens of cycles in future billion transistor architectures [2].

VLSI techniques enable a variety of different wire implementations that are typically not exploited at the microarchitecture level. For example, by tuning wire width and spacing, we can design wires with varying latency and bandwidth properties. Similarly, by tuning repeater size and spacing, we can design wires with varying latency and energy properties. To take advantage of VLSI techniques and better match the interconnect design to communication requirements, heterogeneous interconnects have been proposed [6], where every link consists of wires that are optimized for either latency, energy, or bandwidth. In this study, we explore optimizations that are enabled when such a heterogeneous interconnect is employed for coherence traffic. For example, on a cache write miss, the requesting processor may have to wait for data from the home node (a two hop transaction) and for acknowledgments from other sharers of the block (a three hop transaction). Since the acknowledgments are on the critical path and have low bandwidth needs, they can be mapped to wires optimized for delay, while the data block transfer is not on the critical path and can be mapped to wires that are optimized for low power.

The paper is organized as follows. Section 2 reviews techniques that enable different wire implementations and the design of a heterogeneous interconnect. Section 3 describes the proposed innovations that map coherence messages to different on-chip wires. Section 4 presents preliminary results that indicate the potential of our proposed techniques. Section 5 discusses related work and we conclude in Section 6.

2. Wire Implementations

We begin with a quick review of factors that influence wire properties. It is well-known that the delay of a wire is a function of its RC time constant (R is resistance and C is capacitance). Resistance per unit length is (approximately) inversely proportional to the width of the wire [19]. Likewise, a fraction of the capacitance per unit length is inversely proportional to the spacing between wires, and a fraction is directly proportional to wire width. These wire properties provide an opportunity to design wires that trade off bandwidth and latency. By allocating more metal area per wire and increasing wire width and spacing, the net effect is a reduction in the RC time constant. This leads to a wire design that has favorable latency properties, but poor bandwidth properties (as fewer wires can be accommodated in a fixed metal area). Our analysis [6] shows that in certain cases, nearly a three-fold reduction in wire latency can be achieved, at the expense of a four-fold reduction in bandwidth. Further, researchers are actively pursuing transmission line implementations that enable extremely low communication latencies [12, 16]. However, transmission lines also entail significant metal area overheads in addition to logic overheads for sending and receiving [8, 12]. If transmission line implementations become cost-effective at future technologies, they represent another attractive wire design point that can trade off bandwidth for low latency.

Similar trade-offs can be made between latency and power consumed by wires. Global wires are usually composed of multiple smaller segments that are connected with repeaters [5]. The size and spacing of repeaters influences wire delay and power consumed by the wire. When smaller and fewer repeaters are employed, wire delay increases, but power consumption is reduced. The repeater configuration that minimizes delay is typically very different from the repeater configuration that minimizes power consumption. Banerjee *et al.* [7] show that at 50nm technology, a five-fold reduction in power can be achieved at the expense of a two-fold increase in latency.

Thus, by varying properties such as wire width/spacing and repeater size/spacing, we can implement wires with different latency, bandwidth, and power properties. If a data packet has 64 bits, global interconnects are typically designed to min-

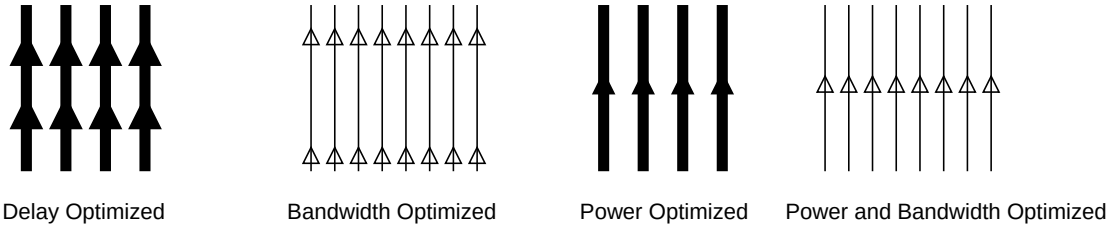


Figure 1. Examples of different wire implementations. Power optimized wires have fewer and smaller repeaters, while bandwidth optimized wires have narrow widths and spacing.

imize delay for the transfer of 64-bit data, while not exceeding the allocated metal area. We refer to these wires as *B-Wires*. In addition to this base 64-bit interconnect, there are at least three other wire implementations that are potentially beneficial:

- *P-Wires*: Wires that are power-optimal. The wires have longer delays as they employ small repeater size and wide repeater spacing.

- *W-Wires*: Wires that are bandwidth-optimal. The wires have minimum width and spacing and have longer delays.

- *L-Wires*: Wires that are latency-optimal. These wires employ very wide wires and have low bandwidth.

To limit the range of possibilities, *P-Wires* and *W-Wires* can be combined to form a single wire implementation *PW-Wires*, that have poor delay characteristics, but allow low power and high bandwidth. While a traditional architecture would employ the entire available metal area for B-Wires, we propose the design of a heterogeneous interconnect, where part of the available metal area is employed for B-Wires, part for L-Wires, and part for PW-Wires. Thus, any data transfer has the option of using one of three sets of wires to effect the communication. Figure 1 demonstrates the differences between the wire implementations. In the next section, we will demonstrate how these options can be exploited to improve performance and reduce power consumption. If the mapping of data to a set of wires is straightforward, the logic overhead for the decision process is likely to be minimal. This issue will be treated in more detail in subsequent sections.

3. Optimizing Coherence Traffic

The previous section outlines wire implementation options available to an architect. For each cache coherence protocol, there exist myriad coherence operations with varying bandwidth and latency needs. Because of this diversity, there are numerous opportunities to improve performance and power characteristics by employing a heterogeneous interconnect. The goal of this section is to present a comprehensive listing of such opportunities. In Section 3.1 we focus on protocol-specific optimizations. We then discuss a variety of protocol-independent techniques in Section 3.2. Finally, we discuss the implementation complexity of the various techniques in Section 3.3.

3.1. Protocol-dependent Techniques

We begin by examining the characteristics of coherence operations in both directory-based and snooping bus-based coherence protocols. We then describe how these coherence operations can be mapped to the appropriate set of wires. In a bus-based design, the ability of a cache to directly respond to another cache's request leads to low L1 cache-to-cache miss latencies. L2 cache latencies are relatively higher as a processor core has to acquire the bus before sending the request to L2. It is difficult to support a large number of processor cores with a single bus due to the bandwidth and electrical limits of a centralized bus [11]. In a directory-based design [14, 25], each L1 connects to the L2 cache through a point-to-point link. This design has low L2 hit latency and scales better. However, each L1 cache-to-cache miss must be forwarded by the L2 cache, which implies high L1 cache-to-cache latencies. The performance comparison between these two design choices depends on the cache size, miss rate,

number of outstanding memory requests, working-set size, sharing behavior of the targeted benchmarks, etc. Since either option may be attractive to chip manufacturers, we will consider both forms of coherence protocols in our study.

Write-Invalidate Directory-based Protocol

Write-invalidate directory-based protocols have been implemented in existing dual-core CMPs [32] and will likely be used in larger scale CMPs as well. In a directory-based protocol, every cache line has a directory where the states of the block in all L1s are stored. Whenever a request misses in an L1 cache, a coherence message is sent to the directory at the L2 to check the cache line's global state. If there is a clean copy in the L2 and the request is a READ, it is served by the L2 cache. Otherwise, another L1 must hold an exclusive copy and the READ request is forwarded to the exclusive owner, which supplies the data. For a WRITE request, if any other L1 caches hold a copy of the cache line, coherence messages are sent to each of them requesting that they invalidate their copies. When each of these invalidation requests is acknowledged, the L2 cache can supply an exclusive copy of the cache line to the requesting L1 cache.

Hop imbalance is quite common in a directory-based protocol. To exploit this imbalance, we can send critical messages on fast wires to increase performance and send non-critical messages on slow wires to save power. For the sake of this discussion, we assume that the hop latencies of different wires are in the following ratio: L-wire : B-wire : PW-wire :: 1 : 2 : 3

Proposal I: *Read exclusive request for block in shared state*

In this case, the L2 cache's copy is clean, so it provides the data to the requesting L1 and invalidates all shared copies. When the requesting L1 receives the reply message from the L2, it collects invalidation acknowledgment messages from the other L1s before returning the data to the processor core[1]. Figure 2 depicts all generated messages.

The reply message from the L2 takes only one hop, while the invalidation acknowledgment messages take two hops – an example of hop imbalance. Since there is no benefit to receiving the cache line early, latencies for each hop can be chosen that equalize communication latency for the cache line and the acknowledgment messages. Acknowledgment messages include identifiers so they can be matched against the outstanding request in the L1's MSHR. Since there are only a few outstanding requests in the system, the identifier requires few bits, allowing the acknowledgment to be transferred on low-bandwidth low-latency L-Wires. Simultaneously, the data block transmission from the L2 can happen on low-power PW-Wires and still finish before the arrival of the acknowledgments. This strategy improves performance (because acknowledgments are often on the critical path) and reduces power consumption (because the data block is now transferred on power-efficient wires). While circuit designers have frequently employed different types of wires within a circuit to reduce power dissipation without extending the critical path, the proposals in this paper represent some of the first attempts to exploit wire properties at the architectural level.

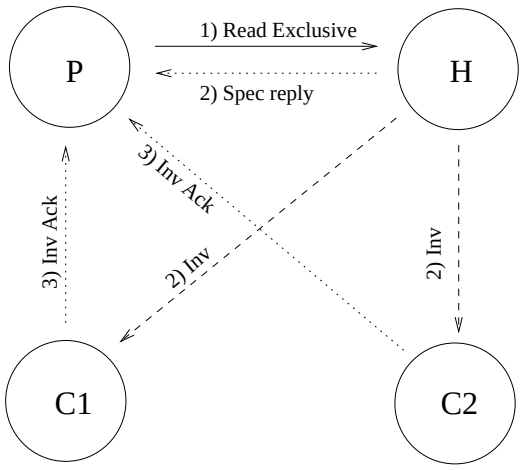


Figure 2. Read exclusive request for a block in shared state

Proposal II: *Read request for block in exclusive state*

In this case, the value in the L2 is likely to be stale and the following protocol actions are taken. The L2 cache sends a speculative data reply to the requesting L1 and forwards the read request as an intervention message to the exclusive owner. If the cache copy in the exclusive owner is clean, an acknowledgment message is sent to the requesting L1, indicating that the speculative data reply from the L2 is valid. If the cache copy is dirty, a response message with the latest data is sent to the

[1]Some coherence protocols may not impose all of these constraints, thereby deviating from a sequentially consistent memory model.

requesting L1 and a write-back message is sent to the L2. Since the requesting L1 cannot proceed until it receives a message from the exclusive owner, the speculative data reply from the L2 (a single hop transfer) can be sent on slower PW-Wires. The forwarded request to the exclusive owner is on the critical path, but includes the block address. It is therefore not eligible for transfer on L-Wires. If the owner's copy is in the exclusive clean state, a low-bandwidth acknowledgment to the requestor can be sent on L-Wires. If the owner's copy is dirty, the cache block can be sent over B-Wires, while the low priority writeback to the L2 can happen on PW-Wires. With the above mapping, we accelerate the critical path by using faster L-Wires, while also lowering power consumption by sending non-critical data on PW-Wires. The above protocol actions apply even in the case when a read-exclusive request is made for a block in the exclusive state.

Proposal III: *NACK messages*
When the directory state is busy, incoming requests are often NACKed by the home directory, i.e., a negative acknowledgment is sent to the requester rather than buffering the request. Typically the requesting cache controller reissues the request and the request is serialized in the order in which it is actually accepted by the directory. A NACK message can be matched by comparing the request id (MSHR index) rather than the full address, so a NACK is eligible for transfer on low-bandwidth L-Wires. When network contention is low, the home node should be able to serve the request when it arrives again, in which case sending the NACK on fast L-Wires can improve performance. In contrast, when network contention is high, frequent backoff-and-retry cycles are experienced. In this case, fast NACKs only increase traffic levels without providing any performance benefit. In order to save power, NACKs can be sent on PW-Wires.

Write-Invalidate Bus-Based Protocol
We next examine techniques that apply to bus-based snooping protocols. The role of the L1s and the L2 in a bus-based CMP system are very similar to that of the L2s and memory in a bus-based SMP (symmetric multiprocessor) system.

Proposal IV: *Signal wires*
Three wired-OR signals are typically used to avoid involving the lower/slower memory hierarchy [15]. Two of these signals are responsible for reporting the state of snoop results and the third indicates that the snoop result is valid. The first signal is asserted when any L1 cache, besides the requester, has a copy of the block. The second signal is asserted if any cache has the block in the exclusive state. The third signal is an inhibit signal, asserted until all caches have completed their snoop operations. When the third signal is asserted, the requesting L1 and the L2 can safely examine the other two signals. Since all of these signals are on the critical path, implementing them using low-latency L-Wires can improve performance.

Proposal V: *Voting wires*
Another design choice is whether to use cache-to-cache transfers if the data is in the shared state in a cache. The Silicon Graphics Challenge [17] and the Sun Enterprise use cache-to-cache transfers only for data in the modified state, in which case there is a single supplier. On the other hand, in the full Illinois MESI protocol, a block can be preferentially retrieved from another cache rather than from memory. However, when multiple caches share a copy, a "voting" mechanism is required to decide which cache will supply the data, and this voting mechanism can benefit from the use of low latency wires.

3.2. Protocol-independent Techniques

Proposal VI: *Narrow Bit-Width Operands for Synchronization Variables*
Synchronization is one of the most important factors in the performance of a parallel application. Synchronization is not only often on the critical path, but it also contributes a large percentage (up to 40%) of coherence misses [27]. Locks and barriers are the two most widely used synchronization constructs. Both of them use small integers to implement mutual exclusion. Locks often toggle the synchronization variable between zero and one, while barriers often linearly increase a barrier variable from zero to the number of processors taking part in the barrier operation. Such data transfers have limited bandwidth needs and can benefit from using L-Wires.

This optimization can be further extended by examining the general problem of cache line compaction. For example, if a cache line is comprised mostly of 0 bits, trivial data compaction algorithms may reduce the bandwidth needs of the cache line, allowing it to be transferred on L-Wires instead of B-Wires. If the wire latency difference between the two wire implementations is greater than the delay of the compaction/de-

compaction algorithm, performance improvements are possible.

Proposal VII: *Assigning Writeback Data to PW-Wires*
Writeback data transfers result from cache replacements or external request/intervention messages. Since writeback messages are rarely on the critical path, assigning them to PW-Wires can save power without incurring significant performance penalties.

Proposal VIII: *Assigning Narrow Messages to L-Wires*
Coherence messages that include the data block address or the data block itself are many bytes wide. However, many other messages, such as acknowledgments and NACKs, do not include the address or data block and only contain control information (source/destination, message type, MSHR id, etc.). Such narrow messages can be assigned to low latency L-Wires.

3.3. Implementation Complexity

In a conventional multiprocessor interconnect, a subset of wires are employed for addresses, a subset for data, and a subset for control signals. Every bit of communication is mapped to a unique wire. When employing a heterogeneous interconnect, a communication bit can map to multiple wires. For example, data returned by the L2 in respose to a read-exclusive request may map to B-Wires or PW-Wires depending on whether there are other sharers for that block (**Proposal I**). Thus, every wire must be associated with a multiplexor and de-multiplexor.

The decision process in selecting the right set of wires is minimal. For example, in **Proposal I**, an OR function on the directory state for that block is enough to select either B- or PW-Wires. In **Proposal II**, the decision process involves a check to determine if the block is in the exclusive state. To support **Proposal III**, we need a mechanism that tracks the level of congestion in the network (for example, the number of buffered outstanding messages). There is no decision process involved for **Proposals IV**, **V**, and **VII**. **Proposals VI** and **VIII** require logic to compute the width of an operand, similar to logic used in the PowerPC 603 [18] to determine the latency of integer multiply.

Cache coherence protocols are already designed to be robust in the face of variable delays for different messages. In all proposed innovations, a data packet is not distributed across different sets of wires. Therefore, different components of an entity do not arrive at different periods of time, thereby eliminating any timing problems. It may be worth considering sending the critical word of a cache line on L-Wires and the rest of the cache line on PW-Wires. Such a proposal may entail non-trivial complexity to handle corner cases and is not discussed further in this paper.

In a snooping bus-based coherence protocol, transactions are serialized by the order in which addresses appear on the bus. None of our proposed innovations for snooping protocols affect the transmission of address bits (address bits are always transmitted on B-Wires), so the transaction serialization model is preserved.

The use of heterogeneous interconnects does not imply an increase in metal area. Rather, we advocate that the available metal area be partitioned among different wire implementations.

4. Results

4.1. Methodology

The evaluation is performed with a detailed CMP architecture simulator based on UVSIM [34], a cycle-accurate execution-driven simulator. The CMP cores are out-of-order superscalar processors with private L1 caches, shared L2 cache and all lower level memory hierarchy components. Contention for memory hierarchy resources (ports, banks, buffers, etc.) are modeled in detail. In order to model an aggressive future generation CMP, we assume 16 processor cores, connected by a two-level tree interconnect. The simulated on-chip interconnect is based on SGI's NUMALink-4. A set of four processor cores is connected through a crossbar router, allowing low-latency communication to neighboring cores. As shown in Figure 3, the crossbar routers are connected to the root router, where the centralized L2 lies. We do not model contention within the routers, but do model port contention on the network interfaces. Each cache line in the L2 cache has a directory which saves sharing information for the processor cores. Every L1 cache miss is sent to the L2 cache, where further actions are taken based on the directory state. We model the directory-based cache coherence protocol that is employed in the SGI Origin 3000 [31]. It is an MESI write-invalidate protocol with

Parameter	Value
Processor	4-issue, 48-entry active list, 2GHz
L1 I-cache	2-way, 64KB, 64B lines, 1-cycle lat.
L1 D-cache	2-way, 64KB, 64B lines, 2-cycle lat.
On-chip Network	10 processor cycles per hop
L2 cache	4-way, 8MB, 64B lines, 4-cycle bank-lat.
MSHR per CPU	16
DRAM	16 16-bit-data DDR channels

Table 1. System configuration.

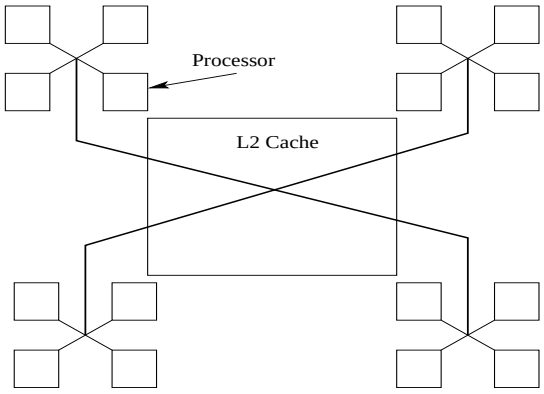


Figure 3. Interconnect Topology

migratory optimization. The migratory optimization causes a read (shared) request to return exclusive ownership if the requested cache line is in the UNOWN state. Important simulation parameters are listed in Table 1.

To test our ideas, we employ a workload consisting of all programs from the SPLASH-2 [33] and NAS parallel benchmark [4] suites that were compatible with our simulator. The programs were run to completion, but all experimental results reported in this paper are for the parallel phase of these applications and employ the default input sets for SPLASH-2 and the S-class for NAS.

4.2. Preliminary Results

While Section 3 provides a comprehensive list of potential optimizations, our preliminary study only examines the effect of one class of optimizations on a directory-based cache coherence protocol. Our base model assumes a fat-tree structure with four children on each non-leaf node. Processor cores are on the leaf node, and four neighboring cores are grouped into a sub-tree. We define the hop number as the number of routers used to transfer a network message between two processor cores. As illustrated in Figure 3, it takes one network hop to transfer a message between two processors in the same domain (sub-tree), and three network hops to transfer a message between any two processors which belong to different domains (sub-trees). The latencies on the interconnects would depend greatly on the technology, processor layout, and available metal area. The estimation of some of these parameters is beyond the scope of this study. For the base case, we assume the metal layer is comprised entirely of B-Wires, and one hop latency is 10 processor cycles. This assumption is based on projections [2, 3] that claim on-chip wire delays of the order of tens of cycles.

In the base case, each processor can issue a single 64-bit packet every cycle that is transmitted on B-Wires. In our proposed heterogeneous interconnect, we again assume that each processor can issue a single packet every cycle, but that packet can be transmitted on one of three possible sets of wires. The transmission can either happen on a set of 64 B-Wires, a set of 64 PW-Wires, or a set of 16 L-Wires. While we have kept the packet throughput per processor constant in the base and proposed cases, the metal area cost of the heterogeneous interconnect is higher than that of the base case. The comparison of designs that consume equal metal area is part of future work. The experiments in this paper are intended to provide preliminary best-case estimates of the potential of a heterogeneous interconnect. Relative energy and delay estimates of wires have been derived in [6]. L-Wires (latency-optimal) have a latency of five cycles for each network hop and consume about 45% more dynamic energy than B-Wires. PW-Wires (low power and high bandwidth) have a latency of 15 cycles per hop and consume about half the dynamic energy of B-Wires. We assume that every interconnect is perfectly pipelined. It must be noted that L-Wires can yield lower latency than B-Wires even when sending messages larger than 16 bits (and lower than 112 bits). However, sending large messages through L-Wires will increase wait time for other messages, resulting in overall lower performance.

We will consider only the simplest subset of techniques proposed in Section 3, that entail the least complexity in mapping critical data transfers to L-Wires and non-critical data transfers to PW-Wires. Firstly, not all critical messages are eligible for transfer on L-

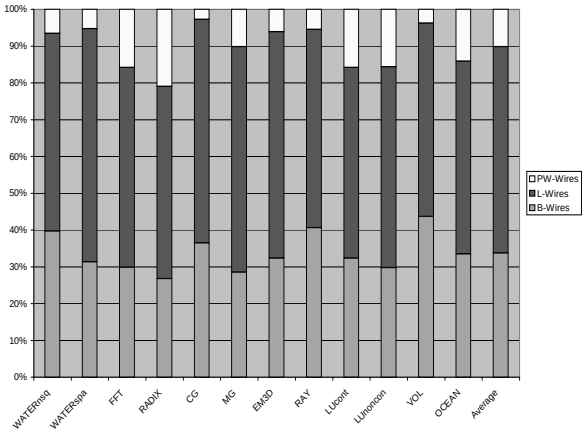


Figure 4. Percentage of critical and non-critical messages

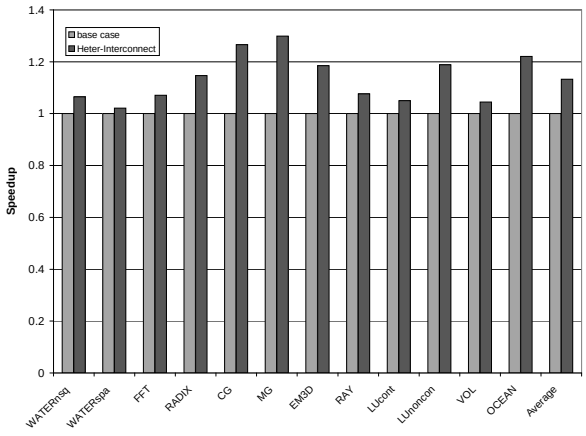


Figure 5. Performance improvements

Wires. Since we assume 16 L-Wires and 64-bit addresses, a message that includes the full address is always sent on B-Wires or PW-Wires. To identify what messages can be sent through B-Wires and PW-Wires, we classify all coherence messages into six categories: REQUEST, WRITE, PROBE, REPLY2MD, RESPONSE, and REPLY2PI.

Every memory transaction that misses in the local L1 cache will send out a REQUEST message to the L2 cache. A REQUEST message includes request type (READ, RDEXC, UPGRADE, etc.), source id, MSHR id, and data address. Although REQUEST messages are mostly on the critical path, they are too wide to benefit from L-Wires. Therefore, REQUEST messages are always transmitted on B-Wires.

WRITE messages result from L1 cache replacement. WRITE messages are often not on the critical path and they can be always sent on PW-Wires without degrading performance.

PROBE messages happen in cache-to-cache misses and can be further classified as INTERVENTION messages and INVALIDATE messages. An INTERVENTION request retrieves the most recent data for a line from an exclusive copy that may exist within a remote cache. An INVALIDATE request removes copies of a cache line that may exist in remote caches. Both INTERVENTION and INVALIDATE messages include request type, source id, address, and the MSHR id of the REQUEST message that generated the PROBE message. PROBE messages are usually critical, but can only be sent through B-Wires due to the bandwidth limitation of L-Wires.

When a processor receives a PROBE message, it sends a REPLY2MD message to the home directory (if it has a dirty copy of the cache line) and a RESPONSE message to the processor that generated the REQUEST message. Response messages can have data, in which case, they must be sent on B-Wires. In most cases, the message is simply an acknowledgment that only needs reply type (4 bits), source id (4 bits in 16-core CMP), and MSHR id (4 bits), and can be sent on L-Wires. REPLY2MD is often off the critical path and can be sent on PW-Wires.

REPLY2PI messages are the messages sent from the home directory to the requester. Some REPLY2PI messages include data, while others only include control bits (such as NACK or the reply for an UPGRADE request). REPLY2PI messages not only have variable bandwidth needs, but also have variable criticality, as discussed in Section 3. For example, in cache-to-cache misses, REPLY2PI messages tend to arrive at the requester faster than the PROBE/RESPONSE messages and are therefore off the critical path.

In order to simplify the decision process in mapping data to wires, we adopt the following policy: (i) WRITE and REPLY2MD messages are always sent on PW-Wires, (ii) all other messages that are narrower than 16 bits are sent on L-Wires, and (iii) all other messages that are wider than 16 bits are sent on B-Wires. Thus, we are only incorporating **Proposal VII**, **Proposal VIII**, and parts of **Proposal III** in our simulation model. Detailed evaluations of other proposals are left for future work.

Figure 4 shows the percentage of messages sent

through different sets of wires, while assuming the allocation policy described above. It must be noted that a significant fraction of all messages are narrow enough that they can be sent on L-Wires. Messages sent on B- and PW-Wires are wider than messages sent on L-Wires. As a result, the fraction of bits transmitted on L-Wires is lower than that indicated in Figure 4. If we assume that dynamic energy consumed by transmissions on B, L, and PW-Wires are in the ratio 1: 1.45: 0.52 (as estimated in a prior study [6]), interconnect dynamic energy in the proposed design is reduced by 40%. Further, this reduction in interconnect energy is accompanied by improvements in performance. Figure 5 shows the performance speedup achieved by the transmission of some signals on low-latency L-Wires. The overall average improvement across the benchmark set is 13.3%.

5. Related Work

Beckmann *et al.* [9] address the problem of long L2 cache access times in a chip multiprocessor by employing low latency, low bandwidth transmission lines. They utilize transmission lines to send data from the center of the L2 cache to different banks. Kim *et al.* [21] proposed a dynamic non-uniform cache access (DNUCA) mechanism to accelerate cache access. Our proposal is orthogonal to the above technique and can be combined with non-uniform cache access mechanisms to improve performance.

A recent study by Citron *et al.* [13] examines entropy within data being transmitted on wires and identifies opportunities for compression. Unlike the proposed technique, they employ a single interconnect to transfer all data. Balasubramonian *et al.* [6] utilize heterogeneous interconnects for the transmission of register and load/store values to improve energy-delay characteristics in a partitioned microarchitecture.

Recent studies [20, 24, 26, 29] have proposed several protocol optimizations that can benefit from heterogeneous interconnects. For example, in the *Dynamic Self Invalidation* scheme proposed by Lebeck *et al.* [26], the self-invalidate [24, 26] messages can be effected through power-efficient PW-Wires. In a processor model implementing token coherence, the low-bandwidth token messages [29] are often on the critical path and thus, can be effected on L-Wires. A recent study by Huh *et al.* [20] reduces the frequency of false sharing by employing incoherent data. For cache lines suffering from false sharing, only the sharing states need to be propagated and such messages are a good match for low-bandwidth L-Wires.

6. Conclusions and Future Work

Coherence traffic in a chip multiprocessor has diverse needs. Some messages can tolerate long latencies, while others are on the program critical path. Further, messages have varied bandwidth demands. On-chip global wires can be designed to optimize latency, bandwidth, or power. We advocate partitioning available metal area across different wire implementations and intelligently mapping data to the set of wires best suited for its communication. This paper presents numerous novel techniques that can exploit a heterogeneous interconnect to simultaneously improve performance and reduce power consumption.

Our preliminary evaluation of a subset of the proposed techniques shows that a large fraction of messages have low bandwidth needs and can be transmitted on low latency wires, thereby yielding a performance improvement of 13%. At the same time, a 40% reduction in interconnect dynamic energy is observed by transmitting non-critical data on power-efficient wires. These improvements are achieved at a marginal complexity cost as the mapping of messages to wires is extremely straightforward.

For future work, we plan to strengthen our evaluations by comparing processor models with equal metal area. We will carry out a sensitivity analysis with respect to important processor parameters such as latencies, interconnect topologies, etc. We will also evaluate the potential of other techniques listed in this paper.

References

[1] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. The Use of Prediction for Accelerating Upgrade Misses in CC-NUMA Multiprocessors. In *Proceedings of PACT-11*, 2002.

[2] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of ISCA-27*, pages 248–259, June 2000.

[3] S. I. Association. International Technology Roadmap for Semiconductors 2003. http://public.itrs.net/Files/2003ITRS/Home2003.htm.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1994.

[5] H. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI*. Addison-Wesley, 1990.

[6] R. Balasubramonian, N. Muralimanohar, K. Ramani, and V. Venkatachalapathy. Microarchitectural Wire Management for Performance and Power in Partitioned Architectures. In *Proceedings of HPCA-11*, February 2005.

[7] K. Banerjee and A. Mehrotra. A Power-optimal Repeater Insertion Methodology for Global Interconnects in Nanometer Designs. *IEEE Transactions on Electron Devices*, 49(11):2001–2007, November 2002.

[8] B. Beckmann and D. Wood. TLC: Transmission Line Caches. In *Proceedings of MICRO-36*, December 2003.

[9] B. Beckmann and D. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *Proceedings of MICRO-37*, December 2004.

[10] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood. Multicast Snooping: A New Coherence Method using a Multicast Address Network. *SIGARCH Comput. Archit. News*, pages 294–304, 1999.

[11] F. A. Briggs, M. Cekleov, K. Creta, M. Khare, S. Kulick, A. Kumar, L. P. Looi, C. Natarajan, S. Radhakrishnan, and L. Rankin. Intel 870: A building block for cost-effective, scalable servers. *IEEE Micro*, 22(2):36–47, 2002.

[12] R. Chang, N. Talwalkar, C. Yue, and S. Wong. Near Speed-of-Light Signaling Over On-Chip Electrical Interconnects. *IEEE Journal of Solid-State Circuits*, 38(5):834–838, May 2003.

[13] D. Citron. Exploiting Low Entropy to Reduce Wire Delay. *IEEE Computer Architecture Letters*, vol.2, January 2004.

[14] Corporate Institute of Electrical and Electronics Engineers, Inc. Staff. *IEEE Standard for Scalable Coherent Interface, Science: IEEE Std. 1596-1992*. 1993.

[15] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: a Hardware/software Approach*. Morgan Kaufmann Publishers, Inc, 1999.

[16] W. Dally and J. Poulton. *Digital System Engineering*. Cambridge University Press, Cambridge, UK, 1998.

[17] M. Galles and E. Williams. Performance Optimizations, Implementation, and Verifi cation of the SGI Challenge Multiprocessor. In *HICSS (1)*, pages 134–143, 1994.

[18] G. Gerosa and et al. A 2.2 W, 80 MHz Superscalar RISC Microprocessor. *IEEE Journal of Solid-State Circuits*, 29(12):1440–1454, December 1994.

[19] R. Ho, K. Mai, and M. Horowitz. The Future of Wires. *Proceedings of the IEEE*, Vol.89, No.4, April 2001.

[20] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence Decoupling: Making Use of Incoherence. In *Proceedings of ASPLOS-XI*, pages 97–106, 2004.

[21] J. Kim, M. Taylor, J. Miller, and D. Wentzlaff. Energy Characterization of a Tiled Architecture Processor with On-Chip Networks. In *Proceedings of ISLPED*, pages 424–427, 2003.

[22] K. Krewell. UltraSPARC IV Mirrors Predecessor: Sun Builds Dualcore Chip in 130nm. *Microprocessor Report*, pages 1,5–6, Nov. 2003.

[23] A.-C. Lai and B. Falsafi . Memory Sharing Predictor: The Key to a Speculative Coherent DSM. In *Proceedings of ISCA-26*, 1999.

[24] A.-C. Lai and B. Falsafi . Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction. In *Proceedings of ISCA-27*, pages 139–148, 2000.

[25] J. Laudon and D. Lenoski. The SGI Origin: A cc-NUMA Highly Scalable Server. In *Proceedings of ISCA-24*, pages 241–251, June 1997.

[26] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proceedings of ISCA-22*, pages 48–59, 1995.

[27] K. M. Lepak and M. H. Lipasti. Temporally Silent Stores. In *Proceedings of ASPLOS-X*, pages 30–41, 2002.

[28] N. Magen, A. Kolodny, U. Weiser, and N. Shamir. Interconnect Power Dissipation in a Microprocessor. In *Proceedings of System Level Interconnect Prediction*, February 2004.

[29] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of ISCA-30*, 2003.

[30] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. In *Proceedings of ASPLOS-VII*, October 1996.

[31] Silicon Graphics, Inc. *SGI™Origin™3000 Series Technical Report*, Jan 2001.

[32] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. Technical report, IBM Server Group Whitepaper, October 2001.

[33] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of ISCA-22*, pages 24–36, June 1995.

[34] L. Zhang. UVSIM Reference Manual. Technical Report UUCS-03-011, University of Utah, May 2003.

Reducing the Power and Complexity of Path-Based Neural Branch Prediction

Gabriel H. Loh
College of Computing
Georgia Institute of Technology
loh@cc.gatech.edu

Daniel A. Jiménez
Department of Computer Science
Rutgers University
djimenez@cs.rutgers.edu

Abstract

A conventional path-based neural predictor (PBNP) achieves very high prediction accuracy, but its very deeply pipelined implementation makes it both a complex and power-intensive component. One of the major reasons for the large complexity and power is that for a history length of h, the PBNP must use h separately indexed SRAM arrays (or suffer from a very long update latency) organized in an h-stage predictor pipeline. Each pipeline stage requires a separate row-decoder for the corresponding SRAM array, inter-stage latches, control logic, and checkpointing support. All of these add power and complexity to the predictor.

We propose two techniques to address this problem. The first is modulo path-history *which decouples the branch outcome history length from the path history length allowing for a shorter path history (and therefore fewer predictor pipeline stages) while simultaneously making use of a traditional long branch outcome history. The pipeline length reduction results in decreased power and implementation complexity. The second technique is* bias-based filtering *(BBF) which takes advantage of the fact that neural predictors already have a way to track strongly biased branches. BBF uses the bias weights to filter out mostly always taken or mostly always not-taken branches and avoids consuming update power for such branches.*

Our proposal is complexity effective because it decreases the power and complexity of the PBNP without negatively impacting performance. The combination of modulo path-history and BBF results in a slight improvement in predictor accuracy of 1% for 32KB and 64KB predictors, but more importantly the techniques reduce power and complexity by reducing the number of SRAM arrays from 30+ down to only 4-6 tables, and reducing predictor update activity by 4-5%.

1. Introduction

After decades of academic and industrial research efforts focused on the branch prediction problem, pipeline flushes due to control flow mispredictions remain one of the primary bottlenecks in the performance of modern processors. A large amount of recent branch prediction research has centered around techniques inspired and derived from machine learning theory, with a particular emphasis on the *perceptron* algorithm [3, 4, 7–10, 14, 18]. These neural-based algorithms have been very successful in pushing the envelope of branch predictor accuracy.

Researchers have made a conscious effort to propose branch predictors that are highly amenable to pipelined and ahead-pipelined organizations to minimize the impact of predictor latency on performance. There has been considerably less effort on addressing power consumption and implementation complexity of the neural predictors. Reducing branch predictor power is not an easy problem because any reduction in the branch prediction accuracy can result in an overall increase in the *system* power consumption due to a corresponding increase in wrong-path instructions. On the other hand, peak power consumption, which limits the processor performance, and average power consumption, which impacts battery lifetime for mobile processors, are important design concerns for future processors [5]. Furthermore, it has been shown that the branch predictor, and the fetch engine in general, is a thermal hot-spot that can potentially limit the maximum clock frequency and operating voltage of the CPU, which in turn limits performance [16].

This paper focuses on the *path-based neural predictor* which is one of the proposed implementations of neural branch prediction [7]. In particular, this algorithm is highly accurate and pipelined for low effective access latency. We explain the organization of the predictor and the major sources of power consumption and implementation complexity. We propose a new technique for managing branch path-history information that greatly reduces the

number of tables, the pipeline depth, and the checkpointing overhead required for path-based neural prediction. We also propose a simple bias-based filtering mechanism to further reduce branch prediction power. While this paper specifically discusses the original path-based neural predictor [7], the techniques are general and can be easily applied to other neural predictors that use path history.

The rest of this paper is organized as follows. Section 2 provides an overview of the path-based neural predictor and discusses its power and complexity. Section 3 explains our proposed techniques for reducing the power consumption and implementation complexity. Section 4 presents the simulation-based results of our optimized path-based neural predictor in terms of the impact on prediction accuracy and power reduction. Section 5 concludes the paper.

2. Path-Based Neural Prediction

This section describes the original path-based neural predictor (PBNP), and then details the power and complexity issues associated with the PBNP.

2.1. Predictor Organization

The path-based neural predictor (PBNP) derives from the original *perceptron* branch predictor [9]. We define a vector $\overrightarrow{x} = \langle 1, x_1, x_2, ..., x_h \rangle$ where x_i is the ith most recent branch history outcome represented as -1 for a not taken branch and 1 for a taken branch. The branch history is the collection of taken/not-taken results for the h most recent conditional branches. The perceptron uses the branch address to select a set of weights $\overrightarrow{w} = \langle w_0, w_1, ..., w_h \rangle$ that represent the observed correlation between branch history bits and past branch outcomes. The sign of the dot-product of $\overrightarrow{w} \cdot \overrightarrow{x}$ provides the final prediction where a positive value indicates a taken-branch prediction. Figure 1a shows a block diagram of the lookup logic for the perceptron predictor.

At a high-level, the PBNP is very similar to the perceptron in that it computes a dot-product between a vector of weights and the branch history. The primary difference is that the PBNP uses a different branch address for each of the weights of $\overrightarrow{w}$. Let PC_0 be the current branch address, and PC_i be the ith most recent branch address in the *path history*. For the perceptron, each weight is chosen with the same index based on PC_0. For the PBNP, each weight w_i is chosen based on an index derived from PC_i. This provides path history information that can improve prediction accuracy, and spreading out the weights in different entries also helps to reduce the impact of inter-branch aliasing.

To implement the PBNP, the lookup phase is actually pipelined over many stages based on the overall path-/branch-history length. Figure 1b illustrates the hardware organization of the PBNP. For a branch at cycle t, the PBNP starts the prediction at cycle $t - h$ using PC_h. For each cycle after $t - h$, the PBNP computes partial sums of the dot-product of $\overrightarrow{w} \cdot \overrightarrow{x}$. Pipeline stage i contains the partial sum for the branch prediction that will be needed in i cycles. At the very end of the pipeline, the critical lookup latency consists of looking up the final weight and performing the final addition.

2.2. Power and Complexity

During the lookup phase of the PBNP, each pipeline stage reads a weight corresponding to the exact same PC. This is due to the fact that the current PC_0 will be next cycle's PC_1 and next-next cycle's PC_2 and so on. This allows an implementation where the weights are read in a single access using a single large SRAM row that contains all of the weights. During the update phase however, a single large access would force the update process to use a pipelined implementation as well. While at first glance this may seem desirable, this introduces considerable delay between update and lookup. For example a 30-stage update pipeline implies that even after a branch outcome has been determined, another 30 cycles must elapse before the PBNP has been fully updated to reflect this new information. This update delay can create a decrease in predictor accuracy. There are also some timing effects due to the fact that some weights of a branch will be updated before others.

An alternative organization uses h tables in parallel, one for each pipeline stage/history-bit position [7], as illustrated in Figure 1b. This organization allows for a much faster update and better resulting accuracy and performance. The disadvantage of this organization is that there is now a considerable amount of area and power overhead to implement the row decoders for the h separate SRAM arrays. Furthermore, to support concurrent lookup and update of the predictor, each of these SRAM arrays needs to be dual-ported (one read port/one write port) which further increases the area and power overhead of the SRAM row decoders. To use the PBNP, the branch predictor designer must choose between an increase in power and area or a decrease in prediction accuracy.

On a branch misprediction, the PBNP pipeline must be reset to the state that corresponded to the mispredicting branch being the most recent branch in the branch

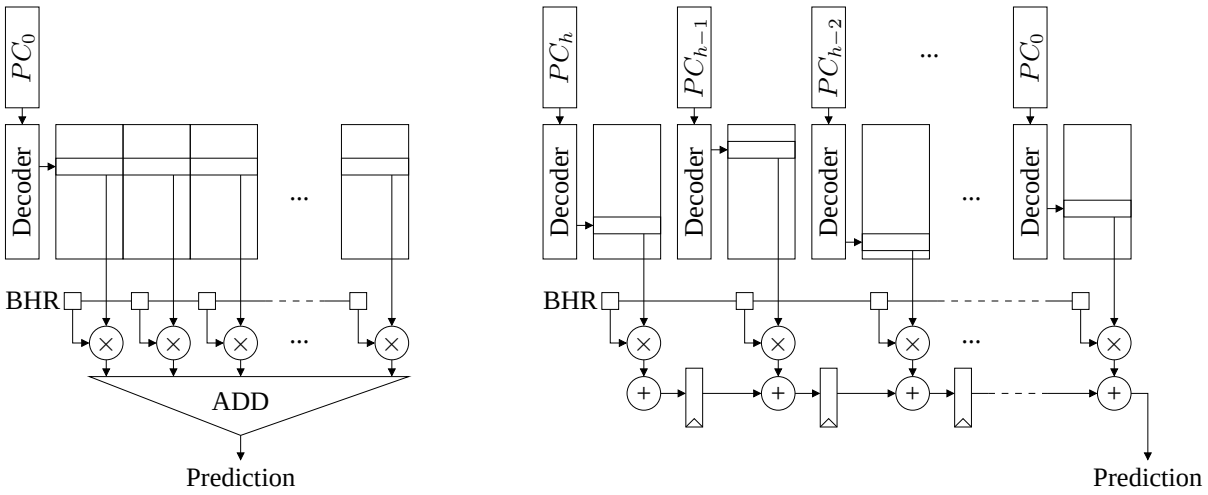


Figure 1: (a) Organization of the lookup logic for the perceptron branch predictor. (b) Lookup logic for the pipelined path-based neural branch predictor.

and path history. To support this predictor state recovery, each branch must checkpoint all of the partial sums in the PBNP pipeline. On a branch misprediction, the PBNP restores all of the partial sums in the pipeline using this checkpointed state. For b-bit weights and a history length of h, a PBNP checkpoint requires approximately bh bits of storage. The total number of bits is slightly greater because the number of bits required to store a partial sum increases as the sum accumulates more weights. The total storage for all checkpoints corresponds to the maximum number of in-flight branches permitted in the processor. For example, assuming one branch occurs every five instructions, then a 128-entry ROB would on average have 25 branches in flight. This means the PBNP checkpoint table must have about 25 entries to support the average number of branches. To avoid stalls due to a burst of branch instructions, the checkpoint table may need to be substantially larger. For proposals of very-large effective instruction window processors such as CFP [17], the checkpointing overhead further increases.

The checkpointing overhead represents additional area, power, and state that is often unaccounted for in neural predictor studies. This overhead increases with the history/path-length of the predictor since the PBNP must store one partial sum per predictor stage. A further source of complexity is the additional control logic required to manage the deeply pipelined predictor.

3. Reducing Perceptron Power and Complexity

In this section, we propose two techniques for reducing the power and complexity of the path-based neural predictor. Modulo-Path History is a new way to manage path-history information which also provides a new degree of freedom in the design of neural predictors. Bias-Based Filtering is a technique similar to previously proposed filtering mechanisms that takes advantage of the information encoded in the neural weights to detect highly biased branches.

3.1. Modulo-Path History

In the original PBNP, the path history length is always equal to the branch history length. This is a result of using PC_i to compute the index for the weight of x_i. As described in the previous section, the pipeline depth directly increases the number of tables and the checkpointing overhead required. On the other hand, supporting a long history length requires the PBNP to be deeply pipelined.

We propose *modulo path-history* where we decouple the branch history length from the path history length. We limit the path history to only the $P < h$ most recent branch addresses. Instead of using PC_i to compute the index for w_i, we use $PC_{i \mod P}$. In this fashion, we can reduce the degree of pipelining down to only P stages. Figure 2a shows the logical predictor organiza-

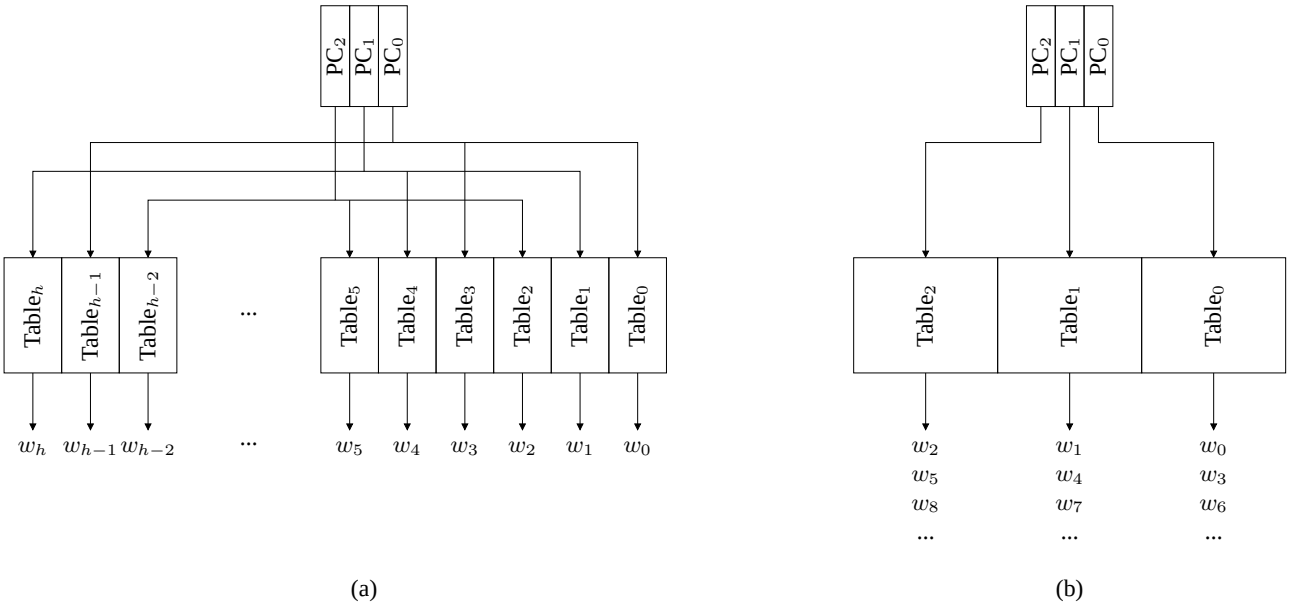


Figure 2: (a) Logical organization of a PBNP using modulo path-history. (b) Physical organization of a PBNP using modulo path-history for $P = 3$.

tion of a PBNP using modulo path-history (for $P = 3$). Since every Pth weight is indexed with the same branch address, we can interleave the order of the weights in the table such that only P tables are necessary. Figure 2b shows how each table provides weights that correspond to h/P branch history outcomes, where each branch history outcome is separated by P bit positions.

By reducing the PBNP implementation to only use P distinct tables, we address several of the main sources of power and complexity as described in Section 2. Using only P tables reduces the duplicated row-decoder overhead. The reduction in the number of tables reduces the overall pipeline depth of the predictor which reduces the amount of state that must be checkpointed (i.e. there are only P partial sums). With fewer stages, the control logic for the predictor pipeline can also be reduced. The number of inter-stage latches and associated clocking overhead is also correspondingly reduced.

Modulo path-history may also make the single-table implementation feasible. The pipelined update still adds latency to the update phase of the branch predictor, but the update latency has been reduced from $O(h)$ cycles down to $O(P)$ cycles. For sufficiently small values of P, the substantial reduction in complexity and associated power may justify the small increase in the update latency.

Modulo path-history is a unique way to manage the branch path history information. A PBNP can now choose between different lengths of branch and path history. Tarjan and Skadron proposed a comprehensive taxonomy of neural branch predictor organizations that can describe a very wide variety of neural predictor variations [18]. Nevertheless, modulo path-history is a new addition that does not fall into any of their categories.

3.2. Bias-Based Filtering

Earlier branch prediction studies have made the observation that there are a large number of branch instructions whose outcomes are almost always in the same direction [2, 6]. Some of this research has proposed various ways for detecting these strongly biased branches and removing or *filtering* them out to reduce the amount of interference in the branch prediction tables. We make the observation that from an energy and power perspective, keeping track of h distinct weights and performing an expensive dot-product operation is an overkill for these easy-to-predict branches. We also observe that the family of neural predictors have built-in mechanisms for detecting highly-biased branches. Combining these observations, we proposed *Bias-Based Filtering* (BBF).

The BBF technique is simple in principal. We consider a branch whose bias weight (w_0) has saturated (equal to maximum or minimum value) as a highly-biased branch. When the predictor detects such a branch, the prediction is determined only by the bias weight as opposed to the entire dot-product. If this prediction turns out to

be correct, the predictor skips the update phase which saves the associated power and energy. BBF does not reduce the lookup power because the pipelined organization must start the dot-product computation before the predictor knows whether the branch is highly biased. Besides the power reduction, BBF has a slight accuracy benefit because the act of filtering the strongly biased branches reduces the interference among the remaining branches.

The relatively long history lengths of neural predictors combined with the usage of multi-bit weights results in a table that has relatively few entries or rows. This greatly increases the amount of inter-branch aliasing in the tables which potentially reduces the effectiveness of BBF. To address this, we propose that the bias table uses a larger number of entries than any of the other tables. This makes sense since the bias table now has to keep track of all of the strongly biased branches as well as provide the bias weights for the regular branches.

To increase the number of strongly biased branches covered by BBF, we modify the neural prediction lookup slightly such that the bias weight (and only the bias weight) is indexed in a gshare fashion (xor of branch address and branch history). This improves the filtering mechanism by allowing the bias table to detect branches that are strongly biased but only under certain global history contexts. We also reduce the width of the bias weights to 5 bits which allows the bias weights to saturate more quickly and start filtering strongly biased branches sooner.

4. Performance and Power Results

In this section, we present the simulation results for an optimized PBNP predictor that uses modulo path-history and bias-based filtering.

4.1. Simulation Methodology

For our prediction accuracy results, we used the in-order branch predictor simulator sim-bpred from the SimpleScalar toolset [1]. We simulated all twelve SPECint applications using the reference sets and single 100M instruction simulation points chosen by SimPoint 2.0 [13]. Our applications were compiled on an Alpha 21264 with Compaq `cc` with full optimizations.

4.2. Impact of Modulo-Path History

To measure the impact of modulo-path history, we started with the original PBNP where there are $P = h$ tables that each provide a single weight corresponding to a single branch history position. We then increased the number of weights provided by each table by setting $P = h/2, h/3$, and so on. This reduces the length of the path-history while maintaining a constant branch history length. Figure 3 shows the impact on prediction accuracy as we vary the number of weights per lookup table over a range of predictor sizes. For the smaller predictors (2KB and 4KB), there is an initial increase in the misprediction rate when we add modulo path-history. For predictors 16KB and larger, the increase in the misprediction rate is less than 0.5% (8KB) and in some cases even *improve* prediction accuracy by a small amount (0.1% for 64KB).

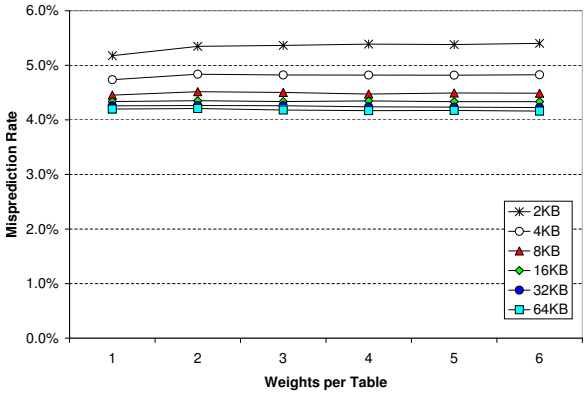


Figure 3: Average misprediction rates on SPECint when using modulo path-history.

As we increase the number of weights per table, the total number of tables decreases. This reduces the power and energy cost per access due to a reduction in the number of row decoders in the entire predictor. The power cost per table lookup increases with the number of weights per table, but the number of tables decreases. The modulo path-history approach for managing path history in the PBNP is overall performance-neutral while providing a power benefit by reducing the power consumed per lookup. We have not quantified the exact power benefit in Watts due to limitations of CACTI-like tools. We also have not quantified in detail the reduction of the checkpointing overhead or the impact of simplifying the control logic for the reduced pipeline depth, but simply observe that there will be some power and complexity benefit.

4.3. Impact of Bias-Based Filtering

For a fixed hardware budget, increasing the number of entries in the bias table forces the remaining tables to be decreased in size. We evaluated a range of table sizes. Figure 4 shows the prediction accuracy impact of dedicating some more weights to the bias table while reduc-

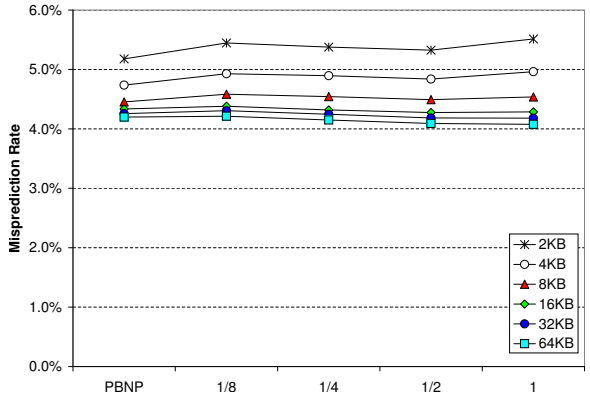


Figure 4: Average misprediction rates on SPECint when using bias-based filtering.

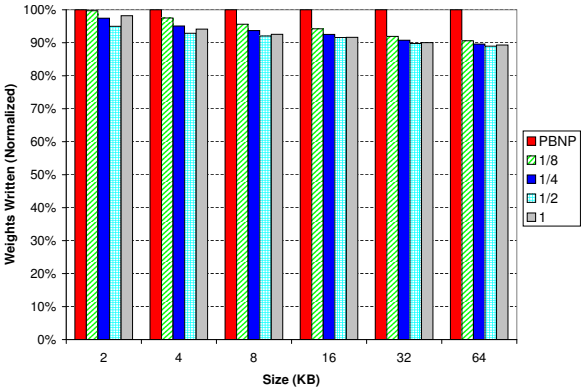


Figure 5: Average reduction in update activity for a PBNP using bias-based filtering.

ing the size of the other tables. These results include the Bias-Based Filtering effects. The left-most set of points in Figure 4 correspond to the baseline PBNP. The remaining configurations use BBF where "$1/n$" indicates that the bias table has X/n entries, where X is the number of bytes in the table. For example, the 1/4 configuration for an 8KB budget has a bias table with 8K/4 = 2K entries (*not* 2KB worth of entries). Similar to the modulo path-history results, BBF is less effective at the smallest predictor sizes, and is relatively performance-neutral at larger sizes. For predictors sized 16KB and greater, BBF actually results in a slight (1-3%) decrease in mispredictions. The reason for this slight accuracy benefit is that gating updates for highly-biased branches creates an interference-reducing effect similar to a partial update policy [12].

The primary purpose of BBF is to reduce the number of weights written to the tables during the update phase. Figure 5 shows the reduction in the number of weights written as compared to the baseline PBNP. Overall, the mid-sized to larger sized predictors achieve the greatest benefit, with about a 10% reduction in the update activity.

4.4. Impact of Power-Reduced Path-Based Neural Predictor

In the previous subsections, we have shown how the techniques of modulo path-history and bias-based filtering are relatively performance-neutral for mid-sized predictors and performance-beneficial for larger predictors. Simultaneously, these techniques provide a reduction in the predictor's power consumption by reducing the power per access and reducing the total number of accesses, and a reduction in the implementation complexity by reducing

	Baseline PBNP	With Mod. Path and BBF		
Predictor Size	History Length	History Length	Path Length	Bias Weights
2KB	17	17	4	1K
4KB	25	24	4	2K
8KB	31	29	4	4K
16KB	32	33	5	8K
32KB	42	42	3	16K
64KB	47	42	3	32K

Table 1: Parameters for the baseline PBNP and a PBNP using both modulo path-history and bias-based filtering.

the number of predictor pipeline stages. We now observe the effects of combining the two techniques. Table 1 lists the final configurations used for the PBNP with modulo path-history and bias-based filtering, as well as the baseline PBNP configurations.

Figure 6 shows the average misprediction rate for a conventional PBNP and a PBNP augmented with modulo path-history and bias-based filtering. Similar to the individual results, our techniques are not recommended for small predictor sizes. At 16KB the techniques do not help or hurt accuracy, and at 32KB and 64KB they provide a small accuracy benefit (about 1%).

As discussed earlier, the modulo path-history reduces power by reducing the number of tables and the reducing the pipeline depth of the predictor. BBF reduces the number of table updates. Figure 7 shows the relative decrease in update activity when compared to a conventional PBNP. Note that for the 2KB predictor size, the activity actually *increases*. This is due to the fact that for the smaller predictor, the slight decrease in prediction accuracy causes the neural prediction algorithm to train more frequently which causes more overall activity in the table

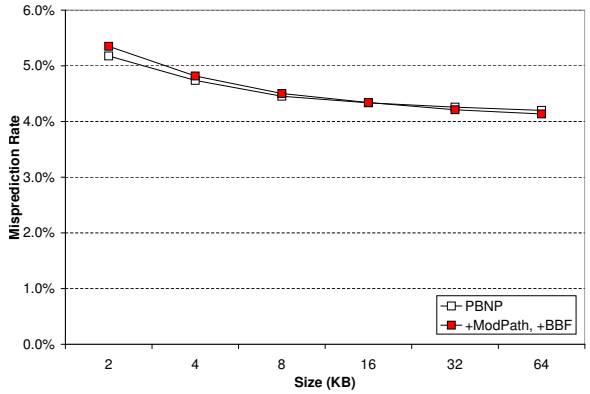


Figure 6: Average SPECint misprediction rate for the baseline PBNP and a PBNP using both modulo path-history and bias-based filtering.

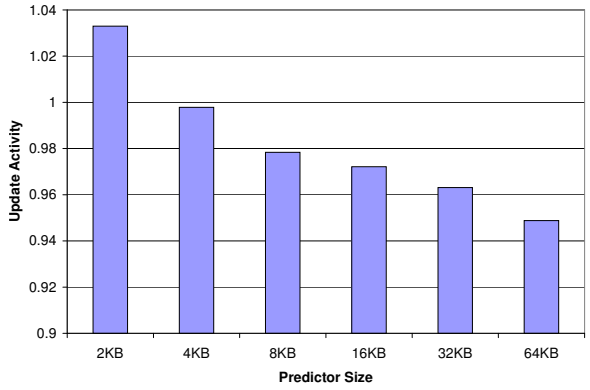


Figure 7: Change in the update activity of a PBNP using modulo path-history and bias-based filtering as compared to a conventional PBNP.

of weights. At the larger sizes, BBF reduces the frequency of updates by 4-5%.

4.5. Overall Impact on Implementation Complexity

The original goal of this research was to redesign a path-based neural predictor to be less complex and hence easier to implement. Table 2 summarizes the overall benefits in terms of key sources of implementation complexity. Modulo-path history reduces the number of separate SRAM arrays from 18-48 down to only 4-6. The reduction in the table count is directly correlated to the depth of pipelining needed to implement a PBNP. This 72-92% reduction of the predictor pipeline length greatly simplifies the control logic needed to control stalling, checkpointing and recovering the predictor pipeline.

The extra storage needed for checkpointing the predictor at every branch impacts the physical design of the predictor. Table 2 lists the number of bits required per checkpoint for copying the partial sums. The bit counts are slightly underestimated because we assumed that every stage only needs an 8-bit value to simplify the arithmetic. In practice the bit-width increases with the number of accumulated weights. The original PBNP needs 144-384 bits of information checkpointed on every branch, while using modulo-path history reduces this to only 32-48 bits per branch. The checkpointing overhead reduction not only reduces the size of the SRAM needed to store the checkpoints, but it also reduces the number of wires entering and leaving the predictor for the checkpoints. To reduce the impact on the physical layout and latency of the predictor, the checkpoint SRAM may be placed slightly further away from the main predictor. This physical separation requires longer wires (more capacitance) which results in increased power consumption for the communication between the predictor and the checkpoint SRAM.

Modulo path-history and BBF impact the power consumed by the predictor itself and also the overall system-wide power. In this study, we did not quantify the exact power benefits because SRAM latency/power estimation tools such as CACTI [15] and eCACTI [11] do not handle the non-power-of-two SRAM sizes used in this study. For the non-SRAM portions of the predictors such as the adders, pipeline latches, control logic, and communication between the main predictor and the checkpoint SRAM, we would need a detailed physical design and layout to even begin to accurately estimate the overall power impact. This level of analysis is beyond the scope of this paper, but will be examined in future research.

5. Conclusions

We have introduced two new techniques for reducing the complexity and power of path-based neural branch predictors. While this study has focused on the original path-based neural predictor, our proposal can apply to any of the similar neural prediction algorithms such as the hashed perceptron [18] or the piecewise-linear branch predictor [8]. We have shown that the combination of modulo path-history and bias-based filtering can reduce power by decreasing the total number of tables used by the predictor as well as reducing the activity factor of the update phase of prediction. The modulo path-history technique also reduces the implementation complexity of the path-based neural predictor by reducing the predictor pipeline depth to only 4-6 stages, as opposed to 18-48 stages for the original predictor.

While this study has focused on a conventional path-

Predictor Size	PBNP		With Mod. Path and BBF		% Reduction
	Num SRAM Arrays	Bits per Checkpoint	Num SRAM Arrays	Bits per Checkpoint	
2KB	18	144	5	40	72.2
4KB	26	208	5	40	80.8
8KB	32	256	5	40	84.4
16KB	33	264	6	48	81.8
32KB	43	344	4	32	90.7
64KB	48	384	4	32	91.7

Table 2: Impact on predictor pipeline depth and checkpointing overhead. The number of SRAM arrays is equal to the path length, plus one for the bias table. Checkpoint overhead estimates assume one 8-bit value per predictor pipeline stage.

based neural predictor, other similar predictors could also benefit from either or both modulo path-history and BBF. The piecewise-linear neural branch predictor is a generalization of the PBNP that computes m different PBNP summations in parallel [8]. These m parallel computations increase the complexity of a deeper pipelined predictor, and modulo-path history may be very useful in this context to keep that complexity under control. The m computations also require m times as many weights to be updated, and bias-based filtering may also be very useful to reduce the activity of the piecewise-linear predictor. There are likely other predictors designs that can make use of the ideas presented in this study.

Acknowledgements

Gabriel Loh is supported by funding and equipment from Intel Corporation. Daniel Jiménez is supported by a grant from NSF (CCR-0311091) as well as a grant from the Spanish Ministry of Education and Science (SB2003-0357).

References

[1] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Micro Magazine*, pages 59–67, February 2002.

[2] Po-Yung Chang, Marius Evers, and Yale N. Patt. Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 48–57, 1996.

[3] Veerle Desmet, Hans Vandierendonck, and Koen De Bosschere. A 2bcgskew Predictor Fused by a Redundant History Skewed Perceptron Predictor. In *Proceedings of the 1st Championship Branch Prediction Competition*, pages 1–4, Portland, OR, USA, December 2004.

[4] Hongliang Gao and Huiyang Zhou. Adaptive Information Processing: An Effective Way to Improve Perceptron Predictors. In *Proceedings of the 1st Championship Branch Prediction Competition*, pages 1–4, Portland, OR, USA, December 2004.

[5] Simcha Gochman, Ronny Ronen, Ittai Anati, Ariel Berkovitz, Tsvika Kurts, Alon Naveh, Ali Saeed, Zeev Sperber, and Robert C. Valentine. The Intel Pentium M Processor: Microarchitecture and Performance. *Intel Technology Journal*, 7(2), May 2003.

[6] Dirk Grunwald, Donald Lindsay, and Benjamin Zorn. Static Methods in Hybrid Branch Prediction. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 222–229, Paris, France, October 1998.

[7] Daniel A. Jiménez. Fast Path-Based Neural Branch Prediction. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 243–252, San Diego, CA, USA, December 2003.

[8] Daniel A. Jiménez. Piecewise Linear Branch Prediction. In *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005.

[9] Daniel A. Jiménez and Calvin Lin. Neural Methods for Dynamic Branch Prediction. *ACM Transactions on Computer Systems*, 20(4):369–397, November 2002.

[10] Gabriel H. Loh. The Frankenpredictor. In *Proceedings of the 1st Championship Branch Prediction Competition*, pages 1–4, Portland, OR, USA, December 2004.

[11] Manhesh Mamidipaka and Nikil Dutt. eCACTI: An Enhanced Power Estimation Model for On-Chip Caches. TR 04-28, University of California, Irvine, Center for Embedded Computer Systems, September 2004.

[12] Pierre Michaud, Andre Seznec, and Richard Uhlig. Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 292–303, Boulder, CO, USA, June 1997.

[13] Erez Perelman, Greg Hamerly, and Brad Calder. Picking Statistically Valid and Early Simulation Points. In *Proceedings of the 2003 International Conference on Parallel Architectures and Compilation Techniques*, pages 244–255, New Orleans, LA, USA, September 2004.

[14] Andrè Seznec. Revisiting the Perceptron Predictor. PI 1620, Institut de Recherche en Informatique et Systèmes Aléatoires, May 2004.

[15] Premkishore Shivakumar and Norman P. Jouppi. CACTI 3.0: An Integrated Timing, Power, and Area Model. TR 2001/2, Compaq Computer Corporation Western Research Laboratory, August 2001.

[16] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-Aware Microarchitecture: Modeling and Implementation. *Transactions on Architecture and Code Optimization*, 1(1):94–125, March 2004.

[17] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton. Continual Flow Pipelines. In *Proceedings of the 11th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, Boston, MA, USA, October 2004.

[18] David Tarjan and Kevin Skadron. Merging Path and Gshare Indexing in Perceptron Branch Prediction. CS 2004-38, University of Virginia, December 2004.

A Break-Even Formulation for Evaluating Branch Predictor Energy Efficiency

Michele Co, Dee A.B. Weikle, and Kevin Skadron
Department of Computer Science
University of Virginia

Abstract

Recent work has demonstrated that a better branch predictor can increase the energy-efficiency of the system, even if the new predictor consumes more energy. Consequently, understanding the tradeoff between reduced misspeculation, execution time, and increased power spent within a branch predictor is critical.

This paper proposes a simple, effective metric for evaluating the tradeoff between processor energy-efficiency and branch predictor energy. By calculating a *break-even* branch predictor energy budget for a given program and an energy-efficiency target, we are able to evaluate the energy-efficiency of several existing branch predictor designs and provide a simple way to think about energy-efficiency.

Furthermore, we develop a method for deriving a branch predictor energy budget without requiring a power model for the proposed branch predictor. We evaluate this approach by comparing the budgets we calculate with results from simulation. Average error in our estimates is less than 1.5% for all pipeline configurations with a confidence of ± 0.02 to ± 0.06 Joules (1.7%-2.1%) for the integer benchmarks and ± 0.01 to ± 0.02 Joules (1.1%-1.7%) for the floating point benchmarks for the evaluated configurations.

1 Introduction

The computer engineering community has focused a great deal of attention in recent years on energy-efficiency. This is important for almost all new chip designs today. Battery life is a concern for mobile devices, and even for systems running off wall power, utility costs are a concern–especially for large data centers.

Recent work by Parikh et al. [14] explored the energy-efficiency of branch predictors, and generally concluded that the better the predictor, the better the chip's energy-efficiency, since better prediction reduces mis-speculated work and execution time, and hence reduces activity throughout the CPU. This means that spending *more* power in the predictor can still reduce power and improve energy-efficiency. Parikh et al. [14] and other subsequent work [1, 3] have looked at ways to reduce power in the branch predictor.

We are not aware of any work that establishes a general framework for when a new branch predictor design is energy-efficient. Recent innovations in branch prediction that consume more power per prediction, coupled with continuing concerns about energy-efficiency, make a break-even analysis valuable. This is especially important for the very large predictors that have been proposed in recent years. For example, the piecewise-linear branch predictor proposed by Jimenez et al. [11] only confers significant benefits above 32 KB, consuming from 1% - 11% of total chip power. A number of other recent predictors could consume non-trivial amounts of power as well [9, 16, 19]. The contribution to total chip power of a few predictor organizations for four different pipeline configurations is categorized in Figure 1 (See Section 3 for details on processor configurations). These power estimates are based on Wattch [4] simulations.

This paper derives a simple metric for the break-even energy efficiency of a branch predictor. It captures the tradeoff between reduced mis-speculation and execution time, and increased power in the branch predictor. The metric is based on the ED^2 metric [20] that has become popular because it is voltage independent. This technique gives the designer a way to predict how much energy is available to spend on a new branch predictor design relative to an existing processor design for a given performance of a particular program. This method for deriving break-even branch predictor energy budget can be used with or without a power model for the new design.

In this paper we:

- evaluate several existing branch predictor designs for energy-efficiency using our technique (1KB bimodal branch predictor as reference)
- observe that in general, for a single program, the power of the remainder of the processor excluding branch predictor is relatively constant regardless of branch predictor used
- demonstrate how to determine a break-even budget

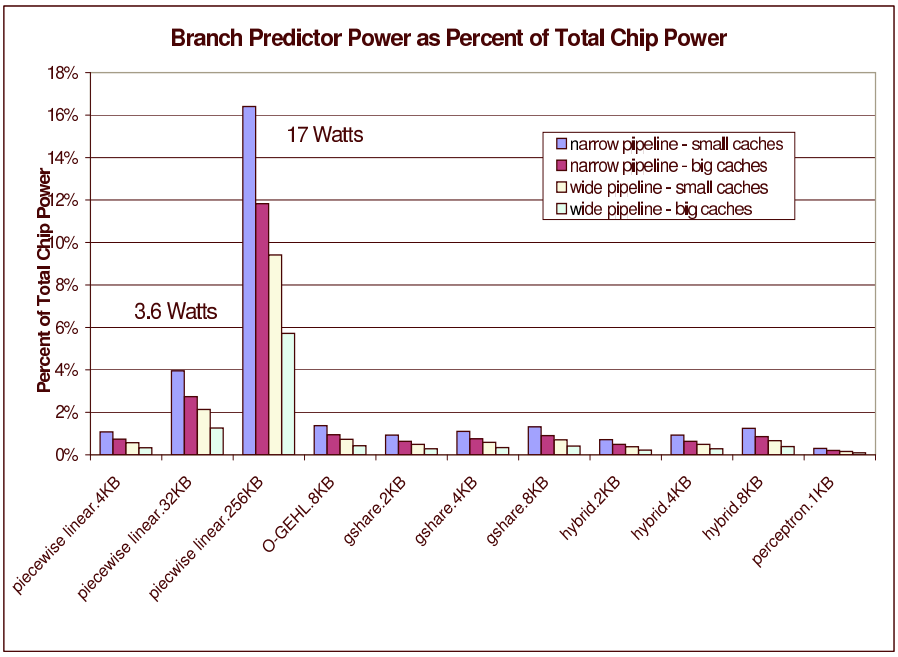


Figure 1: Branch predictor power as percent of total chip power in various processor configurations. (See Section 3.1 for configuration details.)

for a new design without a power model by estimating the new non-branch predictor energy and examine the validity of this estimation

- evaluate the independence of pipeline width, cache size, and leakage on the energy-efficiency trends between existing branch predictor designs

- show an approximate bound on branch predictor energy budget and on benefit that can be obtained through perfect branch prediction

This paper shows that pipeline width and cache size affect the energy budget of a predictor, but do not affect the relative performance of the branch predictors evaluated. It also shows that many branch predictors, including path perceptron and hybrid global/local predictors, use only a fraction of the energy they could to maintain energy-efficiency when compared to a 1 KB bimodal branch predictor reference. In some cases, the branch predictor uses only 8.4% of its energy budget.

We believe that our technique is general enough to apply to other processor structures as long as the assumption of constant energy per cycle for the remainder of the processor is adjusted to accurately describe the relationship of that structure's behavior within the processor.

The rest of the paper is organized as follows: Section 2 presents an explanation of our energy-efficiency evaluation technique, Section 3 presents experimental methodology, Section 4 presents current experimental results, Section 5 presents related work, and Section 6 presents conclusions and directions for future work.

2 Derivation of Break-Even Formula

This section describes the reasoning behind the technique for calculating the energy budget for a branch predictor design given a specific program, a reference processor configuration, and a reference branch predictor. The goal is two-fold. First, we want to determine if a new branch predictor is at least as successful as a previously designed predictor in terms of speed and power consumption. Second, we would like to determine an upper bound on the the benefits of a new branch predictor - specifically whether there is a sufficient benefit available to warrant further work toward an even better predictor.

2.1 Derivation of Energy Budget Formula

Assume for a program P, that the total energy of a processor (E_{total}) can be described as the sum of the energy of the branch predictor (E_{bpred}) plus the energy of the remainder of the processor ($E_{remainder}$):

$$E_{total} = E_{bpred} + E_{remainder} \quad (1)$$

Given a reference processor configuration, $Config_{ref}$, and a new processor configuration, $Config_{new}$,[1] we would like to have the same or better ED^2 for $Config_{new}$

[1] $Config_{new}$ differs from $Config_{ref}$ by only the branch predictor design

when compared to $Config_{ref}$. We define this goal, the *break-even point*, based on the ED^2 metric to be:

$$E_{total\_new} \times D^2_{total\_new} = E_{total\_ref} \times D^2_{total\_ref} \quad (2)$$

where E_{total_new} is the total energy for running program P on $Config_{new}$ and D_{total_new} is the total new delay (execution time). While E_{total_ref} is the total energy for running the same program P on $Config_{ref}$, and D_{total_ref} is the total reference delay (execution time).

Expanding and rearranging (2) using (1), we obtain:

$$\frac{(E_{remainder\_new} + E_{bpred\_budget}) \times D^2_{total\_new}}{(E_{remainder\_ref} + E_{bpred\_ref}) \times D^2_{total\_ref}} = 1 \quad (3)$$

$E_{predicted\_remainder\_new}$ is the predicted energy that $Config_{new}$ will consume not including the branch predictor energy. E_{bpred_budget} is the amount of energy that the new branch predictor may consume and still maintain the same ED^2 as $Config_{ref}$. E_{bpred_ref} and $E_{remainder\_ref}$ are the energy for the branch predictor of $Config_{ref}$ and the energy for the remainder of $Config_{ref}$ (not including branch predictor energy), respectively. D_{total_ref} is the delay or total execution time (in seconds) of running program P on $Config_{ref}$.

We would like to know how much energy the new branch predictor is allowed to consume given a particular ED^2. Therefore, we solve for E_{bpred_budget}. Rearranging the terms of (3) yields:

$$E_{bpred\_budget} = \frac{(E_{remainder\_ref} + E_{bpred\_ref}) \times D^2_{total\_ref}}{D^2_{total\_new}} - E_{remainder\_new} \quad (4)$$

All of the energy and delay values for the reference processor ($E_{remainder\_ref}$, E_{bpred_ref}, and D_{total_ref}) can be obtained from cycle-accurate simulation, or if available, from actual hardware measurement paired with the use of performance counters. The total execution time of $Config_{new}$, D_{total_new}, is gathered from cycle-accurate simulation. The energy for $Config_{new}$ not including the branch predictor, $E_{remainder\_new}$, may be obtained from simulation results as described in Section 4.1 or may be estimated as described in Section 4.2. The branch predictor energy budget, E_{bpred_budget}, is calculated from Equation (4).

The intent behind the derivation of this formula is to make explicit the tradeoffs designers make between performance and power consumption involved in branch prediction. This formula breaks the branch prediction budget down into individual components that can be filled in with the most accurate information the designer has on hand. In some cases, it may allow quick calculations based on simple simulations or even estimated prediction rates to determine if a particular approach is worth pursuing further.

3 Experimental Methodology

All experiments in this work use SimpleScalar [5] and a modified Wattch [4] infrastructure with a power model based on the 0.13μ Alpha 21364 [18]. The microarchitecture model is summarized in Table 1.

To model leakage, when a port or unit is not in use, a fixed ratio of maximum power dissipation is charged: 10% in most of our experiments.

3.1 Parameters

In our experiments, we evaluate variations of three factors in processor design:

- pipeline width: 4-wide issue (narrow) and 16-wide issue (wide)
- L1 and D1 cache sizes: 16KB (small) and 256KB (big)
- leakage ratio: 10% and 50%

We chose these parameters as a starting point for our study. In terms of pipeline width, we chose a pipeline that is 4-wide to consider today's processors and a 16-wide to look forward to more aggressive processor issue widths. For caches, we chose a very small cache and a cache more representative of what might be seen in current processors. In addition, we modeled leakage ratios of 10% to reflect current technology and 50% leakage ratio to look forward to the future process technology trends.

3.2 Branch Predictors Evaluated

The specific details of the branch predictor designs evaluated are listed in Table 2. We evaluate variations on path perceptron [19], gshare [13], hybrid [6, 12], O-GEHL [15], and piecewise linear [10] predictors. The same size BTB (2k-entry, 2-way set associative) and RAS (32-entry) are used for all branch predictors. We use the bimodal predictor as the reference model for all of the calculations described in Section 2.

3.3 Benchmarks

We evaluate our results using the integer and floating point benchmarks from the SPEC CPU2000 suite. The benchmarks are compiled and statically linked for the Alpha instruction set using the Compaq Alpha compiler with SPEC *peak* settings and include all linked libraries but no operating-system or multiprogrammed behavior.

Simulations are fast-forwarded according to Sherwood and Calder's SimPoint numbers [17], then run in full-detail cycle-accurate mode (without statistics-gathering)

Processor Core	
Active List	128 entries
Physical registers	80
LSQ	128 entries
Issue width	**wide**: 16 instructions per cycle **narrow**: 4 instructions per cycle
Functional Units	**wide**: 16 IntALU,4 IntMult/Div, 8 FPALU,4 FPMult/Div, 2 mem ports **narrow**: 4 IntALU,1 IntMult/Div, 2 FPALU,1 FPMult/Div, 2 mem ports
Memory Hierarchy	
IL1 & DL1	**small**: 16KB, 64-byte line, 2-way set associative, LRU **big**: 256KB, 64-byte line, 2-way set associative, LRU
L2	Unified Cache, 4 MB, 8-way LRU, 128B blocks, 12-cycle latency, writeback
Memory	225 cycles (75ns)
TLB Size	128-entry, fully assoc., 30-cycle miss penalty
Branch Predictor	
BTB	2 K-entry, 2-way
RAS	32-entry

Table 1: Simulated processor microarchitecture.

Name	Area	Description
bimodal	1KB	4k-entry, 2-bit counters
path perceptron	10KB	64-bit global history register 10 1KB tables 10 bits of history per table
gshare	2KB	13-bit history, 8k-entry
gshare	4KB	14-bit history, 16k-entry
gshare	8KB	15-bit history, 32k-entry
hybrid	2KB	2k-entry meta, 2k-entry bimodal, 4k-entry 2lev
hybrid	4KB	4k-entry meta, 4k-entry bimodal, 8k-entry 2lev
hybrid	8KB	8k-entry meta, 8k-entry bimodal, 16k-entry 2lev
O-GEHL	8KB	6, 2k-entry, 4-bit tables 1k-entry, 5-bit counter table 2k-entry, 5-bit counter table 1k-entry, 1-bit tag table 48-bit global history register 48-entry, 8-bit global address register
piecewise linear	8KB	weight table: 8590-entry 7-bit counters bias table: 599-entry 7-bit counters global path history: 48 8-bit addresses 48-bit global history register local history table: 55 16-bit shift registers
piecewise linear	32KB	weight table: 34360-entry 7-bit counters bias table: 3496-entry 7-bit counters global path history: 26 8-bit addresses 26-bit global history register local history table: 220 16-bit shift registers

Table 2: Branch predictors evaluated

for 300 million instructions to train the processor structures –including the L2 cache—and the branch predictor before statistics gathering is started. This interval was found to be sufficient to yield representative results [8].

4 Results

The following sections describe the results of comparing the break-even branch predictor budgets of several current branch predictor designs with their actual energy consumption. Section 4.1 shows how budgets are calculated using cycle-accurate simulated values for $E_{remainder\_new}$. Section 4.2 then explains how to estimate $E_{remainder\_new}$ and validates this estimate by comparing it to the simulated results. Section 4.3 describes an upper bound for branch predictor budgets based on perfect branch prediction. Section 4.4 then shows how changing the leakage ratio to 50% affects the results.

4.1 Branch Predictor Budgets from Simulation

E_{bpred_budget} can be calculated using $E_{remainder\_new}$ gathered from cycle-accurate simulation. Figure 2 shows the calculated E_{bpred_budget} values for each benchmark for each processor configuration. The actual values of the results differ based on processor configuration, but the overall relationship between the branch predictors evaluated is the same. Both the path perceptron and the O-GEHL predictor have a higher branch predictor energy budget on average than the other branch predictors evaluated. This indicates that both the path perceptron's and O-GEHL's prediction accuracy are much higher than the other predictors evaluated and the increased prediction accuracy boosts the energy budget available to be spent on their implementation and still satisfy the *break-even point*.

For all the graphs, the *gshare.2KB* predictor often displays negative branch predictor energy budget values. A negative budget on a particular benchmark indicates that the branch predictor's performance does not improve the overall processor performance sufficiently to recoup the energy expended in the predictor itself. In other words, even if the branch predictor were to consume zero energy, the branch predictor would not be able to fulfill the ED^2 *break-even point* and the reference predictor would be a better choice. On these specific benchmarks this is because the 2KB area causes destructive aliasing which cripples the gshare predictor's accuracy.

Figure 3 shows the percent of the calculated branch predictor energy budget consumed by the evaluated branch predictors. Any predictor that uses less than 100% on a particular benchmark is worth considering for that benchmark. Branch predictors which use less than 100% of their energy budget are providing additional energy-efficiency beyond the break-even point. On average the trends of the branch predictor energy consumption amongst the benchmarks are the same. Several benchmarks exhibit consumption in excess of the budget for one or more predictors, some in excess of 300%. For all these benchmarks, the particular branch predictor was unable to come close to the *break-even* ED^2 point. More specifically, branch predictors which exhibit greater than 300% energy budget consumption can not fulfill the energy break-even point, and often have negative branch predictor budgets, which we represent in our graphs as infinitely large numbers above the 300% demarcation line.

The trend evident in the per benchmark graph of Figure 3 is still maintained when the average of the benchmarks is calculated as in Figure 4 which summarizes the overall results for both integer and floating point benchmarks. Once the average is plotted per predictor, though, it is clear that it is the gshare.2KB predictor that is not a benefit to power consumption when all benchmarks are considered. Note that on the floating point benchmarks few examples exist of predictors exceeding their budget. This is because the floating point benchmarks are very predictable, enabling even simple predictors to obtain a maximum benefit.

At this point one may ask why not just simulate the entire configuration and determine which predictors deliver better performance with less energy consumption. In the next section, we show an example of estimating one of the simulated components of the equation. We believe that this formula enables us to understand better and more quickly not only the benefits of a particular predictor, but the potential benefits of additional improvements as well.

4.2 Branch Predictor Energy Budgets without an Existing Power Model

It is desirable to estimate $E_{predicted\_remainder\_new}$ when there is no pre-existing power model readily available for the candidate branch predictor.

If and only if it can be assumed that the average energy per cycle used by the non-predictor portion of the reference design is equal to the average energy per cycle of the non-predictor portion of the new design while running the same program, then[2]:

$$\frac{E_{remainder\_ref}}{D_{total\_ref}} = \frac{E_{predicted\_remainder\_new}}{D_{total\_new}} \tag{5}$$

Solving for $E_{predicted\_remainder\_new}$:

$$E_{predicted\_remainder\_new} = \frac{E_{remainder\_ref} \times D_{total\_new}}{D_{total\_ref}} \tag{6}$$

Equation (6) will then allow a designer to solve for E_{bpred_budget}, which is the value of interest. Substitute

[2]The accuracy of this assumption is explored in Section 4.2

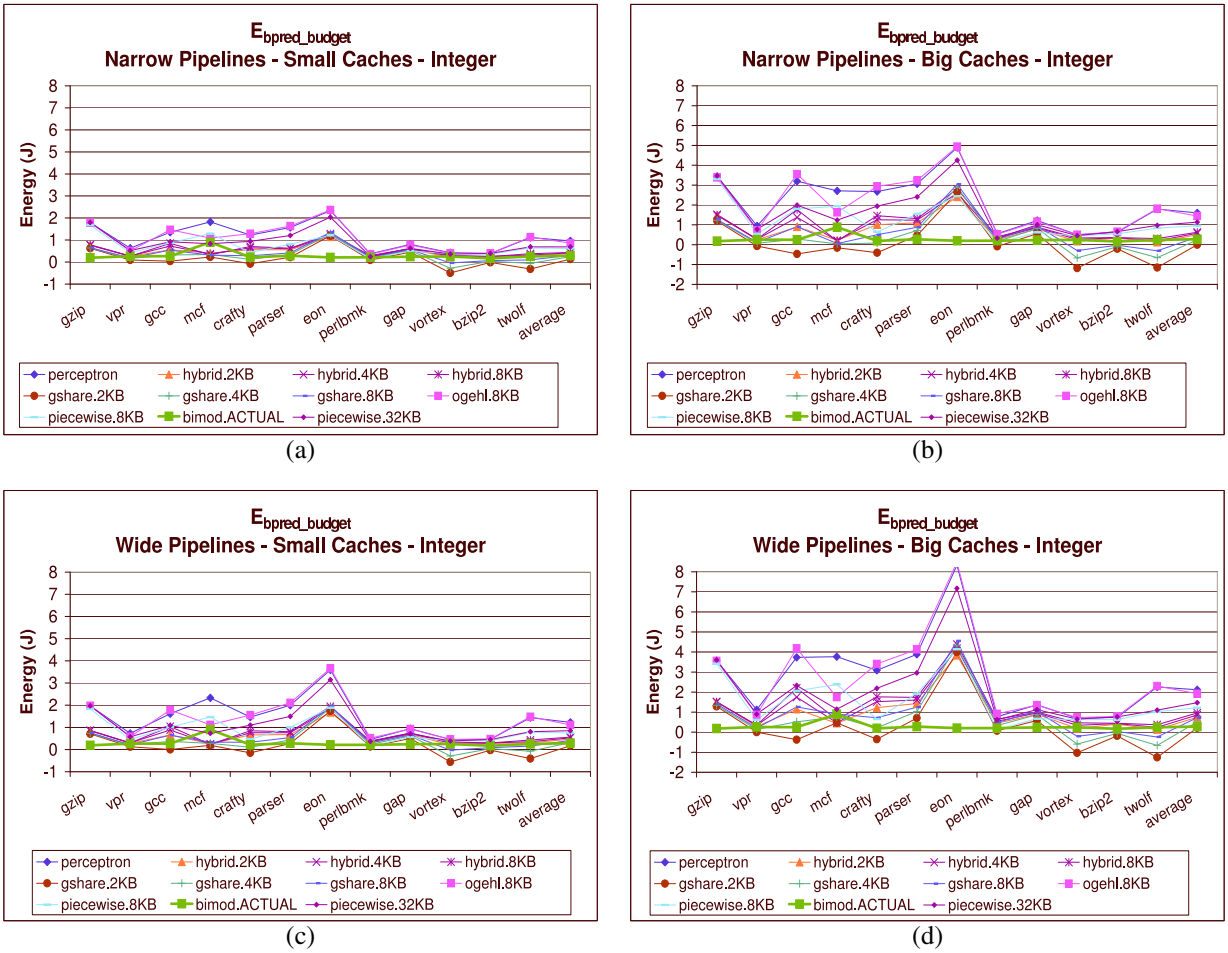


Figure 2: Calculated branch predictor energy budget (E_{bpred_budget}) for processor configurations. Note that in some cases that the energy budget is negative. This indicates that even at zero branch predictor energy consumption, the new predictor's performance is insufficient to recoup the energy (in excess of the reference predictor's energy) expended in the predictor itself.

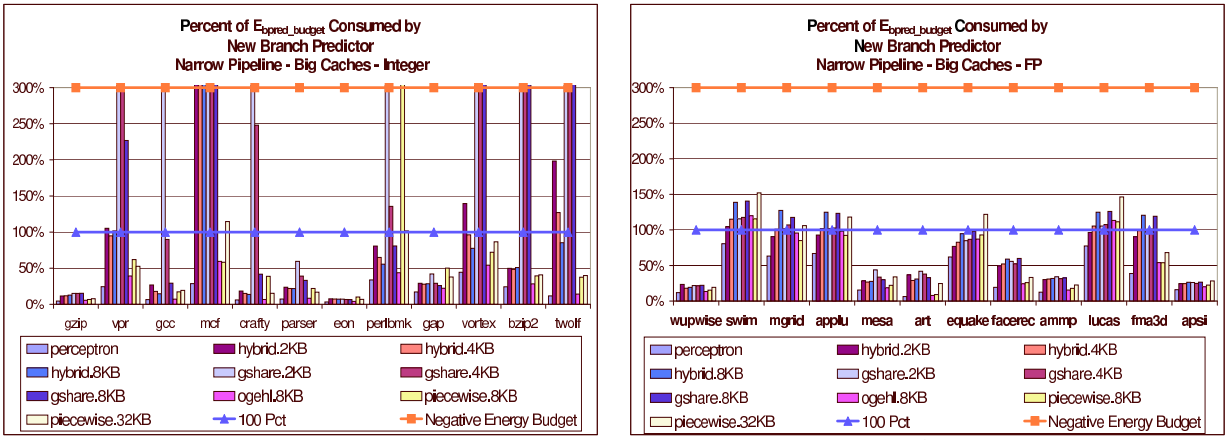
(6) into (4):

$$E_{bpred\_budget} = \overbrace{\frac{(E_{remainder\_ref} + E_{bpred\_ref}) \times D^2_{total\_ref}}{D^2_{total\_new}}}^{\text{Total energy budget for } Config_{new}} - \overbrace{\frac{E_{remainder\_ref} \times D_{total\_new}}{D_{total\_ref}}}^{E_{remainder\_new}} \quad (7)$$

Note that Equation (7) is simply an expanded version of Equation (1).

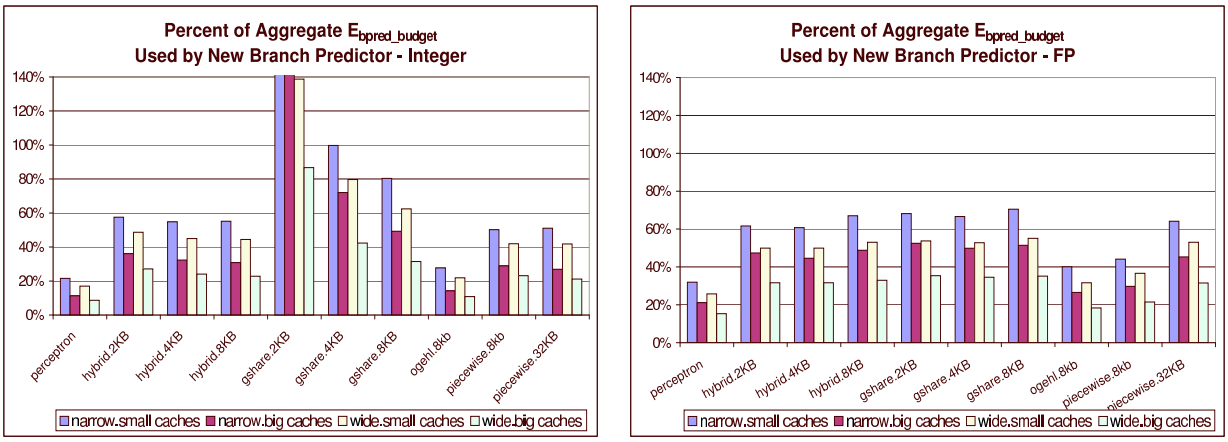
Figure 5 (a) shows the estimated $E_{predicted\_remainder\_new}$ for the SPECint2000 benchmarks run on a processor configuration with a narrow pipeline and big caches. Figure 6 shows the raw $E_{predicted\_remainder\_OGEHL.8KB}$ and $E_{actual\_remainder\_OGEHL.8KB}$ results overlaid for the 8 KB O-GEHL predictor for both integer and floating point benchmarks. Notice that there is little difference between the remainder energy predicted when compared to the remainder energy gathered from simulation data. We display the 8 KB O-GEHL predictor results because our technique exhibits the largest absolute difference on it of all the branch predictors evaluated. Raw results for the other processor configurations included in the study were similar in trend, so are omitted due to space constraints.

Figure 7 shows the absolute percent difference between $E_{predicted\_remainder\_new}$ and $E_{actual\_remainder\_new}$ for the four main processor configurations evaluated. The range of error is at most 11% and on average less than 1.5% for all benchmarks and for all processor configurations. Figure 8 summarizes the average percent deviation of $E_{predicted\_remainder\_new}$ from $E_{actual\_remainder\_new}$ for each of the configurations and clearly shows that the standard deviation from $E_{actual\_remainder\_new}$ for both integer and floating point benchmarks is very small. The



(a) (b)

Figure 3: Percent of branch predictor energy budget (E_{bpred_budget}) actually consumed by branch predictor as measured by simulation for narrow pipeline with big caches processor configuration: (a) integer benchmarks, (b) floating point benchmarks. Note that lower values indicate branch predictor energy consumption lower than (better) E_{bpred_budget} and higher values indicate energy consumption greater than E_{bpred_budget} (worse). Note that branch predictor energy budget percentages below 100% line perform better than the energy break-even point. Configurations with negative branch predictor energy budgets are manually set to exceed the 300% threshold.



(a) (b)

Figure 4: Percent of aggregate E_{bpred_budget} consumed by new branch predictor. This is the ratio between the total E_{bpred_actual} across the entire workload and the total E_{bpred_budget} across the entire workload

error bars on the graph show $\pm$ 1 standard deviation.

The closeness of our estimated $E_{predicted\_remainder\_new}$ to $E_{actual\_remainder\_new}$ shows that the assumption made in Equation (5) is relatively accurate. Since the expectation is that the same amount of work is being performed in each case (executing a particular benchmark or program), one might think that the processor configuration with the shorter execution time would use a greater energy per cycle at the break-even point. The configuration with the slower execution time is actually carrying out additional work to recover from mis-speculation, rather than performing useful work. Since this estimate of the $E_{remainder\_new}$ is fairly accurate, we believe it is possible to determine a branch predictor energy budget without actually having a power model for the branch predictor. Section 4.4 shows that with increasing leakage ratio, our estimate only becomes more accurate.

4.3 Estimating an Upper Bound for Branch Predictor Energy Budgets

It would be interesting to evaluate the limits of branch predictor energy budgets. The best possible branch predictor energy budget comes from ideal or perfect branch prediction. We performed an experiment using an ap-

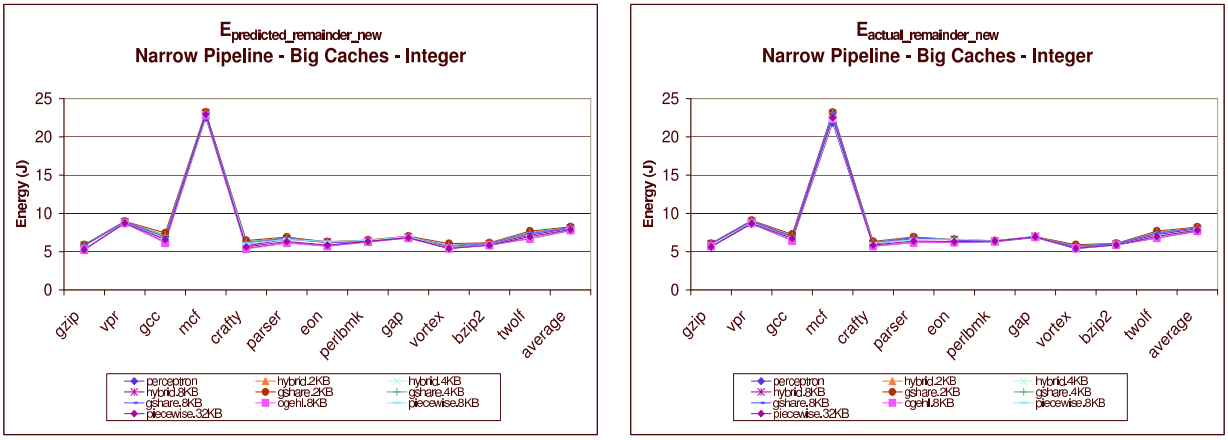


Figure 5: (a) predicted energy of remainder of $Config_{new}$, $E_{predicted\_remainder\_new}$ and (b) actual/simulated energy of remainder $E_{actual\_remainder\_new}$.

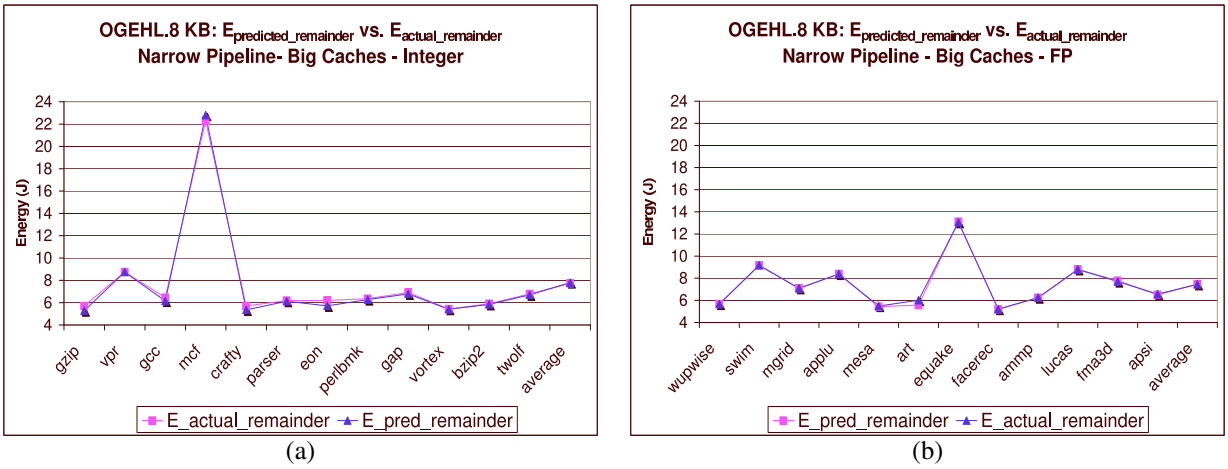


Figure 6: OGEHL.8KB $E_{actual\_remainder}$ vs. $E_{pred\_remainder}$ (a) integer and (b) floating point benchmarks.

proximated perfect branch prediction technique in which branch mispredictions are detected in the decode stage and then corrected. Misfetches are not avoidable with this technique. However, this method gives us an idea of where the upper bound of branch predictor energy budgets lies.

Figure 11 shows the results of our experiment for both integer and floating point benchmarks. Since the integer benchmarks are less predictable, the E_{bpred_budget} with our pseudo-perfect technique is clearly greater (better) than that of the other predictors in our evaluation, with O-GEHL and piecewise linear branch predictors coming very close on *eon*. For the floating point benchmarks, one might be puzzled that the perfect prediction E_{bpred_budget} is not the highest budget number on average, but rather O-GEHL and piecewise linear branch predictors have similar or higher budgets, especially for *art*. Both *eon* and *art* are benchmarks in which our pseudo-perfect prediction technique fails due to high numbers of instruction misfetches. For example, on *eon* our pseudo-perfect branch prediction technique misfetches 20 instructions per 1k instructions, and 8.6 instructions per 1k instructions for *art*. This is much higher than both O-GEHL's and piecewise linear branch predictors' misfetch rates which are effectively zero (0.05-0.08 misfetches per 1k instructions).

The graph shows that there is still some benefit to be gained by improving branch prediction accuracy and gives us an approximate idea of where this upper bound lies.

4.4 Leakage Effects

All of the previous experiments were run using a value of 10% for leakage. To see the effects of leakage values more in line with future processor technology, we ran selected experiments using a 50% leakage ratio.

Figure 9 shows that the general trend for the branch

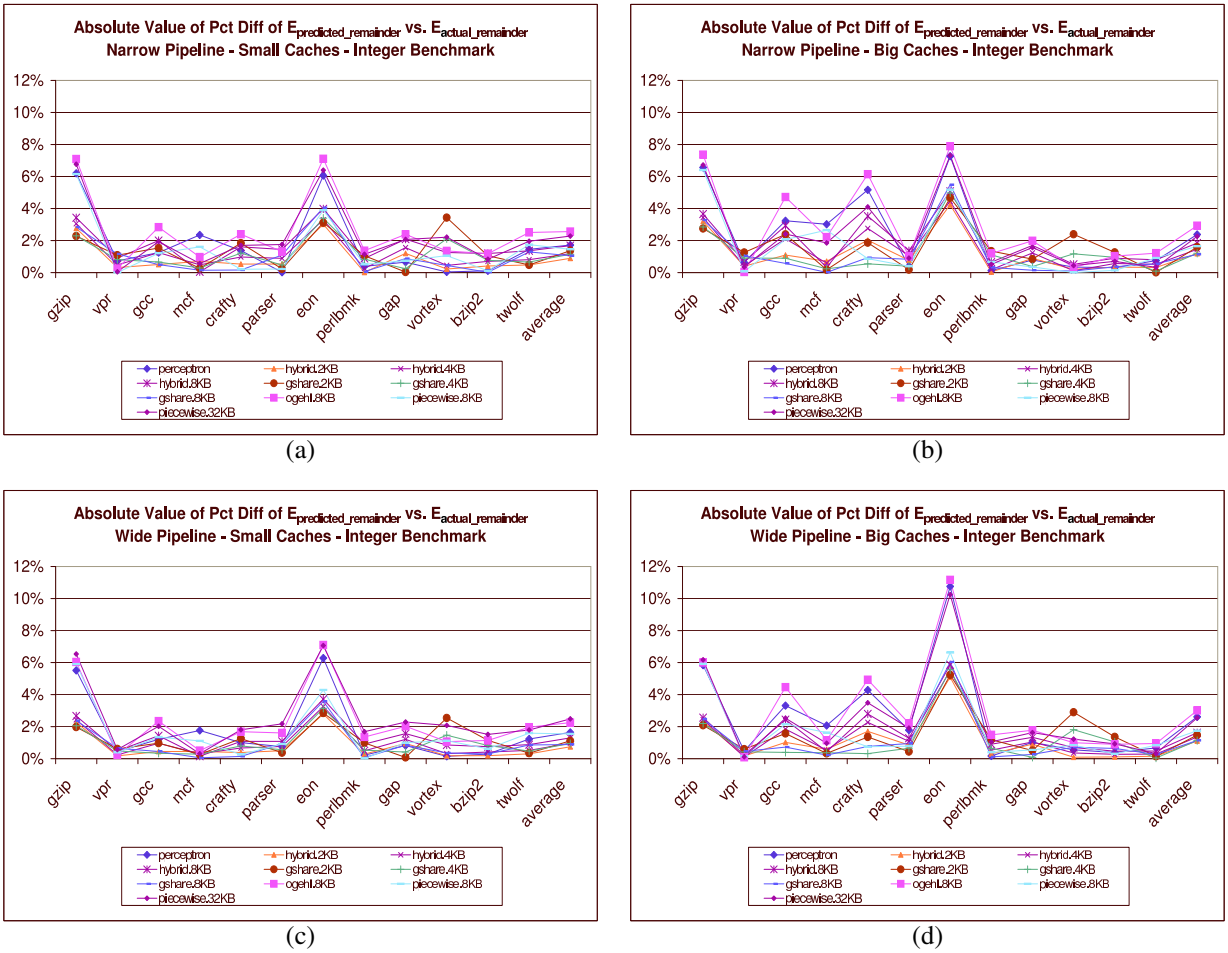


Figure 7: Absolute Value of Percent Difference between $E_{predicted\_remainder\_new}$ and $E_{actual\_remainder\_new}$ for (a) narrow pipeline with small caches, (b) narrow pipeline with big caches, (c) wide pipeline with small caches, (d) wide pipeline with big caches.

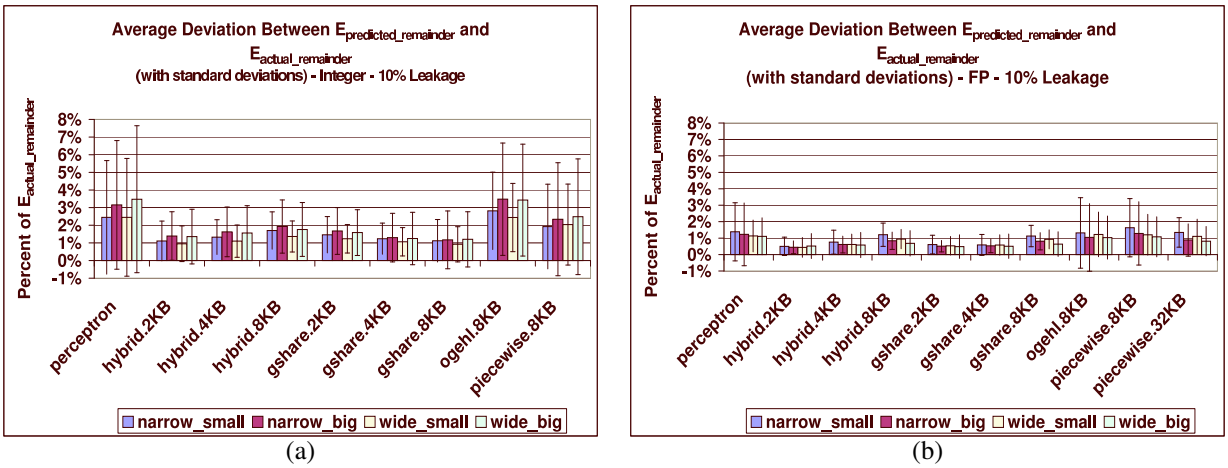


Figure 8: Average deviation of $E_{predicted\_remainder\_new}$ from $E_{actual\_remainder\_new}$ for 10% leakage ratio: (a) integer and (b) floating point benchmarks. Error bars illustrate $\pm$ 1 standard deviation.

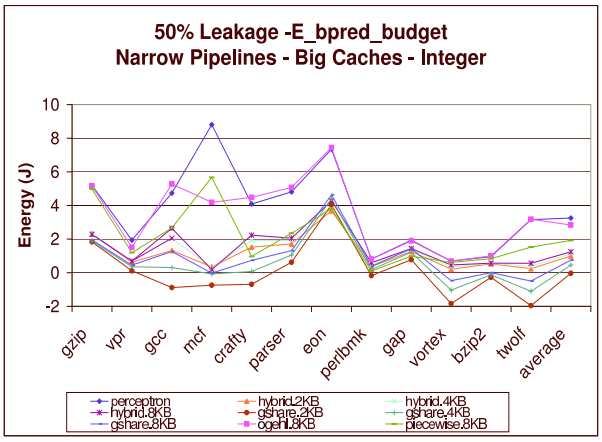


Figure 9: 50% Leakage Ratio: $E_{bpred\_budget\_new}$ for narrow pipelines, big caches for integer benchmarks. Note that although the magnitude of the graph is amplified, the general shape of the graph is very similar to the shape of the graph in Figure 2(b).

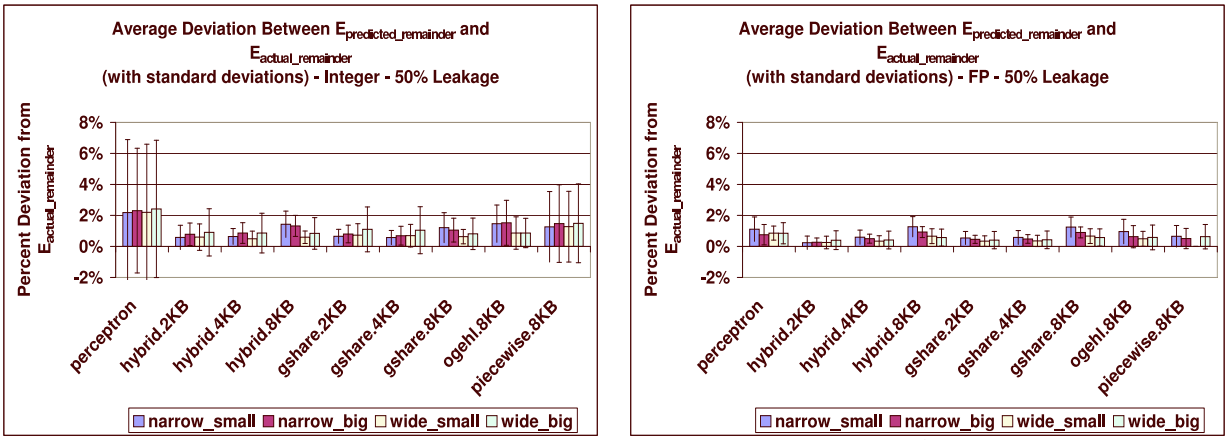


Figure 10: 50% Leakage Ratio: Average $E_{predicted\_remainder\_new}$ and standard deviation from $E_{actual\_remainder\_new}$ for (a) integer and (b) floating point benchmarks. Note that although the predicted energy goes up compared to 10% leakage results, the standard deviation does not increase proportionally.

predictor budget is the same between predictors and amongst benchmarks. Results for other configurations and for the floating point benchmarks were very similar and are not included due to space constraints.

We also see from the results that our estimation of $E_{pred\_remainder\_new}$ is still accurate as the leakage ratio increases. In fact, the standard deviation does not increase proportionally with the leakage ratio as shown in Figure 10.[3] This is due to the fact that the power difference between activity and inactivity in the structures is less due to increased leakage. This demonstrates that our technique is still useful as leakage begins to dominate.

[3]Piecewise linear 32 KB results not included.

5 Related Work

Parikh et al. [14] explored the energy-efficiency of branch predictors. They concluded that better prediction accuracy led to better processor energy-efficiency. They also made the insight that spending additional power in the predictor can still reduce overall power and improve processor energy-efficiency. The work in this paper builds on this insight and develops a metric for determining a branch predictor energy budget for a new branch predictor design.

Aragon et al. [1] analyze the reasons for performance loss due to conditional branch mispredictions and develop a simple technique for fetching, decoding, and renaming along the alternate path for low confidence branches to reduce misprediction penalty and thus reduce overall energy

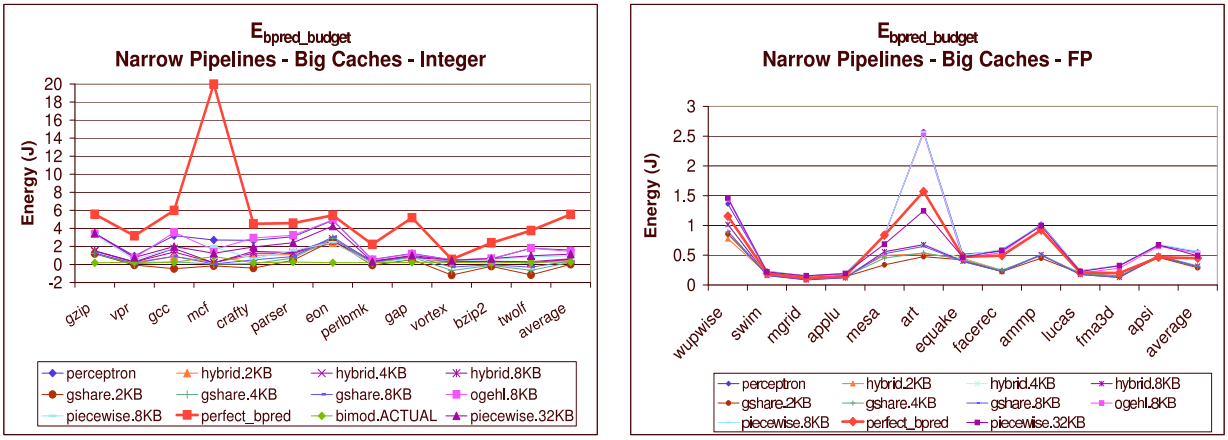


Figure 11: (E_{bpred_budget}) for perfect branch prediction for narrow pipeline, big caches configuration.

consumption. Baniasadi and Moshovos [2] examined reducing branch predictor power dissipation by selectively turning off tables in a combined branch predictor design. In additional work, Baniasadi and Moshovos exploit the insight that branches in steady state do not need to update the branch predictor to reduce the number of branch predictor accesses and therefore reduce branch predictor energy consumption. [3] Chaver et al. [7] proposed a method for using profiling to characterize branch prediction demand. They use this information to selectively disable portions of a hybrid branch predictor and resize the branch target buffer to reduce branch predictor energy consumption. Our work does not develop a specific technique for reducing energy consumption, but rather demonstrates a method for reasoning about the energy-efficiency of branch predictor designs.

6 Conclusions

This paper describes a general, systematic method for calculating the *break-even energy budget* for a branch predictor design. The method requires a cycle-accurate performance and power/energy model for a reference processor and a cycle-accurate simulation of the branch predictor design under consideration. An accurate estimate of the energy budget can then be made without a power/energy model for the candidate branch predictor. The techniques presented in this paper allow the comparison of the energy-efficiency of different branch predictor designs without having to equalize branch predictor area or branch prediction accuracy rates. It further gives a branch predictor designer a technique with which to easily determine the energy available to achieve an energy-efficient branch predictor, given the performance for a set of programs and an upper bound on the energy available for an ideal predictor.

This paper also evaluates the branch predictor energy budgets for several existing branch predictor designs on the SPECcpu2000 benchmarks and evaluates the energy-efficiency of these designs. We also put forth the notion that average energy per cycle consumption of the remainder of the pipeline varies little between different branch predictor designs. We further find that the branch predictor performance and energy trends are fairly independent relative to pipeline width and cache size, thus reducing the design space exploration needed during future branch predictor research. Finally, these results were determined to hold even when the leakage was increased to 50%.

Overall, our results suggest that even the very aggressive branch predictors recently proposed in ISCA 2005 do not yet violate energy efficiency bounds, at least not when aggregating across SPEC overall as a workload. This indicates that research on further improvements in branch prediction is warranted.

7 Future Work

There are many directions in which this work may be extended. The study could easily be expanded to include all configurations of larger, more aggressive, and more complex branch predictor designs as well as no-predictor configurations. branch predictors, which are improved versions of the designs presented in [10, 15].

An upper bound for branch predictor energy budget can be demonstrated and used to estimate the bound on benefit that can be obtained from achieving perfect branch prediction.

This work also has the potential to lead to a technique to estimate a branch predictor energy budget without requiring either cycle-accurate simulator or power model for the future branch predictor design. We envision that in the future, given a functional simulator with which to derive branch prediction accuracy on a particular program, designers will be able to derive the branch predictor break-even energy budget much earlier in the design process, allowing them to narrow the design space search

much more quickly.

This technique could also be combined with program phase detection techniques and adaptive hardware techniques to develop a method to improve energy-efficiency by adapting the branch predictor hardware based on program characteristics and break-even energy information.

Another interesting factor to explore is the impact of training time due to operating system context switches on branch predictor energy budget.

We believe that the method described in this paper can be refined to allow simpler estimates for the power/performance tradeoffs associated with branch predictor design. In addition, the technique described in this paper could be applied as presented to determine the energy budgets of other processor structures such as caches, register files, buffers, etc.

This paper shows how these results are independent of cache size, pipeline width, and leakage ratio. Future work could determine the effect of register file size and buffers as well.

Acknowledgments

This work is supported in part by the National Science Foundation under grant nos. NSF CAREER award CCR-0133634, and CNS-0340813, EIA-0224434, and a grant from Intel MRL. We would also like to thank Jason D. Hiser for his helpful input.

References

[1] J. L. Aragon, J. Gonzalez, A. Gonzalez, and J. E. Smith. Dual path instruction processing. In *Proceedings of the 2002 International Conference on Supercomputing*, pages 220–229, New York, NY, USA, 2002. ACM Press.

[2] A. Baniasadi and A. Moshovos. Branch predictor prediction: A power-aware branch predictor for high-performance processors. In *Proceedings of the 2002 International Conference on Computer Design*, pages 458–461, 2002.

[3] A. Baniasadi and A. Moshovos. Sepas: a highly accurate energy-efficient branch predictor. In *Proceedings of the 2004 International Symposium on Low Power Eelctronics and Design*, pages 38–43, New York, NY, USA, 2004. ACM Press.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[5] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.

[6] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. Patt. Branch classification: a new mechanism for improving branch predictor performance. In *MICRO27*, pages 22–31, New York, NY, USA, 1994. ACM Press.

[7] D. Chaver, L. Pi nuel, M. Prieto, F. Tirado, and M. C. Huang. Branch prediction on demand: an energy-efficient solution. In *Proceedings of the 2003 International Symposium on Low Power Eelctronics and Design*, pages 390–395, New York, NY, USA, 2003. ACM Press.

[8] J. W. Haskins, Jr. and K. Skadron. Memory reference reuse latency: Accelerated sampled microarchitecture simulation. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Syste ms and Software*, pages 195–203, Mar. 2003.

[9] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 14–23, 1997.

[10] D. Jimenez. Idealized piecewise linear branch prediction. In *Proceedings of the First Workshop Championship Branch Prediction in conjunction with MICRO-37*, December 2004.

[11] D. A. Jiménez. Piecewise linear branch prediction. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, page TBD. IEEE Computer Society, 2005.

[12] S. McFarling. Combining Branch Predictors. Technical Report TN-36, June 1993.

[13] S. McFarling and J. Hennesey. Reducing the cost of branches. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 396–403, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[14] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan. Power issues related to branch prediction. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, pages 233–44, Feb. 2002.

[15] A. Seznec. The O-GEHL branch predictor. In *Proceedings of the First Workshop Championship Branch Prediction in conjunction with MICRO-37*, December 2004.

[16] A. Seznec. Analysis of the O-GEometric History Length branch predictor. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, page TBD. IEEE Computer Society, 2005.

[17] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[18] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 2–13, June 2003.

[19] D. Tarjan, K. Skadron, and M. Stan. An ahead pipelined alloyed perceptron with single cycle access time. In *Proceedings of the 5th Workshop on Complexity-Effective Design*, 2004.

[20] V. Zyuban and P. Strenski. Unified methodology for resolving power-performance tradeoffs at the microarchitectural and circuit levels. In *Proceedings of the 2002 International Symposium on Low Power Eelctronics and Design*, pages 166–171. ACM Press, 2002.

Heuristics for Complexity-Effective Verification of a Cache Coherence Protocol Implementation

Dennis Abts* dabts@cray.com

Ying Chen† wildfire@ece.umn.edu

David J. Lilja† lilja@ece.umn.edu

*Cray Inc.
P.O. Box 5000
Chippewa Falls, Wisconsin 54729

†University of Minnesota
Electrical and Computer Engineering
Minnesota Supercomputing Institute
Minneapolis, Minnesota 55455

Abstract

In this paper we investigate several search heuristics to reduce the number of explored states required to discover a cache coherence protocol error. We describe a novel method for extracting traces called "witness strings" from the formal verification process and executing these traces on the cache coherence protocol implementation using a logic simulator. A deeply ensconced hardware error may require searching hundreds of thousands of states to expose the error. We describe a very efficient search heuristic called min-max-predict that, for example, reduced the number of states explored from 101423 to a mere 1787 states for an error embedded in the Cray X1 cache coherence protocol. When executing the witness strings from this optimized search we are able to find the error in about three minutes of logic simulation versus about three hours of logic simulation to execute the witness strings from the non-optimized depth-first search. We show that the min-max-predict search heuristic performs better than BFS and DFS for all 12 protocol errors embedded in the Cray X1 and the Stanford DASH coherence protocols.

1 Introduction

Shared memory multiprocessors must enforce mutual exclusive access over the shared main memory in order to present a consistent view of the memory hierarchy to the programmer. The memory consistency model (MCM) described by the instruction set architecture (ISA) provides a set of rules that the programmer (and compiler) must follow to ensure that a parallel program executes correctly. The cache coherence protocol is a fundamental ingredient of the MCM responsible for propagating writes. Showing that a cache coherence protocol is correct is nontrivial, as there are many aspects to "correctness" and the protocol state space is very large. The complexity of verifying memory coherence was shown by Cantin [1] to be NP-hard. Furthermore, a simple correctness property, such as "a load from an address always returns the value of the last write," specified at the architectural level is often extremely difficult to verify in the implementation. Moreover, discovering errors in the pre-silicon stages of the development is paramount to the both the budgetary and schedule success of the project. Our approach is to formally model the protocol using the Murφ [2] verification system and prove that a set of well-defined, fundamental properties hold over the state space. Then, by capturing *witness strings* from a depth-first search of the protocol state space and executing the traces on the Verilog RTL logic simulation we show that the implementation matches the formal specification. While the witness strings approach has been effective at uncovering errors in the Cray X1 cache coherence protocol [3] both in the high-level protocol specification and the RTL implementation, it does suffer from computational complexity when executing the witness string on the RTL logic simulator. For instance, the model checker explores 214 million states in 208 hours which averages to about 287 states per second. However, the logic simulator is only capable of exploring about 10 states per second which would require weeks of compute time. While it is tractable to execute all the witness strings on the RTL logic simulator, it is not very *practical* within the constraints of a project development schedule.

This research focuses on search heuristics that when used to guide the depth-first search of the model checker will produce witness strings that are better at uncovering "hard to find" errors that are deeply ensconced in the state space. To evaluate the efficacy of each heuristic, we measure the *distance* of the error from the initial state. That is, how many states were explored before the error was discovered. We chose cache coherence protocol from the Cray X1 and Stanford DASH multiprocessors as an experimental substrate. This paper makes several contributions:

- we evaluate several search heuristics to improve the efficiency of a model checker at uncovering errors in a cache coherence protocol — one such heuristic called

min-max-predict, for example, reduced the number of states explored from 101423 to a mere 1787 states for an error embedded in the Cray X1 cache coherence protocol,

- we describe an approach to hierarchical verification of the cache coherence protocol which uses the witness strings generated by the model checker to show the implementation correctly implements the cache coherence protocol, and
- finally we describe assume-guarantee rules to allow compositional techniques enabling a highly-distributed cache coherence protocol to be decomposed into smaller sub-task constituents and verified independently.

Section 2 gives a framework for the system model and describes the hierarchical verification process. We evaluate three heuristics: hamming, cache-score and min-max-predict which are discussed in Section 3. Then, in Section 4 we provide the experimental setup used for evaluating the efficacy of each heuristic. The experimental results are shown in Section 5, with related work in Section 6. Finally we draw some conclusions in Section 7.

2 System Model

The design and verification process proceeds in a hierarchical manner, where each successive stage has increasing levels of detail. In general, the design process has the following steps:

1. functional specification (S_f),
2. design specification (S_d),
3. implementation in register-transfer language (I_{rtl}),
4. gate-level implementation (I_{gates}), and
5. physical layout and fabrication (I_{phys}).

The functional specification stage, S_f, sets forth the high-level description of the system in the form of a programming model, instruction set architecture (ISA), and so forth. It is in step 1 that a functional instruction set simulator is written to provide a coarse-grain model of the system. Continuing with the design specification, S_d, in step 2 which provides more detail about the microarchitectural details such as the organization of the cache memory system, size of buffers, pipeline design, etc. It is in this step that a cycle accurate detailed performance simulator would be used to make design trade-offs and properly size hardware structures. The end result of the S_d stage is a detailed engineering specification with details necessary to begin constructing the hardware description in step 3. Using a hardware description language such as Verilog or VHDL, the system is described using register-transfer language, I_{rtl}. Then, after the I_{rtl} is completed a set of synthesis tools are used to compile the RTL description into a gate-level technology dependent implementation, I_{gates}. Finally, in step 5 the physical design description, I_{phys}, accounts for transistor geometries and ensures that the physical design rules are satisfied. The granularity of the design process is similar to the differences between rocks, stones, gravel, pebbles, and sand.

Similar to the design process, the verification process admits a hierarchical approach to verification with the following stages:

1. architectural verification (V_{arch}),
2. implementation verification (V_{rtl}), and
3. gate-level verification (V_{gate}).

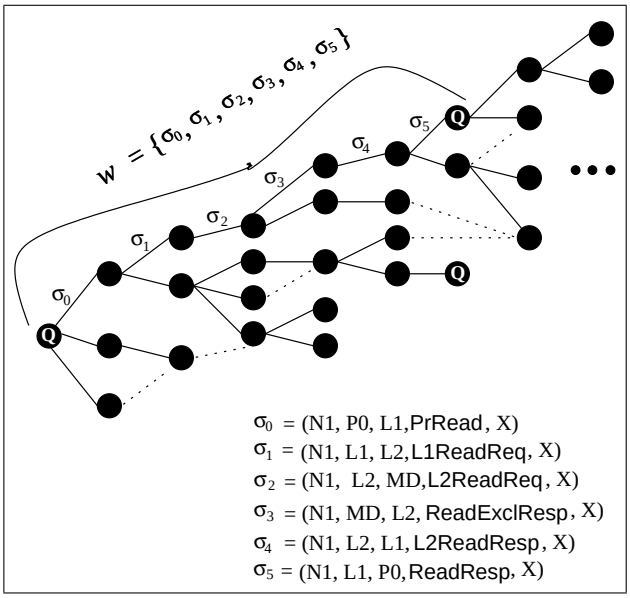
The verification flow occurs simultaneously with the design stages with obvious dependencies on availability of the design stage being checked. Note, however, that architectural verification is usually reserved for critical system components responsible for guarantees made to the user. For example, the memory consistency model makes some formal guarantees about the order of memory events to different locations. To check if these guarantees are being satisfied it is useful to construct a high-level model of the system and ensure that for all possible states the properties of the memory consistency model hold. We call this high-level verification model, V_{arch}, and is carried out early in the design process. Subsequent verification steps will use the actual RTL description, V_{rtl}, with its increasingly detailed description usually being checked using a discrete-event logic simulator. Lastly, the final verification step is gate-level verification, which usually requires checking that the structural description (an ASIC gate-level netlist) is functionally equivalent to the RTL description. In practice, this step is automated by a formal equivalence check between the gate-level and RTL descriptions. Unfortunately, no comparable check exists to ensure that the RTL description is a correct refinement of the more abstract functional specification. This paper addresses this problem for a multiprocessor cache coherence protocol, however, the ideas would apply to other domains.

2.1 Hierarchical verification and refinement

While performing verification in a hierarchical fashion, we must check proof obligations of the form $P \preceq Q$, where P and Q are system descriptions and $\preceq$ is a *refinement relation* on the system descriptions. The assertion $P \preceq Q$ holds if P describes the same system as Q, but perhaps at a finer level of detail. Consider, for example, the design of a microprocessor where P may be a Verilog RTL description of the processor core (functional units, pipelined datapath, etc), and Q may be the corresponding instruction set architecture (ISA). The relationship between P and Q can then be stated as "P implements Q" or equivalently "P is a refinement of Q."

At each level of the memory hierarchy the hardware is controlled by a finite-state machine (FSM) that governs the state transitions specified by the coherence protocol. The different levels of the memory hierarchy exchange data using *micropackets* which provide efficient and reli-

Figure 1: A collection of symbols form a word, w, in the language $\mathcal{L}(\mathbf{V_{arch}})$.



able point-to-point communication between the FSMs at each level of the memory hierarchy.

We begin by formally defining an FSM by the 5-tuple:

$$\mathsf{FSM} = (Q, \Sigma, \delta, q_o, F)$$

where Q is a finite set of states, Σ is a finite *input alphabet*, $q_o \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, and δ is the *transition function* mapping $Q \times \Sigma$ to Q. The transition function, $\delta(q, x)$ takes as its argument the current state, $q \in Q$, and an input symbol, $x \in \Sigma$, to produce a new state Q. The transition function may cause zero or more atomic *actions* based on the current state q and the input message x.

The cache coherence protocol is specified as a set of tables describing the operation of each FSM in the memory hierarchy. These cooperating FSMs encapsulate the *rules* that govern the protocol. Each of the caches have some *state* (Q) that describes their access permission at any point in time. The protocol tables describe the action in terms of the messages (i.e. micropackets) that flow between the components in the memory hierarchy. Each table is decomposed into a set of FSMs according to which virtual communication channel is used. For instance, incoming processor requests (e.g: PrRead and PrWrite messages) flow on virtual channel 0 (vc0) and L2 responses (e.g.: ReadResp and GrantExcl) flow on virtual channel 1 (vc1). So, two concurrent FSMs are constructed: one to handle incoming processor requests on vc0 and another to handle L2 responses on vc1. This decomposition step is applied to the protocol specification at each level of the memory hierarchy creating a set of FSMs that interact in a producer-consumer fashion using the virtual network as the communication medium.

2.2 Formal model of a cache coherence protocol

The formal specification of the cache coherence protocol is described as a set of *rules* in the Murφ specification language. The specification is compiled to produce the protocol verifier. As the formal model executes it produces a *trace tree* (Figure 1), τ, where each node in τ is a new configuration C_i where each edge represents a rule firing. In Figure 1, the dotted arcs represent a rule firing that does not produce a new (unique) state and is therefore not considered part of the trace tree. The symbols $\sigma_1, \sigma_2, \ldots, \sigma_n$ label the rule firings, where $C_i \xrightarrow{\sigma_i} C_{i+1}$. Formally, a word $w = \{\sigma : C_i \xrightarrow{\sigma} C_j\}$, where C_i and C_j are quiescent configurations or C_j is a terminal configuration (leaf node). The σ symbols are recorded on the witness file as each rule is fired. The sequence of symbols $\langle \sigma_0, \sigma_1, \ldots, \sigma_i \rangle$ that traces a path from the starting configuration C_0 to a leaf node is called a *witness string* for the execution of the formal model.

A *quiescent* configuration is one in which there are no messages in-flight in the virtual network and the state at each level of the memory hierarchy is not Pending. Intuitively, this means that all outstanding requests have been satisfied and there are no messages in-flight on the virtual network.

2.3 Language containment

Let V_{arch} be an instance of a formal system specification that describes the behavior of a cache coherence protocol, and $\mathcal{L}(V_{arch})$ (the *language* of V_{arch}) denote the set of all sequences α accepted by V_{arch}. The form of each α_i is consistent with the witness symbols σ_i. The *alphabet* of V_{arch} is the set of commands in the memory system. Specifically, the alphabet is the set of *messages* that are exchanged among the components of the cache coherence protocol.

It is convenient to choose *trace containment* (also referred to as language containment) as a refinement relation $\preceq$ on the system descriptions $V_{rtl} \preceq V_{arch}$. Intuitively, then every sequence of inputs and outputs that is possible for V_{rtl} is also possible for V_{arch}. Trace containment is defined *globally* for arbitrary length input-output sequences, which makes it practically impossible to check for trace containment except for small system descriptions because the check is exponential in the number of states in V_{arch}. However, as Henzinger et al [4] point out, the relation between V_{rtl} and V_{arch} is often much tighter where each implementation state of V_{rtl} corresponds to a specification state of V_{arch}. Instead, a much stronger relation is captured by a *simulation relation* $V_{rtl} \preceq_s V_{arch}$. Formally, V_{arch} is said to simulate V_{rtl} if starting from the initial configuration C_0 and continuing ad infinitum every (input, output) pair can be checked against every (input, output) pair in V_{arch}. If V_{arch} simulates V_{rtl} then a verification certificate (or witness) can be produced in the form of a relation between states of V_{rtl} and states of V_{arch} and the witness, α, can be efficiently (polynomial-time) checked for correctness. It is useful to note, the

length of the witness $|\alpha|$ depends on the number of states in the formal system model which is exponential on the size of the system description V_{arch}.

Because the simulation relation $\preceq_s$ preorder is defined *locally* by considering only individual (input, output) pairs, we can apply an *assume-guarantee principle* [5][6][7] to exploit compositional techniques for dividing the verification task into subtasks $V^1_{rtl} \parallel V^2_{rtl} \preceq_s V^1_{arch} \parallel V^2_{arch}$. Intuitively, assume-guarantee rules can be thought of as *properties* (guarantees) and *constraints* (assumptions) of the subtasks. We establish the properties (i.e. guarantees) of the refinement relation $V^1_{rtl} \preceq_s V^1_{arch}$ subject to the constraints (i.e. assumptions) of V^2_{arch} and denote the subtask refinement as $V^1_{rtl} \parallel V^2_{arch} \preceq_s V^1_{arch}$. In other words, V^1_{rtl} refines V^1_{arch} when constrained by an environment (i.e. witness) that behaves like V^2_{arch}. Likewise, the refinement relation for the subtask $V^2_{rtl} \parallel V^1_{arch} \preceq_s V^2_{arch}$ also holds. As Henzinger et al [4] show, the assume-guarantee principle allows the *compound* witness $V_{rtl} \preceq_s V_{arch}$ to be constructed from the witnesses of the subtasks $V^1_{rtl} \parallel V^2_{arch} \preceq_s V^1_{arch}$ and $V^2_{rtl} \parallel V^1_{arch} \preceq_s V^2_{arch}$.

2.4 Composition

Any large system design requires compositional [5] techniques that deal with the sheer enormity of the system state. For example, a memory controller may be divided into multiple components. Each component being responsible for implementing a portion of the memory hierarchy, and the cache coherence protocol spans multiple components. The system model, V_{arch}, consists of subtasks $L2$, and M, and we write $V_{arch} = L2 \parallel M$, where $L2$ is the component which implements the secondary-level cache, and M implements the memory directory. Each σ symbol is a tuple describing the rule firing as: $\sigma_i = \langle n, src, dest, cmd, addr \rangle$, where n is the node identifier, src is the source component, $dest$ is the destination component, cmd is the command (message) type, and $addr$ is the address. We represent the address symbolically so that, for instance, a three node system would have three addresses: X, Y, and Z. An example of a witness symbol is $\langle$N1, P, L2, PrRead, $X\rangle$ which is a request from the processor (P) to the L2 cache controller to perform a PrRead of address X. A simple memory transaction will consist of several symbols that witness the memory reference as it propagates through the memory hierarchy. For example an allocating read request from the processor results in the following:

$$\langle N1, P, L2, \mathsf{PrRead}, X\rangle$$
$$\langle N1, L2, M, \mathsf{MRead}, X\rangle$$
$$\langle N1, M, L2, \mathsf{ReadExclResp}, X\rangle$$
$$\langle N1, L2, P, \mathsf{PResp}, X\rangle$$

The witness symbols, σ, are instantiated with a physical address which is random but mapped to a given memory directory for variables X, Y, and Z. In general, a cache coherence protocol exhibits the property of *data independence* since the data values are merely being propagated by the protocol. The only constraint on the variable selection is that the address remains fixed for the entire length of a witness string. The function $M = Home(\mathbf{var})$ is used to resolve which memory directory (M) component the address belongs.

Table 1: A snippet from the Cray X1 L2 cache coherence protocol specification.

Current State	*Incoming Command*	*Next State*	*Action*
Invalid	PrRead	Pending	M(MRead)
Invalid	ReadMod	Pending	M(MReadMod) ; increment(wc)
Invalid	PrReadNA	Invalid	M(MGet)
Invalid	VWrite	Pending	M(MReadMod) ; increment(wc)
Invalid	VWriteNA	Invalid	NOP()
Invalid	VWriteData	Invalid	M(MPut) ; increment(wc)
Invalid	ReadSharedResp	*err*	ERROR()
Invalid	ReadExclResp	*err*	ERROR()
Invalid	FlushReq	Invalid	M(MFlushAck)
...	...	...	...

2.5 Composite witness strings

As the Murφ verifier executes the formal system the sequence of transitions that *witnesses* the verification is captured into a witness string file. The witness strings are enumerated in a depth-first manner from the initial start state where each rule firing is delineated with a sequence of dashes. The atomic actions between the dashes represents one or more witness symbols, σ_i. The example in Figure 3 is a very simple sequence of events where a processor is simply making a Read(Y) request. However, this simple request has a total of six memory transactions associated with it. Markers are inserted into the stimulus when the memory system is quiescent, since this is often a good point to make assertions about the state of the memory system.

Transactions in the memory system can be categorized as two types: sending a message to another component, and receiving a message. Witness symbols σ_i generated by Murφ are of the form:

send: [P|E|M]n sending $cmd(var)$ to [P|E|M]n on vc[0|1|2]
recv: [P|E|M]$n(state) \leftarrow cmd(var)$ on vc[0|1|2] from [P|E|M]

where n is the node number associated with that component, *state* is the cache or memory directory state, *cmd* is the message type (e.g. Read, Write, Inval, etc), and vc is the *virtual channel* which the message will travel on within the virtual network. The Cray X1 uses three virtual channels, vc0, vc1, and vc2, to avoid request-response dependencies in the cache coherence protocol.

Since there is a one-to-one mapping of commands to virtual channels, the vc is omitted from the witness symbol. In other words, the vc is implied by the command and is thus unnecessary to explicitly specify it in the witness symbol.

2.6 Inverse abstraction and subtask witnesses

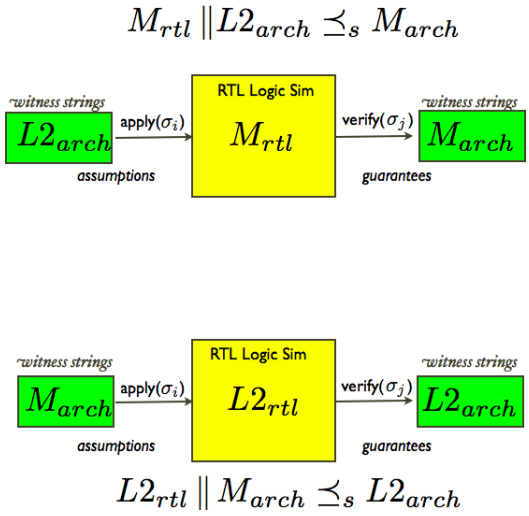
The composite witness strings are decomposed into their subtask components and all unbound variables are bound to concrete instantiations so that the resulting witness symbol can be encoded as a tuple $\sigma_i = \langle n, src, dest, cmd, addr \rangle$. At this time, the abstractions that are exposed because of symmetry reductions [8] in the formal system model are replaced by node numbers and variable names. To accomplish this, we must choose which subtask to decompose. We developed a program that takes the composite witness string as input and produces a subtask witness as $w_{L2} = \mathbf{ws}(w, E)$ or $w_M = \mathbf{ws}(w, M)$. The resulting witness symbols, σ_i, are enumerated so we have the length of the string $|w|$ (Figure 3).

2.7 Assume-guarantee rules for L2 and M subtasks

The Cray X1 cache coherence protocol spans multiple components (chips). Compositional techniques are used to divide this very large state into two smaller subtasks, namely the L2 (E chip) and M (M chip) components. The chip-level subtasks made a convenient choice for several reasons. First, the chips have a well-defined interface (pins) to the external environment and communicate by exchanging messages across a physical channel. The channel is managed using a link-layer protocol (LLP) which implements a sliding window go-back-N protocol for reliable transmission. However, to make the simulation more efficient, the LLP was replaced with a black box which assumed the channel to be reliable. Second, they represent what will actually be fabricated, so the RTL specification of the chip is readily available.

The assume-guarantee rules must ensure that $L2_{rtl} \parallel M_{arch} \preceq_s L2_{arch}$. That is, the L2 RTL when constrained by the memory directory (M) witness correctly implements the L2 architectural specification (Figure 2). Similarly, we must show that the memory directory (M) RTL implementation when constrained by the L2 witness correctly implements the M architectural specification, or concisely $M_{rtl} \parallel L2_{arch} \preceq_s M_{arch}$. Intuitively, the subtask witnesses specify assumptions under which the RTL implementation is verified. Those assumptions must be validated (guaranteed) during the corresponding subtask verification. The Raven [9] transaction-level verification framework provides very simple semantics for making a concrete connection between the assume-guarantee rule and Raven apply-verify functions. Specifically, when verifying the L2 subtask, for instance, the assumptions for the M_{arch} subtask witness are converted into apply(σ_i) and the guarantees being checked are represented as verify(σ_j), and vice versa. After checking each *word* in the witness string we call check_assertions which performs a check of all the invariants specified by the formal specification, V_{arch}, are also satisfied in the implementation, V_{rtl}. Recall, that a word of the witness string is defined between quiescent states, so we are assured that there are no outstanding requests in the network.

Figure 2: The RTL models are checked using the witness strings from the formal model as assume-guarantee rules.



3 Search Heuristics

We propose several search heuristics to optimize model checking of a cache coherence protocol. In particular, the heuristics seek to improve depth-first search by guiding the search process toward a portion of the protocol state space that is more likely to contain an error. Then, using the optimized search from the model checker we are able to produce witness strings that are more efficient, capable of finding errors in less time, than depth-first search. In the presence of computational complexity, faced with the daunting task of executing the voluminous numbers of witness strings on the logic simulator, the witness strings produced by the optimized model checker provide a complexity-effective verification of the protocol implementation. We evaluate the search heuristics on the Cray X1 and Stanford DASH.

3.1 Hamming distance

The hamming distance [10] between two states, s_1 and s_2, is defined as the number of bits by which s_1 and s_2 differ. Two heuristics max-hamming and min-hamming select the rule with either the largest or smallest hamming distance, respectively. Intuitively, the max-hamming heuristic will choose the next state that is most different from the current state, whereas the min-hamming heuristic chooses the next state that is least different. In Murφ each state is stored as a large bit-vector. We simply loop through the bit-vector and compare $s_1 \rightarrow$bits[i] to

Figure 3: The output from ws-Murφ is encoded into a format that is used by a *Raven* program as stimulus and expected results for the logic simulator.

```
Quiescent
E1(X:ShClean)<---Read(Y) on vc0 from P1 [13]
  E1 sending PInvalidate(X) to P1 [14]
  E1 sending MDrop(X) to M1 [15]
  E1 sending MRead(Y) to M2 [16]
E1(Y:PendingReq)<---ReadExclResp(Y) on vc1 from M2 [17]
  E1 sending PReadResp(Y) to P1 [18]
Quiescent
```

$s_2 \rightarrow$bits[i] incrementing a counter each time a difference is detected. Thus the overhead in computing the hamming distance is linear with respect to the size of each state.

3.2 Cache Score

It is common case that a small number of control bits can dominate the circuit functionality. For instance, the cache *state* and *tag* bits dominate the functional behavior for the L2 cache coherence controller. The cache-score heuristic uses a subset of the state information to determine the best rule to fire. For the Cray X1 protocol, the cache-score evaluation function inspects the internal state variables of the ws-Murφ verifier and returns a score, s = [0:4]. For example, for each memory directory (MD) component in the model, we score it according to the following code.

```
int md_score(int n) { // n is the node number
int s=0;
if(Node[n].Directory.State != Noncached &&
   Node[n].Directory.State != Shared    &&
   Node[n].Directory.State != Exclusive)
     s = 1 ;
return s ;
}
```

A similar function e_score( ) is used to score the L2 cache. The md_score( ) and e_score( ) functions take the node number as an argument. So, the aggregate score for each state is as follows:

score = e_score(1) + e_score(2) + md_score(1) + md_score(2);

A rule which produces a next state with a higher score is favored over a rule the produces a lower score. The score gives a metric of the amount of concurrent coherence traffic. Intuitively, a higher score means there are more outstanding coherent memory references and therefore more likely to produce an error.

3.3 Min-Max-Predict

Yang and Dill [11] used the minimum hamming distance as a search heuristic. It was their hope that states with very few bits differing from the error state will require fewer cycles to reach the target. Yang's position is contrary to our reasoning that choosing the next state via *maximum* hamming distance will move toward the error state. As a compromise to these differing positions, we arrived at our min-max-predict heuristic by combining the min-hamming and max-hamming heuristics with the scoring function from cache-score and a 3-bit saturating counter to predict, based on the current state, whether we should use the min-hamming or max-hamming heuristic. The semantics of the 3-bit saturating counter is very similar to the way a branch predictor predicts the next program counter and captures the notion of hysteresis in a program control flow.

The heuristic works by first determining a score for the current state in the range [0:n]. If the score $< n/2$, then we increment the count, otherwise we decrement the count. We make sure the count never exceeds the bounds of the 3-bit counter by first checking if count <8 before incrementing it, and checking if count > 0 before decrementing the count. Then, after we fire a rule we compute the hamming distance from the current state to the new state. We then use the hamming distance to set the min_rule and max_rule values that are used to evaluate the min-hamming and max-hamming heuristcs, respectively. The value from min_rule or max_rule is assigned to the variable best_rule depending on the current counter value. If count <4 then best_rule = max_rule, otherwise best_rule = min_rule. The up-down 3-bit saturating counter is used to steer the search toward the target error.

4 Experimental setup

We chose the Cray X1 and Stanford DASH cache coherence protocols as an experimental substrate. We embed errors into each protocol and use the model checker to search for the known error. This process of hiding and seeking memory coherence errors is repeated for six different errors on each protocol. While the X1 and DASH protocols are both directory-based [12] coherence protocols, they are largely dissimilar in other respects. For example, the X1 protocol is a blocking protocol and DASH uses NAKing (negative acknowledgments) to defer incoming requests to a pending cache line. The DASH protocol is also the predecessor to the SGI Origin2000 [13] cache coherence protocol.

The Murφ formal verification model of the DASH protocol has a total reachable state space of 10466 states where the size of each state is 4912 bits. The Cray X1

cache coherence protocol has a much larger state space with 214 million reachable states where the size of each state is 1664 bits. Each error is listed below with the invariant that failed.

4.1 Invariants

Model checking is a technique that exhaustively searches the states space of a concurrent system to show that certain properties hold. Safety properties are simple invariants expressed in first-order logic that, in essence, ensure that "something bad never happens." The invariants we checked are common to all cache coherence protocols and ensure such properties as exclusive write permission to a cache line, and data consistency among others. As an example, consider the "single writer" property for the Cray X1 cache coherence protocol [3].

> Two different caches, **p** and **q**, should never have write access to the same address, **a**, at the same time.
>
> $\forall_a \forall_p \forall_q$ **IsDirty**$(a, p) \wedge q \neq p \Rightarrow \neg$ **IsDirty**(a, q)

The single writer property is written in Murφ as:

```
Invariant "Single Writer."
Forall a : Address Do
 Forall p : NodeID Do
  Forall q : NodeID Do
    (p != q) & IsDirty(a, p) -> !IsDirty(a, q)
  End
 End
End ;
```

The DASH protocol verifier has an equivalent safety property that detects multiple caches with write permission. There are other invariants that check for data consistency, unexpected messages, etc.

4.2 Model checking

We used the Murφ verification system [2] to check our formal model. Murφ uses explicit state enumeration to exhaustively search the reachable state space of a system of interacting finite-state machines. In addition to deadlock freedom, the verifier checks that a set of invariants are satisfied at each explored state. We did, however, make small modifications to the original Murφ source code to allow us to capture witness strings from the verifier. The new verifier, called ws-Murφ, has minor modifications to several functions to capture witness strings and allow the addition of search heuristics.

To capture the witness strings, we added two lines of code to the original Murφ function verify_dfs in the mu_system.C file. The first modification prints a special token before each rule is fired. The second modification prints a message when the depth-first search backtracks. Then, each cache controller rule prints out the witness symbol as it executes, as shown in Figure 3. The witness symbols are converted into a Raven [9] diagnostic program that can be compiled and run with the Synopsys VCS simulator. The ws-Murφ verifier will generate millions of lines of C++ code to run on the logic simulator. This process would benefit tremendously from a search heuristic that yielded witness strings that are better at discovering errors than those generated from a depth-first search.

To implement search heuristics in ws-Murφ we added a new function AllNextStates_BestNS which chooses the rule to fire based on our heuristic, instead of the SeqNextState function. We also modified functions NextState, was_present, and simple_was_present to include a boolean flag indicating that we are searching a state that is being evaluated by the heuristic and therefore should not be added it to the hash table. For each state the heuristic returns a score that is used as a metric to determine the best rule to execute and produce the next state. The search process is guided by the heuristic toward a potential error.

4.3 Cray X1 embedded errors

Errors for the Cray X1 were chosen to provide a variety of errors with different invariant failures. Error1 violates the single writer invariant, Errors2-4 generate protocol errors, Error5 creates deadlock, and Error6 violates the data coherence invariant. Errors embedded in the X1 protocol are roughly divided into two categories: quiescent and transient state errors. That is, does the error occur when the L2 cache or memory directory is in a quiescent (not pending) or transient (pending) state.

Error1 L2(WaitForVData)←FwdRead
The L2 receives a FwdRead request while in a pending state waiting for vector write data. The FwdRead causes a SupplyDirtyInv reply to be sent to the requester and should transition to the Invalid state. Instead, the L2 state remains in the WaitForVData state.
Invariant violated: Single Writer

Error2 L2(ExClean)←FwdRead
The L2 receives a FwdRead request while in the ExClean state. The L2 responds with a SupplyDirtySh but never transitions from ExClean, to a shared state, ShClean.
Invariant violated: Protocol Error

Error3 L2(Pending)←Inval
The L2 cache receives an Inval message to invalidate the cache line, and erroneously sends a FlushAck instead of an InvalAck message.
Invariant violated: Protocol Error

Error4 MD(Shared)←Drop
The memory directory (MD) receives an L2 cache eviction message, Drop. The MD erroneously transitions to the Exclusive state instead of staying in the Shared state.
Invariant violated: Protocol Error

Table 2: Results for X1 and DASH. The entries in the table are the number of states explored by ws-Murφ.

			Hamming		Cache Score		Min-Max
	DFS	**BFS**	Min	Max	Min	Max	**Predict**
Error1	753	84915	23885	64	9218	265	35
Error2	483	88472	26	1704	2751	1784	93
Error3	1037	53808	2163	537	7293	693	751
Error4	101424	53815	223	30314	46910	109103	2118
Error5	101413	770162	268	78456	46910	109093	2180
Error6	101423	45320	223	30302	46901	109103	1787

(a) Results for the Cray X1. The min-max-predict heuristic performs better than DFS or BFS for all the error cases. The largest improvement reduces the search states from 101423 to only 1787 states.

			Hamming		Cache Score		Min-Max
	DFS	**BFS**	Min	Max	Min	Max	**Predict**
Error1	4719	3742	9093	4958	932	8553	2411
Error2	421	715	25	437	54	409	124
Error3	609	1034	3794	745	955	399	584
Error4	533	418	10265	504	924	78	410
Error5	773	1062	3794	1096	955	582	584
Error6	274	465	120	199	62	339	199

(b) Results for the Stanford DASH. The min-max-predict heuristic performs better than DFS or BFS for all the error cases. Although the improvement is not as drastic as the results for the Cray X1 protocol.

Error5 MD(Shared)←Read
The MD receives a Read request from the L2 cache, but never adds the cache to its sharing vector so the MD is not correctly tracking caches with a shared copy.
Invariant violated: Deadlock

Error6 MD(PendDrop)←Drop
The MD has detected that an L2 cache has re-requested a shared cache line and thus an L2 eviction notice must be in-flight. However, the MD erroneously does not toggle the bit in the sharing vector when it receives the Drop message.
Invariant violated: Data Coherence

4.4 Stanford DASH embedded errors

DASH has three types of requesters: the *local* or *remote* cluster, and *home* cluster, as well as two classes of memory references to local or remote memory. Errors were implanted by inspecting state diagrams of the protocol given in [12] and embedding errors that vary the failing invariant. Error1 is an error that was re-discovered with Murφ after it was originally uncovered after substantial amounts of simulation.

Error1 Handle read exclusive request to home
No invalidation is sent to the master copy requesting cluster, which has already invalidated the cache.
Invariant violated: Consistency of data

Error2 Send reply message instead of NAK
Instead of a negative acknowledgement (NAK) message, an acknowledgement (ACK) message is sent in reply.
Invariant violated: Writeback from non-dirty remote

Error3 Handle read request to remote cluster
Instead of changing the cache block in the remote cluster to be locally shared after sending the data block to the requesting remote cluster, the cache block remains locally exclusive.
Invariant violated: Writeback from non-dirty remote

Error4 Handle read exclusive request to remote cluster
Instead of changing the cache block in the remote cluster to be not locally cached, the cache block remains locally exclusive.
Invariant violated: Only a single master copy

Error5 Handle DMA read request to remote cluster
The dirty cache block changes to be locally shared instead of staying dirty after supplying the data.
Invariant violated: Writeback for non-dirty rmt DMA

Error6 Handle invalidate request to remote cluster
After collecting all the invalidation acknowledgements, the request entry in the Remote Address Cache (RAC) should transition to the invalid state. Instead, request entry is preserved and the state remains as waiting for a read reply.
Invariant violated: Condition for existence of master copy

5 Results

A summary of our experimental results is given in Table 2. Interestingly, for both protocols a depth-first

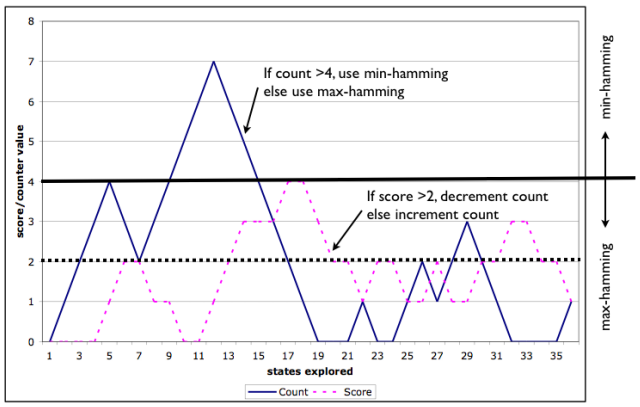
search (DFS) outperforms breadth-first search (BFS) on eight of the twelve error cases. As Dill, et. al. [2] point out, a BFS is generally used because it is believed to produce shorter error traces when an invariant fails. However, this is not consistent with our experience across a broad range of protocol errors. Errors 4-6 for the Cray X1, see Table 2(a), are quite deeply ensconced in the state space, having to explore in excess of 100 thousand states before the targeted error was discovered! These three errors had one thing in common, the error affected only the memory directory state or sharing vector and not the L2 cache state or messages exchanged. Even though the error occurs very early in the search process, it's *discovery* took much longer. On several of the errors in both protocols, the min-hamming heuristic proved to be very fruitful. However, it did significantly worse than DFS on several cases. Interestingly, when min-hamming performed poorly, the max-hamming heuristic always performed well, usually significantly better than DFS. Taken by themselves, the min-hamming and max-hamming heuristics do not perform consistently across the broad set of protocol errors.

The cache-score heuristic uses the cache state and memory directory state to guide the search. By favoring states with more outstanding requests we hope to guide the search toward states that are more likely to have errors. Unfortunately, this heuristic does not perform consistently because it biases the search toward states that are pending without allowing the rules that would resolve the pending state to be fired until much later in the search. The state of a cache memory will waver between quiescent, pending, quiescent, pending, quiescent, etc. This is because new requests that cause an eviction will move the state from quiescent to pending, and likewise responses from the memory controller will move the state from pending to quiescent again.

The min-max-predict heuristic chooses either the min-hamming or max-hamming heuristic depending on a prediction from a small saturating counter. It allows the search to move from a region of the graph to another and switch heuristics to best suit the more local search criteria based on the scoring function from the cache-score heuristic. In effect, it uses the cache state information to make a locally informed search decision in hopes to yield an effective global search result. Figure 4 shows a graph of the counter value and score from the initial state to the error state. From Table 2(a) and (b) we see that min-max-predict is consistently vastly better than BFS and DFS. This improvement is often several orders of magnitude better for the Cray X1 protocol! Contrary to the findings of Yang and Dill [11], we found that min-hamming performed better on only two of the six errors for the DASH protocol.

For the min-max-predict heuristic we experimented with 2-bit, 3-bit and 4-bit counters in the predictor. We found that a 3-bit counter worked best for the Cray X1 protocol and a 4-bit counter worked best for the DASH protocol.

Figure 4: A plot of counter and score values as the search for the Cray X1 embedded Error1 progresses from the initial state to error discovery.



Perhaps a small-valued counter in the range [0,12], for instance, would yield a nice compromise for both protocols. We believe the min-max-predict search heuristic would be suitable for verifying most cache coherence protocols and could be applied to other domains.

6 Related Work

Yang and Dill [11] examined the benefit of minimum hamming distance to improve a bread-first search in the hope that states with very few bits differeing from the target will require very few steps to reach the error. They use a technique called "target enlargement" to expand the size of the error states to all states that are 1 search step away from the error state. They concluded that minimum hamming distance could reduce the number of states explored, however, its performance was very inconsistent across different designs. Yang and Dill also studied a "guideposts" heuristic similar in concept to our cache-score heuristic. They use target enlargement in combination with guideposts to further reduce the states searched. All the heuristics were compared to a baseline BFS. In contrast, our study shows that most of the time DFS performed better than BFS. Yang and Dill explore only a single error on five designs: four from the memory controller of the Stanford FLASH multiprocessor and one from the link-layer communication protocol used by the Sun S3.mp multiprocessor. Yang and Dill only explore minimum hamming distance ignoring any possible value of maximum hamming distance.

Yuan, et. al. [14] use retrograde analysis to combine symbolic verification with simulation. They found hamming distance to be a useful heuristic to reduce the number of simulation trials needed to reach an enlarged set of errors called the pre-image of the error. Our approach does not require computing the enlarged target error states.

7 Conclusion

A detailed formal verification of a highly concurrent cache coherence protocol, such as that used by the

Cray X1, yields a state space with 214 million reachable states. We constructed a formal verifier called ws-Murφ which can explore approximately 300 states per second, whereas an RTL logic simulation of the coherence protocol running in Synopsys VCS can execute about 10 states per second. We evaluated several search heuristics with the goal of reducing the number of explored states compared to depth-first search (DFS) and breadth-first search (BFS). Our min-max-predict consistently performs better than DFS or BFS. We are most concerned with improved performance compared to DFS since this will produce witness strings that are executed with a "best-first" policy. Being able to cull the "best" witness strings from a voluminous protocol state space can reduce the error discovery time in simulation by several hours. We suspect the min-max-predict search heuristic would be suitable for verifying most cache coherence protocols and could be applied to other domains.

References

[1] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. The complexity of verifying memory coherence. In *Proceedings of the 15th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'03)*, pages 254–255. ACM, ACM Press, June 2003.

[2] D. L. Dill, A, J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design, VLSI in Computers and Processors*, pages 522–525, Los Alamitos, Ca., USA, 1992. IEEE Computer Society Press.

[3] Dennis Abts, Steve Scott, and David J. Lilja. So many states, so little time: Verifying memory coherence in the Cray X1. In *Proceedings o fthe 17th International Parallel and Distributed Processing Symposium (IPDPS03)*, April 2003.

[4] Thomas A. Henzinger, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. An assume-guarantee rule for checking simulation. In *ACM Transactions on Programming Languages and Systems*, pages 51–64. UC Berkeley, Jan 2002.

[5] M. Abadi and Leslie Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–535, 1995.

[6] Kenneth L. McMillan. A compositional rule for hardware design refinement. In *Proceedings of the 9th International Conference on Computer Aided Verification*, pages 24–35. Springer-Verlag, 1997.

[7] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Form. Methods Syst. Des.*, 15(1):7–48, 1999.

[8] C. N. Ip and D. L. Dill. Better verification through symmetry. In David Agnew, Luc Claesen, and Raul Camposano, editors, *Proceedings of the 11th International Conference on Computer Hardware Description Languages and their Applications (CHDL'93)*, volume 32 of *IFIP Transactions A: Computer Science and Technology*, pages 97–112, Amsterdam, The Netherlands, 1993. North-Holland.

[9] Dennis Abts and Mike Roberts. Verifying large-scale multiprocessors using an abstract verification environment. In *Proceedings of the 36th Design Automation Conference (DAC99)*, pages 163–168, June 1999.

[10] R. W. Hamming. Error detecting and correcting codes. Technical Report Vol 29, Bell Laboratories Technical Journal, pp. 147-160, 1950.

[11] C. Han Yang and David L. Dill. Validation with guided search of the state space. In *Proceedings of the 35th Annual Design Automation Conference (DAC98)*, June 1998.

[12] D. E. Lenoski. The directory-based cache coherence protocol for the DASH multiprocessor. *Proc of the 17th Annual Int. Symposium on Computer Architecture*, pages 148–159, June 1990.

[13] James Laudon and Daniel Lenoski. The SGI origin: A cc-NUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 241–251, New York, 2–4 1997. ACM Press.

[14] Jun Yuan, Jian Shen, Jacob Abraham, and Adnan Aziz. On combining formal and informal verification. In *Proceedings of 1997 Conference on Computer Aided Verification (CAV1997)*, pages 376–387, 1997.

The Design Complexity of Program Undo Support in a General-Purpose Processor

Radu Teodorescu and Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

1 Introduction

Several recently-proposed architectural techniques require speculation over long program sections. Examples of such techniques are thread-level speculation [4, 6, 11, 12], speculation on collision-free synchronization [7, 10], speculation on the values of invalidated cache lines [5], speculation on conforming to a memory consistency model [3], and even speculation on the lack of software bugs [8, 13].

In these techniques, as a thread executes speculatively, the architecture has to buffer the memory state that the thread is generating. Such state can potentially be quite large. If the speculation is shown to be correct, the architecture commits the speculative state. If, instead, the speculation is shown to be incorrect, the speculative state is discarded and the program is rolled back to the beginning of the speculative execution.

A common way to support these operations with low overhead is to take a checkpoint when entering speculative execution and buffer the speculative state in the cache. If the speculation is shown to be correct, the state in the cache is merged with the rest of the program state. If the speculation is shown to be incorrect, the speculative state buffered in the cache is invalidated and the register checkpoint is restored.

While the hardware needed for these operations has been discussed in many papers, it has not been implemented before. In fact, there is some concern that the hardware complexity may be too high to be cost effective.

In this paper, we set out to build such architectural support on a simple processor and prototype it using FPGA (Field Programmable Gate Array) technology. The prototype implements register checkpointing and restoration, speculative state buffering in the L1 cache for later commit or discarding, and instructions for transitioning between speculative and non-speculative execution modes. The result is a processor that can cleanly roll back (or "undo") a long section of a program.

We estimate the design complexity of adding the hardware support for speculative execution and rollback using three metrics. The first one is the hardware overhead in terms of logic blocks and memory structures. The second one is development time, measured as the time spent designing, implementing and testing the hardware extensions that we add. Finally, the third metric is the number of lines of VHDL code used to implement these extensions.

For our prototype, we modified LEON2 [2], a synthesizable VHDL implementation of a 32-bit processor compliant with the SPARC V8 architecture. We mapped the modified processor to a Xilinx Virtex-II FPGA chip on a dedicated development board. This allowed us to run several applications, including a version of Linux.

Our measurements show that the complexity of supporting program rollback over long code sections is very modest. The hardware required amounts to an average of less than 4.5% of the logic blocks in the simple processor analyzed. Moreover, the time spent designing, implementing, and debugging the hardware support is only about 20% higher than adding write back support to a write-through cache. Finally, the VHDL code written to implement our hardware adds about 14.5% more code to the data cache controller, and 7.5% to the simple pipeline.

This paper is organized as follows: Section 2 outlines the implementation; Section 3 estimates the complexity in terms of hardware overhead, rough development time, and lines of code; and Section 4 concludes.

2 Implementation

In order to support lightweight rollback and replay over relatively long code sections, we need to implement two main extensions to a simple processor: (1) modify the cache to also buffer speculative data and support rollback and (2) add support for register checkpointing and rollback. This allows a retiring speculative instruction to store the speculative datum that it generates into the cache, and ensures that the register state of the processor before speculation can be restored in case of a rollback request. We now describe both extensions in some detail. We also show how the transitions between non-speculative and speculative execution modes are controlled by software.

2.1 Data cache with rollback support

In order to allow the rollback of speculative instructions, we need to make sure that the data they generate can be invalidated if necessary. To this end, we keep the speculative data (the data generated by the system while executing in speculative mode) in the cache, and do not allow it to change the memory state. To avoid a costly cache flush when transitioning between execution modes, the cache must be able to hold both speculative and non-speculative data at the same time. For this, we add a single *Speculative* bit per cache line. If the Speculative bit is 0, the line does not contain speculative data. Otherwise, the line contains speculative data, and the non-speculative version of the line is in memory.

In addition to the Speculative bit, we extend the cache controller with a Cache Walk State Machine (CWSM) that is responsible for traversing the cache and clearing the Speculative bit (in case of a successful commit) or invalidating the lines with the Speculative bit set (in case of a rollback).

The Speculative bit is stored at line granularity. Therefore, while the processor is in speculative mode, for every write hit we check if the line we are writing to contains non-speculative, dirty data. If it does, we write back the dirty data, update the line, and then set the Speculative bit. From this point on, the line is speculative and will be invalidated in case of a rollback.

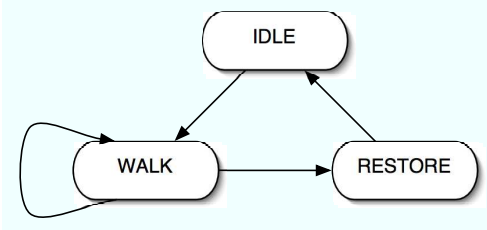


Figure 1. Cache Walk State Machine.

While in speculative mode, if a line is about to be evicted, we first check if it is speculative. If it is, we choose a non-speculative line in the same set for eviction. If none exists, we end the speculative section. In this initial version of our prototype, we commit the section at this point.

The Cache Walk State Machine (CWSM) is used to traverse the entire data cache and either commit or invalidate the speculative data. The state machine is activated when a commit or rollback instruction (Section 2.3) reaches the Memory stage of the pipeline. The pipeline is stalled and the cache controller transfers control to the CWSM. The CWSM has three states as shown in Figure 1.

In case of commit, the CWSM uses the Walk state to traverse the cache and clear the Speculative bits, effectively merging the speculative and non-speculative data. The traversal takes one cycle for each line in the cache. In the case of rollback, the CWSM is called to invalidate all the speculative lines in the cache. This means traversing the cache and checking the Speculative bit for each line. If the line contains speculative data, the Speculative and Valid bits are cleared.

2.2 Register checkpointing and rollback

Before transitioning to speculative state, we must ensure that the processor can be rolled back to the current, non-speculative state. Consequently, we checkpoint the register file. This is done using a Shadow Register File (SRF), a structure identical to the main register file. Before entering speculative execution, the pipeline is notified that a checkpoint needs to be taken. The pipeline stalls and control is passed to the Regis-

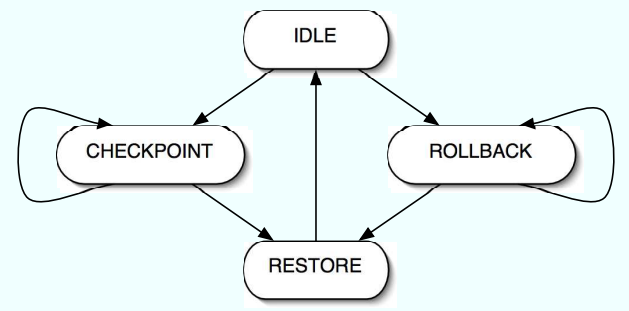


Figure 2. Register Checkpointing State Machine.

ter Checkpointing State Machine (RCSM). The RCSM has four states as shown in Figure 2, and is responsible for coordinating the checkpoint.

The RCSM is in the Idle state while the pipeline is executing normally. A transition to the Checkpoint state occurs before the processor moves to speculative mode. While in this state, the valid registers in the main register file are copied to the SRF. The register file is implemented in SRAM and has two read ports and one write port. This means that we can only copy one register per cycle. Thus the checkpoint stage takes as many cycles as there are valid registers in the register file. In addition, we need one cycle for all the status, control and global registers. These are not included in the SRF and can all be copied in one cycle.

The Rollback state is activated when the pipeline receives a rollback signal. While in this state, the contents of the register file is restored from the checkpoint, along with the status and global registers. Similarly, this takes as many cycles as there are valid registers, plus one.

2.3 Controlling speculative execution

There are several possible approaches to control when to enter and exit speculative execution. One approach is to have instructions that explicitly mark the beginning and end of speculative execution. A second approach is to use certain hardware events to trigger the beginning and the end of speculative execution. Finally, it is possible to implicitly keep the processor always in speculative mode, by using a hardware-managed "sliding window" of speculative instructions. In this prototype, we have implemented the first approach.

2.3.1 Enabling speculative execution

The transition to speculative execution is triggered by a LDA (Load Word from Alternate Space) instruction with a dedicated ASI (Address Space Identifier). These are instructions introduced in the SPARC architecture to give special access to memory (for instance, access to the tag memory of the cache). We extended the address space of these instructions to give us software control over the speculative execution.

The special load is allowed to reach the Memory stage of the pipeline. The cache controller detects, initializes and coordinates the transition to speculative execution. This is done at this stage rather than at Decode because, at this point, all non-speculative instructions have been committed or are about to finish the Write Back stage. This means that, from this point on, any data written to registers or to the data cache is speculative and can be marked as such.

The cache controller signals the pipeline to start register checkpointing. Interrupts are disabled to prevent any OS intervention while checkpointing is in progress. Control is transferred to the RCSM, which is responsible for saving the processor status registers, the global registers, and the used part of register file.

When this is finished, the pipeline sends a *checkpointing complete* signal to the cache controller. The cache controller sets its state to speculative. Next, the pipeline is released and execution resumes. From this point on, any new data written to the cache is marked as speculative.

2.3.2 Exiting speculative execution

Speculative execution can end either with a commit, which merges the speculative and non-speculative states or with a rollback in case some event that requires an "undo" is encountered. Both cases are triggered by a LDA instruction with a dedicated ASI. The distinction between the two is made through the value stored in the address register of the instruction.

An LDA from address 0 causes a commit. In this case, the pipeline allows the load to reach the Memory stage. At that point, the cache controller takes over,

stalls the pipeline, and passes control to the CWSM. The CWSM is responsible for traversing the cache and resetting the Speculative bit. When the cache walk is complete, the pipeline is released and execution can continue non-speculatively.

An LDA from any other address triggers a rollback. When the load reaches the Memory stage, the cache controller stalls the pipeline and control goes to the RCSM. The register file, global and status registers are restored. The nextPC is set to the saved PC. A signal is sent to the cache controller when rollback is done. At the same time, the cache controller uses the CWSM to traverse the cache, invalidating speculative lines and resetting the speculative bits. When both the register restore and cache invalidation are done, the execution can resume.

3 Evaluation

3.1 Experimental infrastructure

As a platform for our experiments, we used LEON2 [2], a synthesizable VHDL implementation of a 32-bit processor compliant with the SPARC V8 architecture.

The processor has an in-order, single-issue, five stage pipeline (Fetch, Decode, Execute, Memory and Write Back). Most instructions take 5 cycles to complete if no stalls occur. The Decode and Execute stages are multi-cycle and can take up to 3 cycles each.

The data cache can be configured as direct mapped or as multi-set with associativity of up to 4, implementing least-recently used (LRU) replacement policy. The set size is configurable to 1-64 KBytes and divided into cache lines of 16-32 bytes. The processor is part of a system-on-a-chip infrastructure that includes a synthesizable SDRAM controller, PCI and Ethernet interfaces. The system is synthesized using Xilinx ISE v6.1.03. The target FPGA chip is a Xilinx Virtex II XC2V3000 running on a GR-PCI-XC2V development board [9]. The board has 8MB of FLASH PROM and 64 MB SDRAM. Communication with the device, loading of programs in memory, and control of the development board are all done through the PCI interface from a host computer.

On this hardware we run a special version of the SnapGear Embedded Linux distribution [1]. SnapGear Linux is a full source package, containing kernel, libraries and application code for rapid development of embedded Linux systems. A cross-compilation toolchain for the SPARC architecture is used for the compilation of the kernel and applications.

3.2 Estimating design complexity

We estimate the design complexity of adding our hardware extensions using three metrics: the hardware overhead in terms of logic blocks and memory structures, implementation time, and VHDL code size.

3.2.1 Hardware overhead

One approach to estimating the complexity of our design is to look at the hardware overhead imposed by our scheme. We synthesize the processor core, including the cache. We look at the utilization of two main resources: Configurable Logic Blocks (CLBs) and SelectRAM memory blocks.

The Virtex II CLBs are organized in an array and are used to build the combinational and synchronous logic components of the design. Each CLB element is tied to a switch matrix to access the general routing matrix. A CLB element comprises 4 similar slices. Each slice includes two 4-input function generators, carry logic, arithmetic logic gates, wide-function multiplexers and two storage elements. Each 4-input function generator is programmable as a 4-input lookup table (LUT), 16 bits of distributed SelectRAM memory, or a 16-bit variable-tap shift register element.

The SelectRAM memory blocks are 18 Kbit, dual-port RAMs with two independently-clocked and independently-controlled synchronous ports that access a common storage area. Both ports are functionally identical. The SelectRAM block supports various configurations, including single- and dual-port RAM and various data/address aspect ratios. These devices are used to implement the large memory structures in our system (data and instruction caches, the register file, shadow register file, etc).

We also measure the hardware overhead introduced by implementing a write-back cache controller. The original LEON2 processor has a write-through data cache. Since our system needs the ability to buffer speculative data in the cache, a write back cache is needed. We implement it by modifying the existing

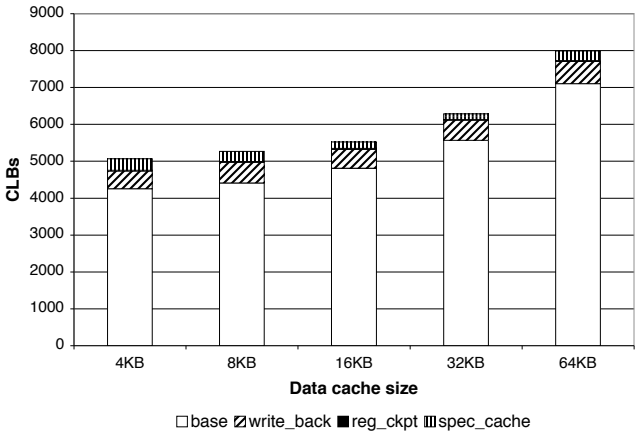


Figure 3. Number of CLBs used by different hardware structures. Each bar corresponds to a core with a different cache size.

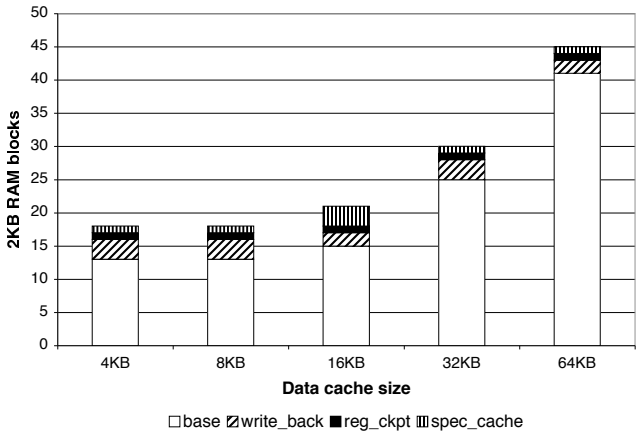


Figure 4. Number of SelectRAM blocks used by different hardware structures. Each bar corresponds to a core with a different cache size.

controller. This allows us to compare the implementation complexity of the hardware we propose with the complexity of modifying a write-through cache into a write-back one.

Figure 3 shows a breakdown of the number of CLBs used by the processor core and each of the main extensions added to it. Each bar corresponds to a core with a different data cache size. The *base* represents the size of the original processor core with a write-through cache; the *write_back* represents the overhead of adding the write back cache; the *reg_ckpt* represents the register checkpointing mechanism; and finally, the *spec_cache* represents the cache support for speculative data.

The CLB overhead of adding program rollback support (*reg_ckpt* plus *spec_cache*) to a processor is small (less than 4.5% on average) and relatively constant across the range of cache sizes that we tested. This overhead is computed with respect to the processor *with* the write back data cache controller (*base* plus *write_back*), a configuration typical for most current processors.

We notice that the hardware overhead introduced by the register checkpointing support is very small compared to the rollback support in the cache. This is due to a simple design of the register checkpointing state machine which requires less state information and fewer control signals. Also, the overhead of adding the write back cache controller is larger than that of adding the full support for speculative execution.

Figure 4 shows a comparison between the same configurations, but looking at the number of SelectRAM blocks utilized. We see that the amount of extra storage space necessary for our additions is small across the five configurations that we evaluated.

3.2.2 Design, implementation and testing time

Another complexity indicator is the time spent designing, implementing and debugging the hardware. We consider the three major components of our design, namely the speculative cache, the register checkpointing and the software control support. We compare the time spent developing them with the time spent developing the write back cache controller.

The estimates are shown in Figure 5. We note that out of the three extensions, the speculative cache took the longest to develop. Overall, the time spent designing, implementing, and debugging our hardware support is only about 20% higher than adding write back support to a write-through cache.

3.2.3 Lines of VHDL code

The third measure of complexity we use is VHDL code size. The processor is implemented in a fully synthe-

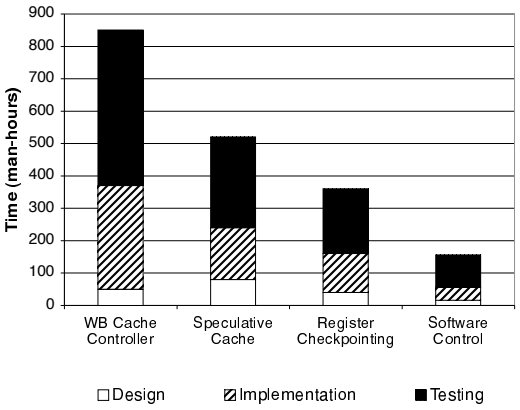


Figure 5. Estimates of the design, implementation and testing time (in man hours) for individual components in our system.

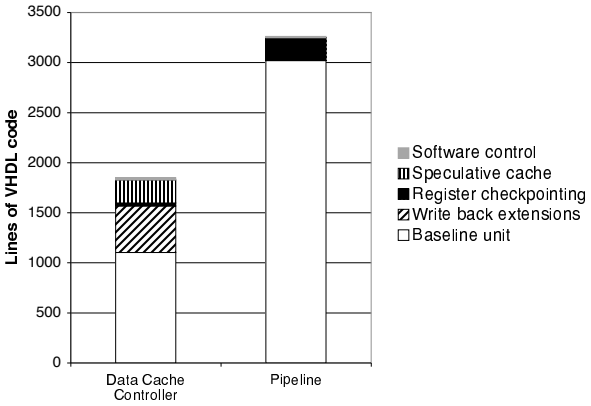


Figure 6. Breakdown of the lines of VHDL code in the data cache controller and the pipeline.

sizable RTL description that provides sufficient details to make code size a good indication of design complexity.

We look at the number of lines of VHDL code needed to implement our hardware extensions in the two main processor components that we modified: the data cache controller and the pipeline. The results are shown in Figure 6. The bars show a breakdown of the lines of code for the two components. We also include data about the write back support in the cache controller.

We note that the code needed to implement our extensions is small. The results are consistent with the previous two experiments. The write back cache controller is the most complex to implement, accounting for as much code as all the other extensions combined. The VHDL code written to implement our hardware adds about 14.5% more code to the data cache controller, and 7.5% to the simple pipeline.

4 Discussion

While our conclusions have to be qualified by the fact that we are dealing with a simple processor, our working prototype has given us good insights into the complexity of developing hardware support for program rollback.

Our analysis shows that the complexity of supporting program rollback over long code sections is very modest. The hardware required amounts to an average of less than 4.5% of the logic blocks in the simple processor analyzed. Moreover, the time spent designing, implementing, and debugging the hardware support is only about 20% higher than adding write back support to a write-through cache. Finally, the VHDL code written to implement our hardware adds about 14.5% more code to the data cache controller, and 7.5% to the pipeline.

Considering that the hardware support described can be used in many novel speculative techniques (Section 1), we argue that it is complexity effective.

References

[1] CyberGuard. Snapgear embedded linux distribution. www.snapgear.org.

[2] J. Gaisler. LEON2 Processor. www.gaisler.com.

[3] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 162–171. IEEE Computer Society, 1999.

[4] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[5] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: Making use of incoherence. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97–106. ACM Press, 2004.

[6] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, pages 866–880, September 1999.

[7] J. F. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 18–29. ACM Press, 2002.

[8] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 184–196, Oct. 2002.

[9] R. Pender. Pender Electonic Design. www.pender.ch.

[10] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO)*, pages 294–305, Austin, TX, Dec. 2001.

[11] G. Sohi, S. Breach, and T. Vijayakumar. Multiscalar Processors. In *22nd International Symposium on Computer Architecture*, pages 414–425, June 1995.

[12] J. G. Steffan and T. C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *High Performance Computer Architecture (HPCA)*, February 1998.

[13] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architecture Support for Software Debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 224–237, June 2004.