Workshop on Complexity−Effective Design

Held in conjunction with the 33rd International Symposium on Computer Architecture

Boston, Massachusetts
June 18, 2006

Workshop Organizers:

Dave Albonesi, Cornell University
Pradip Bose, IBM Corporation
Prabhakar Kudva, IBM Corporation
Diana Marculescu, Carnegie−Mellon University

WCED Program

8:00-8:30AM Breakfast
8:30-9:30AM Session I

- Printed Circuit Board Layout Time Estimation
  C. Bazeghi and J. Renau (University of California, Santa Cruz)

- Accommodating Workload Diversity in Chip Multiprocessors via Adaptive Core Fusion
  E. İpek, M. Kırman, N. Kırman, and J. Martínez (Cornell University)

- Designing Hardware that Supports Cycle-Accurate Deterministic Replay
  B. Greskamp, S.R. Sarangi, and J. Torrellas (University of Illinois)

9:30-9:45AM Short Break
9:45-10:45AM Session II

- Leveraging Bloom Filters for Smart Search Within NUCA Caches
  R. Ricci, S. Barrus, D. Gebhardt, and R. Balasubramonian (University of Utah)

- RegionTracker: Using Dual-Grain Tracking for Energy Efficient Cache Lookup
  J. Zebchuk and A. Moshovos (University of Toronto)

- Using Speculation Cost Predictability in Low-Power Cost-Aware Branch Prediction
  E. Atoofian, A. Baniasadi, and F. Khosrow-Khavar (University of Victoria)

10:45-11:15AM Break
11:15AM-12:30PM Panel Discussion: “Lost in the Bermuda Triangle: Complexity vs. Energy vs. Performance”

Introduction

The quest for higher performance via deep pipelining, speculative and multi-threaded execution, and chip-multiprocessing, has yielded microprocessors with greater performance, but at the expense of 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 Evelyn Duesterwald, the Workshops Chair, and the other ISCA organizers that allowed us to offer the workshop, the WCED Program Committee (Dennis Abts, R. Iris Bahar, David Brooks, Alper Buyuktosunoglu, George Cai, Babak Falsafi, Keith Farkas, Antonio Gonzalez, Peter Hofstee, Gokhan Memik, Chuck Moore, and Jose Renau), 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

Printed Circuit Board Layout Time Estimation

Cyrus Bazeghi and Jose Renau
University of California, Santa Cruz

ABSTRACT

System design complexity is growing rapidly. As a result, current development costs can be staggering and are constantly increasing. As designers produce ever larger and more complex systems, it is becoming increasingly difficult to estimate how much time it will take to design and verify these designs. To compound this problem, system design cost estimation still does not have a quantitative approach. Although designing a system is very resource consuming, there is little work invested in measuring, understanding, and estimating the effort required.

To address part of the current shortcomings, this paper introduces *μPCBComplexity*, a methodology to measure and estimate PCB (printed circuit board) design effort. PCBs are the central component of any system and can require large amounts of resources to properly design and verify. *μPCBComplexity* consists of two main parts, a procedure to account for the contributions of the different elements in the design, which is coupled with a non-linear statistical regression of experimental measures. We use *μPCBComplexity* to evaluate a series of design effort estimators on several PCB designs. By using the proposed *μPCBComplexity* metric, designers can estimate PCB design effort.

1 Introduction

Printed circuit board (PCB) design effort keeps growing as additional constraints such as rising clock frequencies, reduced area, increasing number of layers, mixed signal devices, and the ever increase in component numbers and densities. All of these factors combined have led to a steady rate of increase in development costs for current systems. As we design ever larger, denser and more complex systems, it is becoming increasingly difficult to estimate how much time would be required to design and verify them. To compound this problem, PCB design effort estimation still does not have a quantitative approach. We present in this paper a first step toward creating a design effort metric that is highly correlated with design effort for PCB layout. We follow the same approach taken in [1] as the principles that are applicable to microprocessors are also applicable to PCBs. In this paper, design effort corresponds to the number of engineering-hours required for implementation (layout) of a PCB design.

This work was supported in part by the National Science Foundation under grants 0546819; Special Research Grant from the University of California, Santa Cruz; and gifts from SUN.

This paper analyzes and proposes various statistics to estimate the layout effort required to develop PCBs. We investigate and quantify statistics such as area, component count, pin count and device types and sizes for many PCBs. We analyze several of these statistics, and propose a metric, obtained after applying non-linear regression over the different statistics, which we call *μPCBComplexity*. In addition, we provide insights on the correlation between several statistics and design effort for several known layout design times.

Different designs have different constraints, leading to specific challenges; typical design constraints being area, frequency, and cost. For example, having area being a primary design constraint, may lead to a requirement for additional layers, more expensive package types, and more complex placement and routing. A design constrained by cost, on the other hand, may require a balance between number of layers, area, drill density, types of packages and possibly the number of different drill sizes. Having clear constraints is necessary in estimating layout effort as it can drastically affect complexity.

We define design effort to be the layout time required by one engineer. Design effort is equivalent to layout time when the project has a single developer, which is frequent even for complex PCBs. Nevertheless, for a given effort requirement, it is possible to reduce the design time by increasing the number of workers. Nevertheless, increasing the number of workers decreases the productivity per worker. The relationship between these two elements has been widely studied in software metrics and business models. Since the conversion between design effort and design time can be approximated, the remainder of this paper focuses only on design effort.

The rest of the paper is organized as follows. Section 2 covers other work in this area; Section 3 describes the statistical techniques that allow us to calibrate and evaluate the *μPCBComplexity* regression model; Section 4 describes the setup for our evaluation; Section 5 evaluates several statistics for the boards in our analysis; and Section 6 presents conclusions and future work.

2 Related Work

The capability to rapidly develop complex PCBs is a tremendous competitive advantage, since high development productivity is essential for the success of any design team. Although some companies have used statistical methods to estimate PCB design time, those methods are considered trade secrets [9]. Other companies do not release details because

they provide competitive advantage over other companies. As a result, we are unaware of any published work on the topic of predicting the engineering hours required for a PCB design.

[1] focuses on microprocessor design effort. While the work described in this paper focuses on PCB design metrics, [1] uses the same regression model, but both papers analyze different set of statistics and targets.

Another paper that looks at productivity is [7] which identifies the need for standards or infrastructures for measuring and recording the semiconductor design process. They propose improving design technology, time-to-market, and quality-of-result by addressing the Design Productivity Gap and the Design "Technology" Productivity Gap. However, this previous work focused mostly on the problems associated with the infrastructure and design tools related to the physical implementation of semiconductor designs, while the focus of this paper is layout effort associated with PCB designs.

In [8] a factor similar to the productivity factor is described. They use the "process productivity parameter" to tune the estimating process for software projects. They contend that if you know the size, time, and the process productivity parameter you can use it to make estimates for a new project. So long as the environment, tools, methods, practices, and skills of the people have not changed dramatically from one project to the next.

Much research has been done in Design for Manufacturing (DFM) and Design for Production (DFP) which seek to improve the production and manufacturing times of PCB assemblies. This paper seeks to develop a metric that can aid in predicting the layout effort, based on analysis of characteristics of PCBs at a low-level so as to better plan for future generations of systems. In [2] the issue of embedded passive components is discussed as a necessity to the smaller electronic devices requiring ever smaller PCBs. They note that board area is becoming so critical that to keep pace with the size constraints new techniques are required. Our goal would be to eventually develop a set of metrics and a model that estimates design effort by also taking into account manufacturing times.

3 Approach

Our goal is to develop a quantitative approach and to have a model that quickly estimates design effort based on several easily gathered statistics. This is important because being able to predict design effort is advantageous in helping to reduce design costs. To build the model, we analyze many commercial computer/electronic devices and gather data from the PCBs within. The layout times for these PCBs were well documented which was a requirement for this analysis. Table 1 lists the critical components of PCB designs as determined by [2]. These parameters contribute to the complexity of a design, and hence the time required to do layout.

Some design parameters listed in Table 1 are dependent on other factors. For example, the size of the board is defined by the number of embedded and discrete passive components and total wiring requirements. However, the total wiring requirements are governed by the number of embedded and discrete passive components in the PCB. And further more, the total number of layers in the PCB depends on the size of the board, the number of embedded and discrete resistors and bypass capacitors [2].

1.	Board dimensions (length and breadth)
2.	Total wiring requirements
3.	Number of layers
4.	Number of embedded resistors (if used)
5.	Number of embedded capacitors (if used)
6.	Set of active component types and their number
7.	Thickness of the board
8.	Number of discrete resistors
9.	Number of discrete capacitors

Table 1: Critical design parameters for a PCB.

These critical design parameters are focused towards manufacturability, not design effort estimation. We used them as a starting point in determining what parameters or metrics to analyze and include for correlation with design effort. None of the boards in our study have embedded passive components, instead we focus on the total number of all components (passive and discrete) and the pin count for them. These are easily obtainable values.

Since the routing data is not easily obtainable, the number of pins for all the components in the design are taken into account instead. While this is not an ideal metric since not all pins are used or have very short traces (VDD or GND), it is readily obtainable an does not hamper the focus of this paper, namely effort prediction starting from higher level design descriptions, such as a bill of materials (BOM) or schematics.

In order to find a metric highly correlated with design effort, several statistics were gathered from the existing designs. For each isolated board with a known design effort, we look at several statistics and apply non-linear regression to find a highly correlated metric.

We present our design effort model as the aggregate of a set of statistics (S_i). Each of which has a specific constant (w_i), associated with it, which assigns a weight to the importance of every statistic used as input in the model. The aggregate of the statistics is inversely proportional to the productivity of a specific design team which is represented by a constant (ρ). The model is presented in Equation 1. In order to find suitable values for each of the data weights (w_i) we perform mixed non-linear regressions on this equation. The design team productivity factor (ρ) is constant per design group, and it needs to be adjusted on a per company or design team basis. If the ρ is unknown, then the absolute design effort is invalid and only the breakdown inside the project is correct. Obtaining the value of ρ is simple; all that is needed is to have the design effort for a single project. Alternatively, it is possible to develop a productivity benchmark suite that calibrates ρ for a given company.

$$\text{Design Effort} = \frac{1}{\rho} \times \sum_{k=1}^{n} (w_k \times S_k) \quad (1)$$

In order to determine the weights that give a generalized solution to Equation 1, [1] proposes to use a mixed non-linear regression model. If there are no productivity adjustments, it is possible to use a simpler non-linear regression model. While the sum of a large number of random variables is distributed normally, the product of a number of random variables is distributed *lognormally* — a distribution where the logarithm of the variable is normally distributed [4]. Therefore, since the random variables have a log normal distribution an even simpler linear regression model can not be used.

To evaluate the accuracy of the model (Section 5), we use σ as a measure of error associated with the fit. Consequently, it is important to understand what different values of σ tell us about the quality of the estimate. For a given σ, we can find a *confidence interval* for the estimated effort. The x% confidence interval for a metric is defined to be the range of efforts $(Estimate_{low}, Estimate_{high})$ such that $P(Estimate_{low} <$ metric prediction $< Estimate_{high}) = x/100$. For example, the 90% confidence interval gives us two values a and b such that there is a 90% chance that the actual effort is between metric prediction $\times\ a$ and metric prediction $\times\ b$.

3.1 Productivity Adjustments

In software development projects, it is well known that different development teams have different productivities. For example, it has been shown that the productivity difference between teams can be up to an order of magnitude [5]. We believe that a similar effect occurs between PCB design teams. The productivity differences may be due to multiple factors, including the average experience of the designers in the team and the tools used. In our model, ρ captures this effect.

The boards under study in this analysis all come from one manufacturer and so the use of a productivity factor was not necessary.

3.2 Team Size Dynamics

Although some board designs require long periods of time, it is very rare to find multiple developers doing different sections of the same board. The PCB layout effort by nature is a linear task done by one engineer at a time. To reduce the design time, we have found two approaches: multi-timezone working environments, and "surgical" teams.

A multi-timezone team has different designers working on multiple time zones, this is, once a designer stops working a new designer can continue and pick up where the previous designer left. A "surgical team" [6] follows an alternative design organization, with the surgeon, or chief designer, at the helm and a supporting staff that has their tasks allocated by the chief of staff. In the PCB case, we may have other designers doing such tasks as making footprint images for components, which can be a tedious effort.

3.3 R-Language

This section provides the R-language [10] code to fit the non-linear mixed-effects model and the non-linear regression model. The mixed-effects model is needed when productivity adjustments (ρ) are required, a simpler model is used when no productivity adjustments are required.

Recall that our model has a multiplicative lognormal error and also a lognormal distribution for the random effect ρ. Simply taking the logarithm of both sides of the equation gives us the requisite additive normal error and normal random effect as follows. Hence the need for a non-linear model.

```
# mixed-effects non-linear model
nlme(model=log(Effort) ~
  (log_rho) + log(w1*stat1 + w2*stat2)
  ,random = log_rho ~ 1 | team
  ,fixed  = list(w1 ~ 1, w2 ~ 1)
  ,start = c(0.1, 0.1)
  ,data=(traw)
  ,method="ML")

# non-linear model
nls(log(Effort) ~
  log(w1*stat1 + w2*stat2)
  ,start=list(w1=0.1,w2=0.1)
  ,data=traw)
```

The R-language is also used to compute the confidence intervals. To obtain a 90% confidence interval for a given σ (s) generated, the following R-language code $c(exp(s * qnorm(0.05)), exp(s * qnorm(0.95)))$ is used.

4 Evaluation Setup

We gathered data from a number of PCB designs for the analysis done in this paper. Table 3 shows the types of statistics gathered for each of the boards analyzed. When calculating the area consumed for each component we did not consider the cases where routing, or in the more rare case placement, could be done underneath a component. Several board designers pointed out that the component and pin density of the board was one of the crucial factors to estimating design effort. To capture component and pin density, we define them with equation 2 and equation 3 respectively.

$$\text{Component Density} = \frac{\text{\# Components}}{\text{PCB Area} \times \text{\# Sides w/ components}} \quad (2)$$

$$\text{Pin Density} = \frac{\text{\# Pins}}{\text{(PCB Area)}} \quad (3)$$

Table 2 gives a description of the boards along with the engineering notes that we were able to gather from the designers. Boards B7-B11 used SPECCTRA for OrCAD which is a common autorouter used in industry. No data was available on the use of an autorouter for boards B1-B6 but it can be safely assumed that some autoroute tool was used.

In discussions with the designer of boards B8 and B9 the size of the LCD in the system dictated the size of the PCB and the housing that contained it. The LCD was counted as a

Board	Description	Engineering Notes
B1	Signal Conditioning	Many thru-hole components. Analog board with many important signal paths
B2	AE RMS	Many thru-hole components. Analog board with many important signal paths
B3	PMD Motor Controller	Many high density components
B4	Motor Driver	New footprints
B5	Enviro Controller	Forgot reasons why it took so long
B6	Current Source	Many components on a small board. Mechanical constraints
B7	Arbitrary Waveform Generator/Amplifier	Placement constraints due to noise reduction
B8	ACDC Monitor	Cost major factor. Time consuming to keep to a 2 layer board
B9	Tank Monitor	Cost major factor. Time consuming to keep to a 2 layer board
B10	Air spring remote	Very small. RF constraints
B11	Air Spring Controller	2 Isolated grounds with placement constraints

Table 2: Description of boards analyzed.

Board Statistic	Description
PCB Size (mm^2)	Physical size of the PCB
# of Sides w/ Comp	Either 1 or 2 sides has components
# of Routing Layers	Layers used for routing traces
# of Layers	The total number of layers in the PCB
Components	
# Passive	Passive components (resistors...)
# Digital	Digital integrated circuits (IC)
# Analog	Analog ICs or devices (opamps...)
# Mixed Signal	ICs with both digital and analog sections
Total #	Total count of all components on PCB
Total Area (mm^2)	Total area of all components on PCB
Density	Ratio of component area to area
Pins	
# Passive	Pins for all passive components
# Digital	Pins for all digital components
# Analog	Pins for all analog components
# Mixed Signal	Pins for all mixed signal components
Total	Pins for all devices on PCB
Density	Ratio of number of pins to area

Table 3: Description of the statistics gathered from the PCBs.

component in our analysis and took one complete side of both these boards, forcing the placement and routing of all other components to one side. Cost was the main consideration for both these boards also and this forced the designer to route everything using only 2 layers.

Among boards B7 through B11 the smallest board, B10, was judged to be the most difficult to layout. Where as boards B7 and B11 were the easiest. This was attributed to the areas available to do the placement and routing. B7 and B11 were two of the largest boards reviewed and they were not area constrained, this gives much latitude to the designer for placement and makes the autorouter produce better results. With a more constrained area more human intervention is required during the routing phase which was the case for B10.

For the placement stage we only had to consider the number of sides of the board on which components were mounted. Most of the boards in this study had the components all on one side, though a few had bypass capacitors mounted on one side, which accounted for a negligible amount of space. Again, thru-hole devices would effect the available placement area as it did the available routing area as space would be lost on both sides of the board, unlike with surface mounted components. This was not a factor in this study since most boards only used one side for placement. Boards B8 and B9 had components on both sides but one side was populated by only one component, the LCD. Board B10, the only other board with components on both sides, did not have any thru-hole devices present.

5 Evaluation

Our evaluation analyzes 11 different printed circuit boards. Table 4 shows the main results and characteristics for each of these. The first column corresponds to each of the statistics or metrics presented in Table 3 (Section 4). Columns B1 to B11 correspond to each of the boards (Table 2). The last column corresponds to the σ between the row and design effort. Since all the boards are designed by the same team, we do not evaluate the productivity factor (ρ). This simplifies the analysis, and we can use non-linear regression instead of the mixed-effects non-linear regression model. With σ we can compute the confidence interval. For the lognormal distribution used, the mapping between σ and the 90% confidence interval is shown in Figure 1. We will use this chart to compare the accuracy of different estimators.

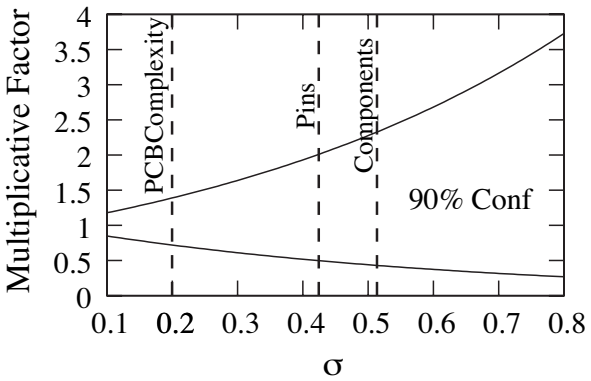

Figure 1: Mapping between the standard deviation of the error (σ) and the 90% confidence interval for the lognormal error distribution used.

The design effort values were obtained by interviewing the original designers. Obviously, there is perfect correlation with itself so $\sigma = 0$. A zero σ results in a perfect $(1, 1)$ confidence interval. We now proceed to analyze easily available statistics like number of components and pin count. These two sets of statistics are easily available before the PCB design starts. They are part of the PCB specification.

From the boards analyzed, we observe that it is best to

	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	σ
Design Effort (hours)	68	35	43	21	48	48	24	40	32	24	12	0
Components												
# Passive	213	165	101	80	108	222	116	86	83	19	47	0.52
# Digital	15	0	17	0	8	2	0	11	8	4	4	0.99
# Analog	24	24	1	10	24	50	28	4	16	1	11	1.10
# Mixed Signal	11	0	7	0	0	3	0	0	0	0	0	0.75
Total #	263	189	126	90	140	277	144	101	107	24	62	0.51
Total Area (mm2)	6214	9053	6964	2719	9144	6579	8104	12193	12296	777	5430	0.93
Pins												
Passive	563	429	365	182	414	578	414	194	188	39	109	0.62
Digital	154	0	518	0	107	32	0	175	173	88	32	2.18
Analog	188	208	8	98	72	400	150	25	53	14	65	1.19
Mixed Signal	172	0	208	0	0	48	0	0	0	0	0	2.00
Total	1077	637	1099	280	593	1058	564	394	414	141	206	0.43
PCB Size (mm2)	22194	22194	22194	16258	38710	20452	22194	10968	10968	1277	25548	0.52
# of Sides w/ Comp	1	1	1	1	1	1	1	2	2	2	1	3.17
# of Routing Layers	2	2	3	2	2	2	3	2	2	4	2	1.18
# of Layers	4	4	6	4	4	4	4	2	2	4	2	0.94
Component Density (x1000)	62	45	30	29	19	71	34	24	26	49	13	0.48
Pin Density	50	30	51	18	16	54	26	37	39	115	8	0.64
μPCBComplexity (hours)	60	44	44	16	37	57	32	35	33	24	13	0.2

Table 4: Statistics, design effort, and correlation results of study boards.

use the total number of components to estimate design effort ($\sigma = 0.51$). Although traces for analog components and digital components are more difficult than traces for passive components, the low amount of digital and/or analog components on several of the boards make it difficult to use them as a method to estimate effort. Figure 1 shows the confidence interval for a $\sigma = 0.51$ as the intersection between the components line and the confidence interval line $(0.43, 2.31)$. This means that using the number of components on the specification, we have a 90% confidence that the design effort would be between 0.43 and 2.31 times the prediction.

Statistics about the pins are as easily available as components even before the design starts. The number of pins is an even better predictor ($\sigma = 0.43$) than the number of components ($\sigma = 0.51$). The resulting 90% confidence interval for the number of pins is $(0.49, 2.03)$. This means that just by using the pins, we have a 90% confidence that the prediction is roughly half or double the expected design effort. Not shown in the table is the result of combining the number of pins and the components to predict design effort. The results did not improve because there is a high correlation between pins and components.

Area is an interesting statistic. Just by knowing the final dimension of the board, we can estimate design effort with a $(0.43, 2.35)$ confidence interval. This is roughly the same accuracy as the number of components. The reason is that PCBs are always area constrained [1]. If the specification provides a realistic area constraint, it could be a good way to estimate design effort. Table 4 also shows other statistics such as number of sides used, routing layers, and number of layers. Those statistics are not so useful by themselves because they are highly quantized, and this makes them difficult to use to predict effort.

The proposed *μPCBComplexity* metrics are now evaluated. To obtain *μPCBComplexity* shown in Table 4, we analyzed multiple combinations of parameters and followed suggestions from experienced board designers. The best results were achieved when using the following equation:

$$\text{Effort} \propto \text{\# Passive Comp.} + \text{Comp. Density} + \text{Pin Density} \quad (4)$$

Section 4 explains how to compute component density and pin density. To obtain the factors on equation 4, we perform non-linear regression as explained in Section 3. Although neither pin nor component density can achieve better predictions than the number of pins, when integrated together in the *μPCBComplexity* metric we achieve a 0.2 σ. As Figure 1 shows, this represents a $(0.72, 1.39)$ confidence interval. This roughly means that by using the proposed *μPCBComplexity* metrics, with a 90% confidence designers can predict design effort with less than 40% error.

Figure 2 shows a scatter-gather plot between design effort and our *μPCBComplexity* metric. This is an intuitive way to see that there is a high correlation between design effort and the metric proposed.

μPCBComplexity works well because PCB design complexity increases as the component and pin density increases. Designers can increase the number of layers on the PCB to decrease the pin density or increase the area to reduce both densities. The problem is that both approaches require more costly boards. As a result, designers trade-off between time to market and density.

6 Conclusions & Future Work

The goal of this paper was to explore the correlation of some easily obtained metrics of a PCB and see which were most correlated to the design effort required during the layout stage of development. Many simplifications were made; we did not

[1] Bigger PCBs have higher cost.

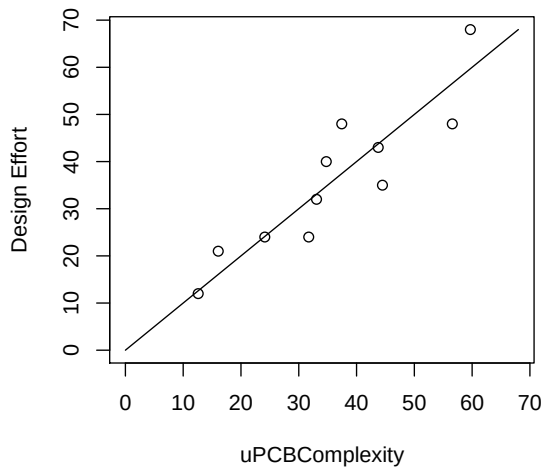

Figure 2: Scatter-gather plot of design effort vs. PCB metric

account for traces of differing sizes, we did not look at hole sizes or density, the frequency of the boards were not considered, nor the extra considerations required for analog noise filtering. Also, we need additional PCBs from more companies with teams of differing sizes to develop a more general model for predicting design effort.

Many factors and constraints effect the design effort required for a board to be successfully placed and routed. Some difficulty metric would be helpful but guidelines need to be established as difficulty is a very subjective term. Being able to analysis different options for a board would be useful, such as being able to change the size of the board to see what effect it would have on the estimated design effort. This could be expanded to also include the number of layers since this would ease routing congestion.

We see this initial research leading into more areas of study in PCB design optimization and analysis. We are currently analyzing data from additional PCB designs from different sources. These new designs have more components, more layers, higher frequencies, and more power plains. This will give us additional metrics to add to our model for possible better correlation to design effort. These designs also have more designers on the team which will necessitate some team or company productivity factor.

We have extended the previously proposed *μComplexity* models [1] to the PCB domain. We plan to apply the model to a number of classes at UCSC that do board development to give design guidelines to students and further refine our approach. Our model and metrics will eventually be available to researchers and industry for use in scheduling and planning PCB projects.

The evaluation shows that a simple statistics like PCB area size and number of components yield some correlation with design effort. With a 90% confidence, area has a (0.43 2.35) confidence interval. This means that roughly by looking at any of those statistics the typical design time error is half/double with a 90% confidence. Much better results can be achieved with the proposed *μPCBComplexity* metric. In that case the confidence interval for a 90% confidence is (0.72 1.39). This roughly means that less than 40% estimation error is done with a 90% confidence.

Despite the good results, we still believe that much work needs to be done in gathering relevant designs to evaluate (with associated known design times) and to refine the metrics and models. A major goal would be a rule of thumb type equation that given some easily obtainable design parameters an accurate estimator of design time would be generated.

Acknowledgments

We thank Martin Greatorex and Adam Harrison from Vena Engineering Corporation for providing design effort numbers and the detailed board data used in this study. Thanks also go to Truong Nguyen from SonicWALL, Inc for his insights on this project and for design data that we plan on using for a future paper in the works. Additional thanks go Brian Greskamp and Josep Torrellas of the university of Illinois at Urbana-Champaign for their help on the methods developed.

REFERENCES

[1] C. Bazeghi, F. Mesa-Martinez, and J. Renau. μComplexity: Estimating Processor Design Effort. In *International Symposium on Microarchitecture*, Nov 2005.

[2] M. Chincholkar and J. Herrmann. Modeling the impact of embedding passives on manufacturing system performance. September 2002.

[3] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum, 1988.

[4] E.L. Crow and K. Shimizu. *Lognormal Distributions: Theory and Application*. Dekker, 1988.

[5] T. DeMarco and T. Lister. *Peopleware Productive Projects and Teams*. Dorset House Publishing, 1999.

[6] JR. Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1995.

[7] A. B. Kahng. Design technology productivity in the dsm era (invited talk). In *Conference on Asia South Pacific Design Automation*, pages 443–448. ACM Press, 2001.

[8] L. H. Putnam and W. Myers. *Five Core Metrics: The Intelligence Behind Successful Software Management*. Dorset House Publishing, May 2003.

[9] Numetrics Management Systems. Design Complexity and Productivity. Technical report, Numetrics Management Systems, Inc., 2004. http://www.numetrics.com.

[10] The R Development Core Team. *The R Reference Manual - Base Package*. Network Theory Limited, 2005.

Accommodating Workload Diversity in Chip Multiprocessors via Adaptive Core Fusion

Engin İpek Meyrem Kırman Nevin Kırman José F. Martínez

Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA

http://m3.csl.cornell.edu/

ABSTRACT

This paper presents *core fusion*, a reconfigurable chip multiprocessor (CMP) architecture where groups of fundamentally independent cores can dynamically morph into a large execution engine, or they can be used as distinct processing elements, as needed at run time by applications. Core fusion gracefully accommodates workload diversity and incremental parallelization in CMPs. It requires no additional programming effort or specialized compiler support, maintans ISA compatibility, and keeps both hardware and software complexity manageable.

Our evaluation pits core fusion against more traditional homogeneous and asymmetric CMP architectures and shows that, when confronted with workload diversity, core fusion is the *only* architecture evaluated in the paper that is consistently at or near the top performance level, whereas every other architecture lags significantly behind in at least one scenario.

1 INTRODUCTION

Chip multiprocessors (CMPs) hold the prospect of delivering long-term performance scalability while dramatically reducing design complexity compared to monolithic wide-issue processors. Complexity is reduced by designing and verifying a single, relatively simple core, and then replicating it [8]. Performance is scaled by integrating larger numbers of cores on the die and harnessing increasing levels of TLP with each new technology generation.

Unfortunately, high-performance parallel programming constitutes a tedious, time-consuming, and error-prone effort. In that respect, the complexity shift from hardware to software that ordinary CMPs represent is one of the most serious hurdles to their success. In the short term, on-chip integration of a modest number of relatively powerful (and relatively complex) cores may yield high utilization when running multiple sequential workloads, temporarily avoiding the complexity of parallelization. However, although sequential codes are likely to remain important, they alone are not sufficient to sustain long-term performance scalability. Consequently, harnessing the full potential of CMPs in the long term makes the adoption of parallel programming inevitable.

To amortize the cost of parallelization, many programmers choose to parallelize their applications incrementally. Typically, the most promising loops/regions in a sequential execution of the program are identified through profiling. A subset of these regions are then parallelized, and the rest of the application is left as "future work." Over time, more effort is spent on portions of the remaining code. We call these *evolving* workloads. As a result of this "pay-as-you-go" approach, the complexity (and cost) associated with software parallelization is amortized over a greater time span. In fact, some of the most common shared-memory programming models in use today (e.g., OpenMP [14]) are designed to facilitate the incremental parallelization of sequential codes. We envision a diverse landscape of software in different stages of paralellization, from purely sequential, to fully parallel, to everything in between. As a result, it will remain important to efficiently support sequential as well as parallel code, whether standalone or as regions within the same application at run time. This requires a level of flexibility that is hard to attain in ordinary CMPs.

Asymmetric chip multiprocessors (ACMPs) [5, 26, 27] attempt to address this by providing cores with varying degrees of sophistication and computational capabilities. The number and the complexity of cores are fixed at design time. The hope is to match the demands of a variety of sequential and parallel workloads by executing them on an appropriate subset of these cores. Balakrishnan et al. [5] study the impact of performance asymmetry on explicitly parallelized applications, finding that asymmetry hurts parallel application scalability and renders the applications' performance less predictable unless relatively sophisticated software changes are introduced. Hence, while ACMPs may deliver increased performance on sequential codes, they may do so at the expense of parallel performance, requiring a high level of software sophistication to maximize their potential. We address ACMPs again in our evaluation (Section 6).

Instead of trying to find the right design *trade-off* between complex and simple cores (as ACMPs do), we would like a CMP to provide the *flexibility* to dynamically synthesize the right mix of simple and complex cores based on application requirements. We propose to accomplish this through *core fusion*, an architectural technique that empowers shared-memory CMP cores with the ability to collaboratively exploit high levels of ILP as needed, while still retaining the capacity to deliver high performance on parallel codes. We use a homogeneous CMP as our substrate, with fundamentally independent cores and conventional memory coherence/consistency support. To materialize core fusion, we build upon and extend concepts and mechanisms originally developed for clustered processors (Section 7). We adopt a highly modular approach in both front- and back-end, and provide the ability to fuse and split cores seamlessly at run time.

Core fusion does not require changes to the ISA, it leverages mature micro-architecture technology, and it can interface with the application through simple instrumentation of ordinary parallelization libraries, macros, or directives, without requiring additional programming effort or specialized compiler support. This alone sets our proposal apart from other reconfigurable architectures, such as TRIPS [38] or Smart Memories [31], and from

speculative architectures such as Multiscalar [39]. Section 7 conducts a review of this and other related work.

Core fusion keeps hardware complexity manageable by maintaining the modularity of fine-grain CMPs, and by avoiding large, monolithic hardware structures. Software complexity is also kept manageable because *(a)* the chip's support for sequential codes and run-time reconfiguration facilitates a smoother, more progressive path to parallelization, and *(b)* programmers can still optimize and reason about parallel code using familiar abstractions, without any additional hurdles (e.g., programming for asymmetry in ACMPs). Hence, core fusion strikes an attractive balance between hardware and software design complexity, and provides a level of flexibility that is hard to come by today in the research literature, much less in the market.

Our evaluation pits core fusion against more traditional CMP architectures, such as fine- and coarse-grain homogenous cores, as well as asymmetric CMPs. When confronted with a variety of evolving, parallel, multiprogrammed, and sequential workloads, our results show that core fusion's flexibility and run-time reconfigurability makes it the *only* CMP architecture evaluated in the paper that is the top performer or more closely tracks the top performer in all cases, whereas all other architectures lag significantly behind in at least one scenario.

Contributions

To our knowledge, this is the first paper that adresses workload diversity and software evolution in CMPs via reconfigurability while maintaining ISA compatibility and requiring no additional programming effort or specialized compiler support. In the course of formulating core fusion, this paper makes the following additional contributions:

- A reconfigurable, distributed front-end and instruction cache organization that can leverage individual cores' front-end structures to support an aggressive fused back-end, without overprovisioning individual front-ends.
- A complexity-effective remote wake-up mechanism that allows operand communication across cores without requiring additional register file ports, wake-up buses, bypass paths, or issue queue ports.
- A reconfigurable, distributed load/store queue and data cache organization that *(a)* fully leverages individual cores' data caches and load/store queues in both fused and split modes of operation, *(b)* supports coherence when running parallel code, generates zero coherence traffic when running sequential code in fused mode, and requires minimal changes to each core's CMP sub-system, *(c)* guarantess correctness without requiring data cache flushes upon run-time configuration changes, and *(d)* supports memory consistency in both modes.
- A reconfigurable, distributed ROB organization that can fully leverage individual cores' ROBs to support an agressive fused core, without overprovisioning or unnecessarily replicating individual ROBs.
- Hardware mechanisms and a simple application interface to enable run-time reconfiguration and support for evolving workloads, that requires no specialized compiler support.

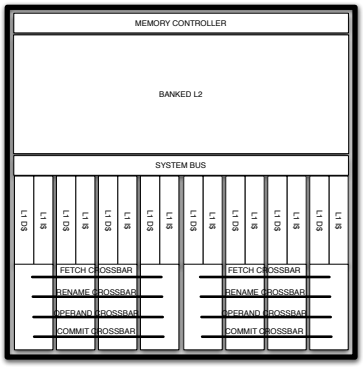

Figure 1: Example CMP organization and the crossbar additions needed to support core fusion.

- A quantitative assessment of the incremental parallelization process on contemporary CMP architectures, and demonstration of the shortcomings of statically defined homogeneous and asymmetric CMPs when confronted with workload diversity in general.

The rest of this paper is organized as follows. Section 2 describes our fine-grain CMP substrate, Section 3 details hardware mechanisms that empower this substrate with fusion capability to run sequential codes fast, Section 4 introduces the support for evolving workloads through runtime reconfiguration, Section 5 describes our methodology, Section 6 presents our results, Section 7 surveys related work, and Section 8 lists our conclusions.

2 FINE-GRAIN CMP SUBSTRATE

In order to deliver high performance on scalable parallel applications, we depart from a CMP substrate with a homogeneous set of small cores. In doing so, we maximize the core count to exploit high levels of TLP, and enjoy the simplicity and modularity advantages of fine-grain CMPs. Figure 1 shows a simplified diagram of our baseline CMP. The chip consists of eight two-issue, out-of-order cores. A system bus connects the L1 data caches and enables snoop-based cache coherence. Beyond the system bus, the chip contains a shared L2 cache and an integrated memory controller. The figure is not meant to represent an actual floorplan, but rather a conceptual one.

3 CORE FUSION HARDWARE

Our proposal delivers high performance on sequential codes by empowering basic CMP cores with the ability to collaboratively exploit high levels of ILP. This is made possible primarily by employing fetch, rename, operand, and commit cross-core wiring (Figure 1), which we call *crossbars* for simplicity. Fetch and rename crossbars coordinate the operation of the distributed front-end, while operand and commit crossbars are responsible for data communication and distributed commit, respectively. Because these wires link nearby cores together, they require three clock cycles to transmit data between any two cores.

The architecture can fuse groups of two or four cores, making it possible to provide the equivalent of eight two-issue, four four-issue, or two eight-issue processor configurations. Asymmetric fusion is also possible, e.g., one eight-issue fused CPU and four more two-issue cores. This flexibility allows core fusion to accommodate workloads of a widely diverse nature, including workloads with multiple parallel or sequential applications.

In this section, we describe in detail the proposed additions to a baseline CMP to enable core fusion and present the structural changes at the front- and back-end of each core's pipeline. Whenever possible, we try to leverage the basic core's fundamentally independent nature.Without loss of generality, the following discussion describes the fusion mechanism involving four basic cores. We assume a RISC ISA where every instruction can be encoded in one word. For CISC-like ISAs, predecoding/translation support is assumed.

3.1 Front-End

3.1.1 Collective Fetch

Due to their fundamentally independent nature, each core is naturally equipped with its own PC, instruction cache, branch predictor and branch target buffer (BTB), as well as return address stack (RAS). A small control unit called the *fetch management unit* (FMU) is attached to the fetch crossbar. The crossbar latency is three cycles. When cores are fused, the FMU coordinates the distributed operation of all core fetch units.

Fetch Mechanism and Instruction Cache

Cores collectively fetch an eight-instruction block in one cycle by each fetching a two-instruction portion (their default fetch capacity) from their own instruction cache. Fetch is generally eight-instruction aligned, with core zero being responsible for the oldest two instructions in the fetch group, core one for the next two, and so forth. When a branch target is not fully aligned in this way, fetch still starts aligned at the appropriate core (lower-order cores skip fetch in that cycle), and it is truncated accordingly so that fully aligned fetch can resume on the next cycle.

Cache blocks, as delivered by the L2 cache on an i-cache miss, are eight words regardless of the configuration. On an i-cache miss, a full block is requested. This block is delivered to the requesting core if it is operating independently, or distributed across all four cores in a fused configuration to permit collective fetch. To achieve this, we make i-caches reconfigurable, along the lines of earlier works [31]. Each i-cache has enough tags to organize its data in two- or eight-word blocks, and each tag has enough bits to handle the worst of the two cases. When running independently, three out of every four tags are unused, and the i-cache handles block transfers in eight-word blocks. When in fused configuration, the i-cache uses all tags, covering two-word blocks. (How to dynamically switch from one i-cache mode to the other is explained later in Section 4.)

Because fetch is collective, it makes sense to just replicate the i-TLB across all cores in a fused configuration. Notice that this would be accomplished "naturally" as cores miss on their i-TLBs, however taking multiple i-TLB misses for a single eight-instruction block is unnecessary, since the FMU can be used to refill all i-TLBs upon a first i-TLB miss by a core. Finally, the FMU can also be used to gang-invalidate an i-TLB entry, or gang-flush all i-TLBs as needed.

Branches and Subroutine Calls

Prediction. During collective fetch, each core accesses its own branch predictor and BTB. Because collective fetch is always aligned, dynamic instances of the same static branch instruction are guaranteed to access the same branch predictor and BTB. Consequently, the effective branch predictor and BTB capacity is four times as large. This is a desirable feature, since the penalty of branch misprediction is bound to be higher with the more aggressive fetch/issue width and the higher number of in-flight instructions in the fused configuration.

Each core can handle up to one branch prediction per cycle. The redirection of the (distributed) PC upon taken branches and branch mispredictions is enabled by the FMU. Each cycle, every core that predicts a taken branch, as well as every core that detects a branch misprediction, sends the new target PC to the FMU. The FMU selects the correct PC by giving priority to the oldest misprediction-redirect PC first, and the youngest branch-prediction PC last, and sends the selected PC to all fetch units. Once the transfer of the new PC is complete, cores use it to fetch from their own i-cache as explained above.

Naturally, on a misprediction, misspeculated instructions are squashed in all cores. This is also the case for instructions fetched along the not-taken path on a taken branch, since the target PC will inevitably arrive with a delay of a few cycles.

Global History. The FMU can also provide the ability to keep global history across all four cores if needed for accurate branch prediction.[1] To accomplish this, the GHR can be simply replicated across all cores, and updates be coordinated through the FMU. Specifically, upon every branch prediction, each core communicates its prediction—whether taken or not taken—to the FMU. Two bits suffice to accomplish this. Additionally, as discussed, the FMU receives nonspeculative updates from every back-end upon branch mispredictions. The FMU communicates such events to each core, which in turn update their GHR. Upon nonspeculative updates, earlier (checkpointed) GHR contents are recovered on each core. The fix-up mechanism employed to checkpoint and recover GHR contents can be along the lines of the outstanding branch queue (OBQ) mechanism in the Alpha 21264 microprocessor [24].

Return Address Stack. As the target PC of a subroutine call is sent to all cores by the FMU (which flags the fact that it is a subroutine call), core zero pushes the return address into its RAS. When a return instruction is encountered (possibly by a different core from the one that fetched the subroutine call) and communicated to the FMU, core zero pops its RAS and communicates the return PC address back through the FMU. Notice that, since all RAS operations are processed by core zero, the effective RAS size does not increase when cores are fused. This is reasonable, however, as call depth is a program property that is independent of whether execution is taking place on an independent core or on a fused configuration.

Handling Fetch Stalls

When one fetch engine stalls as a result of an i-cache or i-TLB miss, or there is contention on a fetch engine for branch predictor ports (two consecutive branches fetched by the same core in the same cycle), it is necessary for all fetch engines to stall, so that correct ordering of the instructions in one fetch block can be maintained (e.g., for orderly FMU resolution of branch targets), and to allow instructions in the same fetch group to flow through later stages of the fused front-end (most notably, through

[1] Because each core is responsible for a subset of the branches in the program, having independent and uncoordinated history registers on each core could place correlated branches on different cores and make it impossible for the branch predictor to learn of their correlation.

rename) in a lock-step fashion. To support this, cores communicate fetch stalls to the FMU, which informs the other cores. Because of the three-cycle crossbar latency, it is possible that the other cores may over-fetch in the shadow of the stall handling by the FMU if *(a)* on an i-cache or i-TLB miss, one of the other cores does hit in its i-cache or i-TLB (very unlikely in practice, given how fused cores fetch), or *(b)* generally in the case of contention for branch prediction ports by two back-to-back branches fetched by the same core (itself exceedingly unlikely). In any case, once all cores have been informed, including the delinquent core, they discard any over-fetched instruction (similarly to the handling of a taken branch) and resume fetching in sync from the right PC—as if all fetch engines had synchronized through a "fetch barrier."

3.1.2 Collective Decode/Rename

After fetch, each core pre-decodes its instructions independently. Subsequently, all instructions in the fetch group need to be renamed. As in clustered architectures, steering consumers to the same core as their producers can improve performance by eliminating communication delays.

Renaming and steering of instructions is achieved through a small control unit called the *steering management unit* (SMU). The SMU consists of a global *steering table* to track the mapping of architectural registers to any core, four free-lists for register allocation (one for each core), four rename maps and steering/renaming logic (Figure 2). The steering table and the four rename maps together allow up to four valid mappings of each architectural register, and enable operands to be replicated across multiple cores. Cores still retain their individual renaming structures, but these are bypassed when cores are fused.

Figure 3 depicts the high level organization of the rename pipeline. After pre-decode, each core sends up to two instructions to the SMU through a set of links. We assume it is possible to support three-cycle cross-core communication over a repeated link. Three cycles after pre-decode, the SMU receives up to two instructions and six architectural register specifiers (three per instruction) from each core. After renaming and steering, it uses a second set of links to dispatch up to six physical register specifiers, two instructions and two copy operations to each core. Restricting the SMU dispatch bandwidth in this way keeps the wiring overhead manageable, lowers the number of required rename map ports, and also helps achieve load balancing among the fused cores. Collectively, the incoming and outgoing SMU links constitute the rename crossbar.

The SMU uses the incoming architectural register specifiers and the four free lists to rename up to eight instructions every pipeline cycle. Each instruction is dispatched to one of the cores via dependence based steering [33]. For each instruction, the SMU consults the steering table to steer every instruction to the core that will produce most of its operands among all cores with free rename crossbar ports. Copy instructions are also inserted into the fetch group in this cycle. In the next cycle, instructions (and the generated copies) are renamed by accessing the appropriate rename map and free list. Since each core receives no more than two instructions and two copy instructions, each rename map has only six read and eight write ports. The steering table requires eight read and sixteen write ports; note that each steering table entry contains only a single bit, and thus the overhead of multi-porting this small table is relatively low. After rename, regular and copy instructions are dispatched to the appropriate cores. If a copy instruction cannot be sent due to bandwidth restrictions, renaming stops at the offending instruction that cycle, and starts with the same instruction next cycle, thereby draining crossbar links and guaranteeing forward progress. Similarly, if resource occupancies prevent the SMU from dispatching an instruction, renaming stops that cycle, and resumes when resources are available. To facilitate this, cores inform the FMU through a four-bit interface when their issue queues, ROBs, and load/store queues are full.

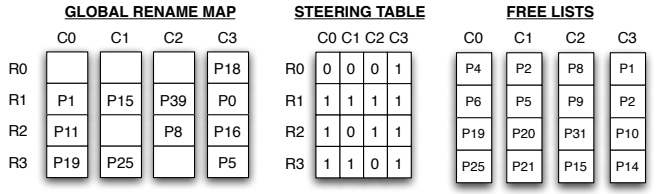

Figure 2: Example SMU organization (only four architectural registers shown). R0 has a valid mapping in core three, whereas R1 has four valid mappings (one in each core).

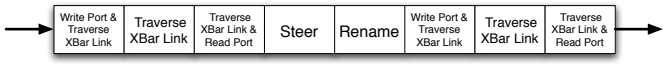

Figure 3: Organization of the Rename Pipeline.

Registers are recycled through two mechanisms. As in many existing microprocessors, at commit time, any instruction that renames an architectural register releases the physical register holding the result of the previous instruction that renamed the same register. This is accomplished in core fusion over a portion of the rename crossbar, by having each ROB send the specifiers for these registers to the SMU. However, copy instructions do not allocate ROB entries, and recycling them requires an alternative strategy. Every copy instruction generates a replica of a physical register in one core on some other core. These replicas are not recovered on branch mispredictions. Therefore, a register holding a redundant replica can be recycled at any point in time as long as all instructions whose architectural source registers are mapped to that physical register have read its value. To facilitate recycling, the SMU keeps a one-bit flag for each register indicating whether the corresponding register is currently holding a redundant replica (i.e., it was the target of a copy instruction). In addition, the SMU keeps a table of per-register read counters for each core, where every counter entry corresponds to the number of outstanding reads to a specific physical register (each counter is four bits in a sixteen-entry issue queue, which is our case (Section 5)). These counters are incremented at the time the SMU dispatches instructions to cores. Every time an instruction leaves a core's issue queue, the core communicates the specifiers for the physical registers read by the instruction, and the SMU decrements the corresponding counters. When a branch misprediction or a replay trap is encountered, as squashed instructions are removed from the instruction window, the counters for the corresponding physical registers are updated appropriately in the shadow of the refetch.

3.2 Back-End

Each core's back-end includes separate floating-point and integer issue queues, a copy-out queue, a copy-in queue, a

physical register file, functional units, load/store queues and a ROB. Each core's load/store queue has access only to its private L1 data cache. The L1 caches are connected via a split-transaction bus and are kept coherent via a MESI protocol. This split-transaction bus is also connected to an on-chip L2 cache that is shared by all cores. When running independently on one core, the operation of the back-end is no different from the operation of a core in a homogeneous CMP. When cores get fused, back-end structures are coordinated to form a large virtual back-end capable of consuming instructions at a rate of eight instructions per cycle.

3.2.1 Collective Execution

Operand Crossbar

Copy instructions wait in the copy-out queues for their operands to become available, and once issued, they transfer their source operand and destination physical register specifier to a remote core over the operand crossbar (Figure 1). The operand crossbar is capable of supporting every cycle two copy instructions per core. In addition to copy instructions, loads use the operand crossbar to deliver values to their destination register.

Wake-up and Selection

When copy instructions reach the consumer core, they are placed in a copy-in queue to wait for the selection logic to schedule them. Each cycle, the issue queue scheduler considers the two copy instructions at the queue head for scheduling along with the instructions in the issue queue. Once issued, copies wake up their dependent instructions and update the physical register file, just as regular instructions would do.

Reorder Buffer and Commit Support

The goal of the fused in-order retirement operation is to coordinate the operation of four ROBs to commit up to eight instructions per cycle. Instructions allocate ROB entries locally at the end of fetch. If the fetch group contains less than eight instructions, NOPs are allocated at the appropriate cores to guarantee alignment. Of course, on a pipeline bubble, no ROB entries are allocated.

When commit is not blocked, each core commits two instructions from the oldest fetch group every cycle. When one of the ROBs is blocked, all other cores must also stop committing on time to ensure that fetch blocks are committed atomically in-order. This is accomplished via the *commit crossbar*, which transfers stall/resume signals across all ROBs. To accommodate the communication delay across the crossbars, each ROB is extended with a *speculative head pointer* in addition to the conventional head and tail pointers. Instructions always pass through the speculative ROB head before they reach the actual ROB head and commit. If they are not ready to commit at that time, they send a stall signal to all cores. Later, when they become ready, they move past the speculative head and send a resume signal to the other cores. The number of ROB entries between the speculative head pointer and the actual head pointer is enough to cover the crossbar delay. This guarantees that ROB stalls always take effect in a timely manner to prevent committing speculative state. In our experiments, we set the crossbar communication latency to three cycles, and consequently the actual head is separated from the speculative head by six instruction slots on each core at all times.

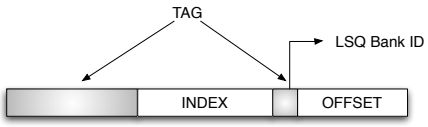

Figure 4: LSQ bank assignment and core-fusion-oblivious cache line indexing/tagging.

3.2.2 Load/Store Queue Organization

Our scheme for handling load and store instructions is conceptually similar to previously proposed clustered architectures [12, 41, 6, 29, 22], but a number of important differences exist. While most proposals in clustered architectures have chosen a centralized L1 data cache or have distributed it based on bank assignment, in core fusion, we keep the private nature of each L1 cache, and consequently we require only minimal modifications on the CMP cache subsystem.

In the fused mode, we adopt a banked-by-address LSQ implementation. This allows us to keep data coherent without requiring cache flushes after dynamic reconfiguration, and to support elegantly store forwarding and speculative loads. The core that issues each load/store to the memory system is determined based on effective addresses. The two bits that follow the block offset are used as the LSQ bank-ID to select one of the four cores (Figure 4), and enough index bits to cover the L1 cache are allocated from the remaining bits. The rest of the effective address and the bank-ID are stored as a tag (note that this does not increase the number of tag bits compared to a conventional indexing scheme). Making the bank-ID bits part of the tag is important to properly disambiguate cache lines regardless of the configuration.

Effective addresses for loads and stores are generally not known at the time they are renamed. This raises a problem since at rename time memory operations need to allocate LSQ entries from the core that will eventually issue them to the memory system. We attack this problem through bank prediction [6, 9]. Upon pre-decoding loads and stores, each core accesses its bank predictor by using the lower bits of the load/store PC. Bank predictions are sent to the SMU through the rename crossbar, and the SMU steers each load and store to the predicted core. Each core allocates load queue entries for the loads it receives. On stores, the SMU also signals all cores to allocate dummy store queue entries regardless of the bank prediction. Dummy store queue entries guarantee in-order commit for store instructions by reserving placeholders across all banks for store bank mispredictions. Upon effective address calculation, remote cores with superfluous store queue dummies are signaled to discard their entries (Recycling these entries requires a collapsing LSQ implementation.) If a bank misprediction is detected, the store is sent to the correct queue.

In the case of loads, if a bank misprediction is detected, the load queue entry is recycled (LSQ collapse) and the load is sent to the correct core. There, it allocates a load queue entry and resolves its memory dependences locally. Notice that, as a consequence of bank mispredictions, loads can allocate entries in the load queues out of program order. Fortunately, this is not a problem for store-to-load forwarding because load queue entries are typically tagged by instruction age to facilitate forwarding. However, there is a danger of deadlock in cases where the mispredicted load is older than all other loads in its (correct) bank and the load queue is full at the time the load arrives at the consumer core. To prevent this situation, loads search the load queue for older instructions

when they cannot allocate entries. If no such entry is found, a replay trap is taken, and the load is steered to the right core at rename time. Otherwise, the load is buffered until a free load queue entry becomes available. Address banking of the LSQ also facilitates speculative loads and store forwarding. Since any load instruction is free of bank mispredictions at the time it issues to the memory system, loads and stores to the same address are guaranteed to be processed by the same core. Dependence speculation can be achieved by integrating a store-set predictor [16] on each core (since cores perform aligned fetches, the same load is guaranteed to access the same predictor at all times.)

When running parallel application threads in fused mode, the memory consistency model must be enforced on all loads and stores. We assume relaxed consistency models where special primitives like memory fences (weak consistency) or acquire/release operations (release consistency) enforce ordering constraints on ordinary memory operations. Without loss of generality, we discuss the operation of memory fences below. Acquire and release operations are handled similarly.

For the correct functioning of synchronization primitives, fences must be made visible to all load/store queues belonging to the same thread. We achieve this by dispatching these operations to all the queues, but having only the copy in the correct queue perform the actual synchronization operation. The fence is considered complete once each one of the local fences completes locally and all memory operations preceding each fence commit. Local fence completion is signaled to all cores through a one-bit interface in the portion of the operand crossbar that connects the load-store queues across the cores.

4 DYNAMIC RECONFIGURATION

Our discussion thus far explains the operation of the cores in a static fashion. This alone improves performance on highly parallel or purely sequential applications, by configuring the CMP prior to executing the application. However, partially parallelized applications will benefit most from the ability to fuse/split cores at run time, as they dynamically switch between sequential and parallel code regions, respectively. Supporting dynamic reconfiguration of the architecture requires several actions to be taken to ensure correctness upon core fusion and fission, and an application interface to coordinate and trigger reconfiguration.

Fortunately, the modular nature of our architecture makes reconfiguration relatively easy. In general, we envision run-time reconfiguration enabled through a simple application interface. The application requests core fusion/fission actions through system calls. In most cases, this can be readily encapsulated in conventional parallelizing macros or directives.

Fusion. After the completion of a parallel region, the application may request cores to be fused to execute the upcoming sequential region. Cores need not get fused on every parallel-to-sequential region boundary: if the sequential region is not long enough to amortize the cost of fusion, execution can continue without reconfiguration on one of the small cores. All in-flight instructions following the system call are flushed, and the appropriate rename map on the SMU is updated with mappings to the architectural state on this core. Data caches do not need any special actions to be taken upon reconfigurations; the coherence protocol naturally ensures correctness across configuration changes. I-caches undergo tag reconfiguration upon fusion and fission as described before (Section 3.1), and all cache blocks are invalidated for consistency. This is generally harmless if it can be amortized over the duration of a configuration. In any case, the programmer or the run-time system may choose not to reconfigure across fast-alternating program regions (e.g., short serial sections in between parallel sections). In no case is the shared L2 affected by reconfiguration.

Fission. Fission is achieved through a second system call, where the application informs the processor about the approaching parallel region. Fetch is stalled, in-flight instructions are allowed to drain, and enough copy instructions are generated to gather the architectural state into core zero's physical register file. When the transfer is complete, control is returned to the application.

5 EXPERIMENTAL SETUP

5.1 Architecture

We evaluate the performance potential of core fusion by comparing it against five popular static CMP architectures. As building blocks for these systems, we use two-, four-, and six-issue out-of-order cores. Table 3 and Table 4 show the microarchitectural configuration of the two-issue cores in our experiments. Four- and six-issue cores have two and three times the amount of resources as each one of the two-issue cores, respectively, except that first level caches, branch predictor, and BTB are four times as large in the six-issue core (the sizes of these structures are typically powers of two.) Across different baselines and core fusion, we always maintain the same parameters for the shared portion of the memory subsystem (system bus and lower levels of the memory hierarchy). All configurations are clocked at the same speed. Our experiments are conducted using a detailed, heavily modified version of the SESC [36] simulator. Contention and latency are modeled at all levels, including three-cycle wire delays for cross-core communication across fetch, rename, operand and commit crossbars, as well as the latency of the eight-stage rename pipe when running in fused mode.

Since we explore an inherently area-constrained design space, choosing the right number of large and small cores requires estimating their relative areas. Palacharla et al. [33] show that the area overheads of key microarchitectural resources scale superlinearly with respect to issue width in monolithic cores. Olukotun et al. [32] indicate that replicating narrow-issue cores supports higher aggregate peak issue width compared to a monolithic processor. Kumar et al. [26, 27] estimate that a single-threaded version of the eight-issue Alpha EV8 core requires more than nine times the area of the four-issue EV6 core when implemented in the same technology. These results suggest that area requirements of monolithic cores grow superlinearly with the width of the pipeline (assuming that resources are scaled proportionally). Therefore, a large monolithic core cannot generally support the same aggregate issue width as a fine-grain CMP with multiple small cores using the same silicon area.

Burns et al. [11] estimates the area requirements of out-of-order processors by inspecting layout from the MIPS R10000 and from custom layout blocks, finding that four- and six-issue cores require roughly 1.9 and 3.5 times the area of a two-issue core, respectively, even when assuming clustered register files, issue queues, and rename maps, which greatly reduce the area penalty of implementing

large SRAM arrays.[2] Furthermore, as mentioned above, our six-issue baseline's first level caches and branch predictor are four times (as opposed to three times) as large as a two issue core. Consequently, we model their area requirements to be two and four times higher than a two-issue core, respectively.[3] We believe these estimates are somewhat optimistic.

We estimate the area overhead of our crossbar additions conservatively, assuming that no logic is laid out under the metal layer for wiring. We use the wiring area estimation methodology described in [28], assuming a 65nm technology and Metal-4 layer wiring with a 280nm wire pitch [4]. Accordingly, we find the area of one fetch crossbar (74 bits/link) to be 0.30mm^2, the area of one rename crossbar (218 bits/link) to be 0.80mm^2, and the area of one operand crossbar (76 bits / link) to be 1.46mm^2. The area of the commit crossbar is negligible as it is two bits wide. This yields a total area overhead of 2.56mm^2 for fusing a group of 4 cores, corresponding to a total crossbar area overhead of 5.12mm^2 for our eight-core CMP. Using CACTI 3.2, we also estimate the total area overhead of the SMU and bank predictors (4 bank predictors, one per core) to be 0.13 and 0.72mm^2, respectively, for a total of 1.7mm^2 for the entire chip. Adding these to the crossbar area estimates, we find the total area overhead of core fusion to be 6.72mm^2. Even for a non-reticle-limited, 200mm^2 die that devotes half of the area to the implementation of the cores, this overhead represents a little over half the area of one core. Nevertheless, we conservatively assume the area overhead to be equal to one core.

Table 1 details the number and type of cores used in our studies for all architectures we model. Our core-fusion-enabled CMP consists of eight two-issue cores. Two groups of four cores can each be fused to synthesize two large cores on demand. For our coarse-grain CMP baselines, we experiment with a CMP consisting of two six-issue cores (CoarseGrain-6i) and another coarse-grain CMP consisting of four four-issue cores (CoarseGrain-4i). We also model an asymmetric CMP with one six-issue and four two-issue cores (Asymmetric-6i), and another asymmetric CMP with one four-issue and six two-issue cores (Asymmetric-4i). Finally, we model a fine-grain CMP with nine two-issue cores (FineGrain-2i). The ninth core is added to compensate for any optimism in the area estimates for six- and four-issue cores, and for the area overhead of core fusion. We have verified that all the parallel applications studied use this ninth core effectively.

CMP Configuration	Composition (Cores)
CoreFusion	8×2-issue
FineGrain-2i	9×2-issue
CoarseGrain-4i	4×4-issue
CoarseGrain-6i	2×6-issue
Asymmetric-4i	1×4-issue + 6×2-issue
Asymmetric-6i	1×6-issue + 4×2-issue

Table 1: Simulated architectures.

[2]Note that, when all resources are scaled linearly, monolithic register files grow as $O(w^3)$, where w is the issue width. This is due to the increase in the number of bit lines and word lines per SRAM cell, times the increase in physical register count.

[3]We also experimented with an eight-issue clustered core (optimistically assumed to be area-equivalent to the six-issue core), but found its performance to be inferior. Consequently, we chose the six-issue monolithic core as our baseline.

5.2 Applications

We evaluate our proposal by conducting simulations on sequential, multiprogrammed, parallel and evolving parallel workloads. Our parallel workloads represent a mix of scalable scientific applications (three applications from the Splash-2 suite [40] and two applications from SPEC OpenMP [3]), and parallelized data mining applications [2, 30, 35]. The input sets we use are listed in Table 2.

Our sequential workloads comprise ten integer and eight floating point applications from the SPEC2000 suite [23]. We use the MinneSpec reduced input sets [25]. In all cases, we skip the initialization parts and then simulate the applications to completion.[4]

5.2.1 Evolving Workload Construction

We derive our evolving workloads from existing applications by following a methodology that aims at mimicking an actual incremental parallelization process. Specifically, we use Swim and Equake from the SPEC OpenMP suite, and MG from the OpenMP version of the NAS benchmarks to synthesize our evolving workloads. These scalable applications contain multiple parallel regions that exploit loop-level parallelism [3]. We emulate the incremental parallelization process by gradually transforming sequential regions into parallel regions, obtaining more mature versions of the code at each turn. To do this, we first run each application in single-threaded mode and profile the runtimes of all regions in the program. We then create an initial version of the application by turning on the parallelization for the most significant region while keeping all other regions sequential. We repeat this process until we reach the fully parallelized version, turning on the parallelization of the next significant region at each step along the process.

5.2.2 Multiprogrammed Workload Construction

Similar to highly parallel applications, workloads with high degrees of multiprogramming typically benefit most from maximizing on-chip core count to exploit TLP aggressively [19]. We do not evaluate these separately as the demands they place on the architecture in terms of core count and per-core performance are virtually the same as highly parallel applications (which we cover in our evaluation), favoring fine-grain CMP configurations with many cores (which core fusion builds upon). Consequently, we turn our attention to desktop workloads which typically exhibit low degrees of multiprogramming.

We derive our multiprogrammed workloads from the SPEC2000 suite [23]. We classify applications as high- and low-ILP benchmarks based on how much speedup they obtain in going from a two-issue core to four- and six-issue cores. We then use these classifications to guide our workload construction process. We set the degree of multiprogramming to two applications, and we form a total of nine workloads with different ILP characteristics: high-ILP workloads, low-ILP workloads, and mixed (both high and low ILP) workloads. When running these workloads, only two CPUs are active at any point in time.

[4]Our simulation infrastructure currently does not support the other SPEC benchmarks.

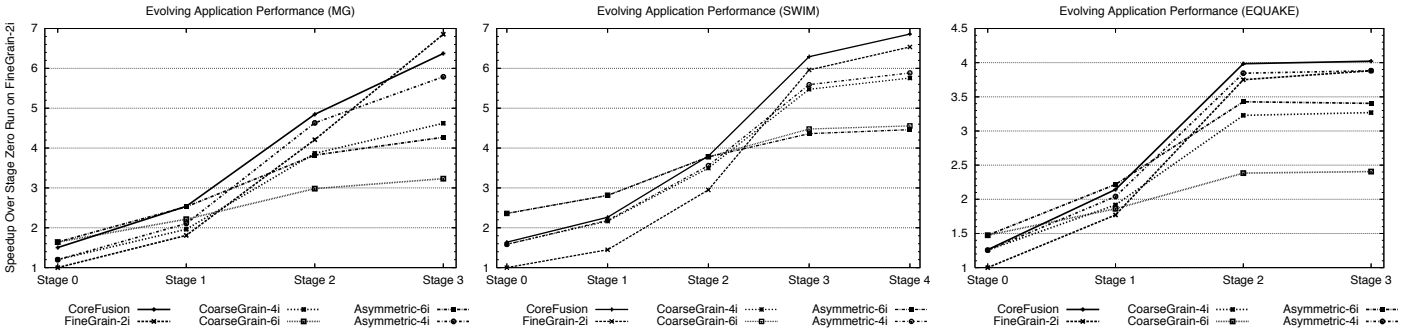

Figure 5: Speedup over stage zero run on FineGrain-2i.

Splash-2	Description	Problem size
BARNES	Evolution of galaxies	16k part.
FMM	N-body problem	16k part.
RAYTRACE	3D ray tracing	car
Spec OpenMP		
SWIM-OMP	Shallow water model	MinneSpec-Large
EQUAKE-OMP	Earthquake model	MinneSpec-Large
NAS OpenMP		
MG	Multigrid Solver	Class A
Data Mining		
BSOM	Self-organizing map	2,048 rec., 100 epochs
BLAST	Protein matching	12.3k sequ.
KMEANS	K-means clustering	18k pts., 18 attr.
SCALPARC	Decision Tree	125k pts., 32 attr.

Table 2: Simulated parallel applications and their input sizes.

Two-Issue Core	
Frequency	4.0 GHz
Fetch/issue/commit	2/2/2
Int/FP issue queues	16/16
ROB entries	48
Int/FP registers	32+40 / 32+40 (Arch.+Rename.)
Integer FUs	1×ALU 1×AGU 1×Br. 1/1×Mul/Div
Floating-point FUs	1×ALU 1/1×Mul/Div
Ld/St queue entries	12/12
Max. unresolved br.	12
Br. penalty	7 cycles (minimum)
Max. br. pred. rate	1 taken/cycle
Br. predictor	Alpha 21264
BTB size	512 entries, direct mapped
RAS entries	32
Bank predictor	2K-entries
iL1/dL1 size	16 kB
iL1/dL1 block size	32B / 64B
iL1/dL1 associativity	2-way
iL1/dL1 round-trip	2 cycles (uncontended)
iL1/dL1 ports	1 / 2
iL1/dL1 MSHR entries	8
dL1 coherence protocol	MESI-based protocol
Consistency model	Release consistency
Cross-core communication	3 cycles

Table 3: Baseline two-issue processor parameters.

Shared-Memory Subsystem	
System bus transfer rate	64GB/s
System bus width	256 bits
Shared L2	4MB, 64B block size
Shared L2 associativity	8-way
Shared L2 banks	16
L2 MSHR entries	32
L2 round-trip	10 cycles (uncontended)
Memory round-trip	320 cycles (uncontended)

Table 4: Shared-memory subsystem parameters.

6 EVALUATION

6.1 Evolving Application Performance

Figure 5 compares the performance of all six architectures on our evolving workloads. Each graph shows the speedups obtained by each architecture as applications evolve from sequential (stage zero) to highly parallel (stage three for MG and Equake-OMP, stage four for Swim-OMP). When running on the asymmetric CMPs, we schedule the master thread on the large core so that sequential regions are sped up. Parallel regions are executed on all cores (we also experimented with running parallel regions on small cores only, but found that the results were inferior). We evaluate our proposal by applying dynamic core fusion to fuse/split cores when running sequential/parallel regions, respectively.

When applications are not parallelized (stage zero), exploiting ILP is crucial to obtaining high performance. As a result, coarse-grain CMPs, asymmetric CMPs and CoreFusion all enjoy speedups over the fine-grain CMP. In this regime, performance is strictly a function of the largest core on the chip. For instance, in MG, configurations with four-issue cores (CoarseGrain-4i and Asymmetric-4i) improve performance by a factor of 1.2, while configurations with six-issue cores (CoarseGrain-6i and Asymmetric-6i) obtain a speedup of 1.64. CoreFusion achieves a speedup of 1.5, outperforming all but the six-issue configurations due to its ability to exploit high levels of ILP. Similar trends are observed on Swim-OMP and Equake-OMP.

When an initial parallelization of the application is performed (stage one), all architectures benefit from exploiting thread-level parallelism (TLP). However, significant portions of the applications are still sequential, and exploiting ILP is still crucial for getting optimum performance. On MG, CoreFusion and Asymmetric-6i both obtain a speedup of 2.5. Asymmetric-6i's monolithic core marginally outperforms CoreFusion's fused core, but as a result of dynamic fusion and fission, CoreFusion enjoys a higher core count on parallel regions, thereby exploiting higher levels of TLP. In this regime, Asymmetric-4i obtains a speedup of only 2.1: this configuration has two more cores than Asymmetric-6i, but the application does not yet support enough TLP to cover the performance hit with respect to Asymmetric-6i's six-issue core on sequential regions. CoarseGrain-4i obtains a speedup of 2.2. Because of the scarcity of TLP in this evolutionary stage, FineGrain-2i performs worst among all architectures, with a speedup of only 1.8. Similar trends emerge in Equake. On Swim, Asymmetric-6i and CoarseGrain-6i obtain speedups of 2.8, followed by CoreFusion's speedup of 2.3.

Over time, greater portions of the applications are parallelized (stage two for MG and Equake-OMP, stages two and three for Swim-OMP). In this regime, exploiting ILP and TLP are equally important for obtaining high speedup. In all three applications, CoreFusion outperforms all other architectures. CoreFusion obtains

speedups of 4.8, 4, and 6.3 on MG, Equake-OMP, and Swim-OMP. This is followed by speedups of 4.6 and 3.8 from Asymmetric-4i on MG and Equake-OMP, and a speedup of 6 from FineGrain-2i on Swim-OMP. Due to the increased level of TLP, the fine-grain CMP starts to outperform architectures that invest heavily in ILP exploitation (Asymmetric-6i and CoarseGrain-6i).

Eventually, enough effort is expended in parallelization to convert each benchmark into a scalable parallel application (stage four). In MG, performance is determined strictly by core count. FineGrain-2i obtains the best speedup (6.9), followed immediately by CoreFusion (6.4). Asymmetric-4i obtains a speedup of 5.8, while architectures that invest in ILP (Asymmetric-6i and CoarseGrain-6i) take a significant performance hit (speedups of 4.3 and 3.2, respectively). In Swim-OMP and Equake-OMP, CoreFusion still performs the best, followed closely by the fine-grain CMP. This is because all parallel applications, regardless of their parallelization stage, have sequential regions (on which CoreFusion outperforms the FineGrain-2i through dynamic fusion). Note, however, that statically allocating a large core to obtain speedup on these regions does not pay off, as evidenced by the lower performance of the asymmetric CMPs compared to CoreFusion: attempting to exploit ILP in these regions is worthwhile only if it does not adversely affect the exploitation of TLP.

In summary, performance differences between the best and the worst architectures at any parallelization stage are high, and moreover, the best architecture at one end of the evolutionary spectrum performs worst at the other end. As applications evolve through the incremental parallelization process, performance improves on all applications. Throughout this evolution, CoreFusion is the only architecture that consistently performs the best or rides close to the best configuration. While all static architectures get "stuck" at some (different) point along the incremental parallelization process, core fusion adapts to the changing demands of the evolving application and obtains significantly higher overall performance.

6.2 Parallel Application Performance

Figure 6 compares the performance of core fusion against our baseline CMP configurations on parallel workloads. Results are normalized to the performance of single-threaded runs on FineGrain-2i. As expected, on scalable parallel applications, maximizing the number of cores leads to significant performance improvements. The fine-grain CMP performs best on this class of applications due to its higher number of cores that allows it to aggressively harness TLP. FineGrain-2i is followed immediately by CoreFusion, which has one fewer core due to its area overheads (our results confirm that the ninth core is effectively used in FineGrain-2i by all the parallel applications).

Asymmetric-4i devotes the area equivalent of two two-issue cores to the implementation of the four-issue superscalar. As a result, it obtains an average speedup of 5.5 while CoreFusion and FineGrain-2i improve performance by factors of 6.6 and 7. Similarly, Asymmetric-6i devotes the area equivalent of four cores to the implementation of the wide-issue superscalar. As a result, this configuration experiences up to 2 times performance degradation with respect to the fine-grain CMP on scalable parallel applications. CoarseGrain-6i performs even worse than Asymmetric-6i, since it only supports two cores, and the additional performance improvements obtained by exploiting ILP are not enough to offset the performance loss caused by foregoing TLP. CoarseGrain-4i obtains an average speedup of 4.6, and outperforms Asymmetric-6i on five applications. This is due to the difficulties of obtaining load balancing on asymmetric architectures [5]: while CoarseGrain-4i distributes work to four equivalent cores, Asymmetric-6i distributes an equal amount of work to cores with significant differences in their computational abilities. Ultimately, this causes Asymmetric-6i to run at the speed of five two-issue cores, which is often outperformed by the four four-issue cores of the CoarseGrain-4i.

In summary, CoreFusion approaches the performance of the fine-grain CMP on scalable parallel applications, whereas coarse-grain and asymmetric designs sacrifice parallel application performance significantly to improve single-thread performance. These architectures are forced to *trade off* TLP for ILP by their static nature, while CoreFusion aims to *synthesize* the right ILP/TLP balance based on workload needs.

6.3 Multiprogramming Performance

Figure 7 shows the performance obtained by all six architectures on multiprogrammed desktop-style workloads (constructed as explained in Section 5.2.2). All results are normalized to the performance of the fine-grain CMP. In this figure, the first three workloads are combinations of two high-ILP benchmarks (as explained in Section 5.2.2), while the next three are combinations of high- and low-ILP benchmarks. The last three workloads are low-ILP workloads. For each workload, we report the geometric mean of the speedups of each application in the workload compared to a fine-grain CMP run of the workload. Taking the geometric mean of the speedups equally penalizes/rewards relative performance degradations/improvements in either benchmark in each workload. When running on asymmetric CMPs, we choose the best static scheduling scheme using oracle knowledge (e.g.; we choose the assignments that maximizes the speedup on the asymmetric CMP). As a sanity check, we also experimented with dynamic core assignment strategies [27] for the asymmetric CMPs, but we found the performance of the best (oracle) static assignment to be better. When running on CoreFusion, we experimented with two fused cores and did not merge/split cores at runtime. As explained in Section 5.2.2, we do not consider workloads with high degrees of multiprogramming as their demands for high core count (at the expense of per-core performance) are aligned with the requirements of scalable parallel applications, and their evaluation reveals no additional insights in the context of our paper.

The results indicate that all architectures obtain speedups over the fine-grain CMP on multiprogrammed workloads. This is because FineGrain-2i heavily partitions its area to maximize the core count. While this is an excellent strategy for scalable parallel applications, the lack of high-performance cores hurts the fine-grain CMP when running multiple sequential applications. Due to its two six-issue cores, CoarseGrain-6i performs the best on all applications, obtaining an average speedup of 1.59. CoreFusion's two fused cores come next with a speedup of 1.36 on average. Asymmetric-6i and CoarseGrain-4i obtain speedups of 1.3 each: while Asymmetric-6i improves the performance of one application in the workload significantly, CoarseGrain-4i improves both applications, but with smaller margins compared to Asymmetric-6i's six-issue core. The winner among these two configurations is application dependent. Finally, Asymmetric-4i performs

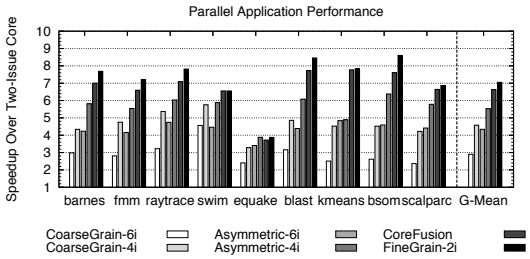

Figure 6: Speedup over single-thread run on FineGrain-2i.

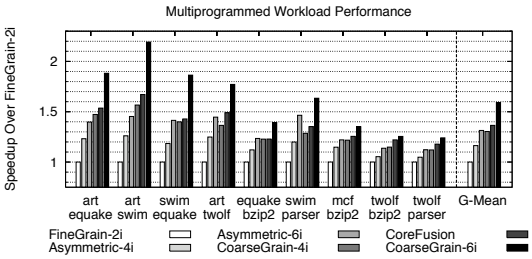

Figure 7: Speedup over FineGrain-2i.

worst after the fine-grain CMP, obtaining a speedup of only 1.16. Not only does Asymmetric-4i have only a single wide-issue core, but also, the performance delivered by this four-issue core is lower than Asymmetric-6i's six-issue core and either one of CoreFusion's fused cores. On high-ILP workloads, the differences are even more pronounced. On art-equake, for example, CoreFusion achieves a speedup of 1.54 while the ACMP-4i obtains a speedup of only 1.23.

In summary, when the degree of multiprogramming is low (typical case for desktop workloads), implementing high-performance cores is the best choice. For this class of workloads, the fine-grain CMP performs worst. CoreFusion is second best after CoarseGrain-6i, while still maintaining the performance advantages of the fine-grain CMP on parallel applications.

6.4 Sequential Application Performance

Figure 8 compares the performance of core fusion against our baseline CMP architectures across SPEC 2000 applications. The plot on the top shows results on integer applications while the bottom plot corresponds to floating-point benchmarks. We report speedups with respect to the fine-grain CMP.

As expected, the results indicate that wide-issue cores have significant performance advantages on sequential applications. Configurations with a six-issue monolithic core obtain average speedups of 89% and 37% on floating-point and integer benchmarks. (Speedups on floating-point benchmarks are typically higher due to higher levels of ILP present in these applications.) Configurations that employ a four-issue core observe average speedups of 40% and 22% on floating-point and integer-benchmarks, respectively. Core fusion improves performance over the fine-grain CMP by 14-72% on floating-point applications, with an average of 42%. On integer applications, speedup improvements are in the 12-87% range, with an average speedup of 30%.

In summary, the monolithic six-issue core performs best when running sequential applications, followed by CoreFusion's fused core. FineGrain-2i is the worst architecture for this class of workloads. While core fusion enjoys a high core count to extract TLP, it can aggressively exploit ILP on single-threaded applications by adopting a fused configuration.

7 RELATED WORK

7.1 Reconfigurable Architectures

Several researchers have voiced the potential advantages of reconfigurable architectures capable of meeting the requirements of a diverse set of workloads. Smart memories [31] is a tiled reconfigurable architecture where each tile consists of an in-order processing core, local memory, local interconnect, and a network interface to connect to a global inter-tile network. TLP is exploited primarily

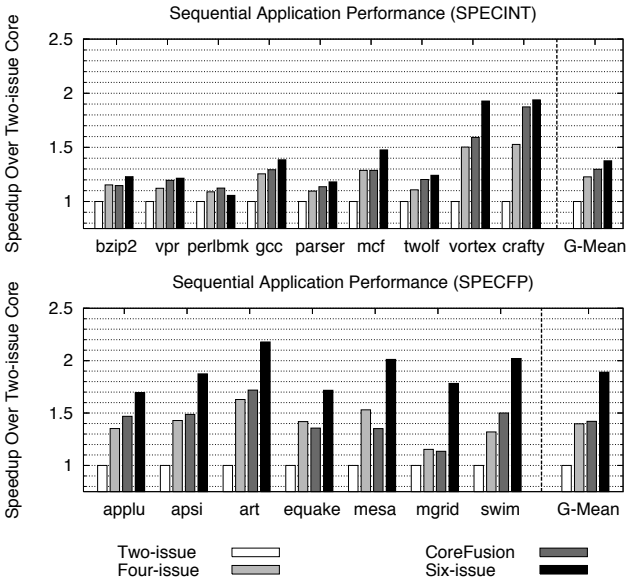

Figure 8: Speedup over FineGrain-2i on SPECINT (top) and SPECFP (bottom) benchmarks.

by using the cores as individual processing elements implementing a RISC ISA. To extract ILP, in-order cores are merged to form a VLIW machine. In contrast, core fusion merges out-of-order cores to obtain wider out-of-order execution engines while remaining transparent to existing ISAs, and this does not require recompilation. The polymorphous TRIPS architecture [38] is another reconfigurable computing paradigm that aims to meet the demands of a diverse set of applications by splitting ultra-large cores based on workload demands. TRIPS implements a custom ISA and relies heavily on compiler support for scheduling instructions to extract ILP.

7.2 Clustered Architectures

Core fusion borrows from some of the mechanisms developed in the context of clustered architectures [1, 6, 7, 10, 12, 13, 15, 18, 21, 33, 41]. Our proposal is closest to the recent thrust in clustered multithreaded processors (CMT) [17, 20, 29]. In this section, we give an overview of the designs that are most relevant to our work, and highlight the limitations that preclude these earlier proposals from supporting workload diversity effectively.

El-Moursy et al. [20] consider several alternatives for partitioning multithreaded processors. Among them, the closest one to our proposal is a CMP that comprises multiple clustered multithreaded cores (CMP-CMT). This design is attractive because it can *(a)* support large numbers of independent threads without incurring excessive design complexity, *(b)* extract high levels of ILP from sequential applications, and *(c)* support fast clock rates by

avoiding large monolithic structures. The authors evaluate CMT designs with both shared and private L1 data cache banks, finding that restricting the sharing of banks is critical for obtaining high performance with multiple independent threads. However, the memory system is not reconfigurable; in particular, there is no mechanism for merging independent cache banks when running sequential code. Consequently, sequential regions/applications can exploit only a fraction of the L1 data cache and load/store queues on a given core. Similarly, each thread is assigned its own ROB, and these ROBs cannot be merged. Finally, since the goal is to support large numbers of independent threads (as opposed to parallel applications), neither coherence nor memory consistency issues are considered. Hence, despite the attractive features listed above (which Core Fusion retains), the lack of reconfigurability in the memory system and the front-end, coupled with the lack of coherence and consistency support makes this architecture inadequate for supporting workload diversity.

Latorre et al. [29] propose a CMT design with multiple front- and back-ends, where the number of back-ends assigned to each front-end can be changed at runtime. Each front-end can fetch from only a single thread, and front-ends cannot be merged or reconfigured. When running a single thread, only one of these front- ends is active. As a result, each front-end has to be large enough to support multiple (potentially all) back- ends, and this replication results in significant area overheads (*each* front-end supports four-wide fetch, has a 512-entry ROB, a 32K-entry branch predictor, a 1K-entry i-TLB and a trace cache with 32K micro-ops). Stores allocate entries on all back-ends, and these entries are not recycled. Thus, when a thread is allocated multiple back-ends, the effective size of the store queue is the size of a single cluster's store queue. In other words, the store queue in each back-end has to be large enough to accommodate *all* of the thread's uncommitted stores. Inevitably, these inefficiencies limit the total number of threads that can be supported on the same die, thereby prohibiting the exploitation of high levels of TLP and making this architecture inadequate for supporting workload diversity.

Collins et al. [17] explore four different alternatives for clustering SMT processors. Among them, the most relevant to our work is a processor with clustered front-ends, execution units, and register files. The L1 data cache and the load/ store queues are centralized. Each front-end is capable of fetching from multiple threads, but the front-ends are not reconfigurable, and multiple front-ends cannot be merged when running a single thread. As the authors explain, the reduced fetch/rename bandwidth of each front- end can severely affect single-thread performance. Due to its inability to deliver high performance on sequential regions, this architecture is also inadequate for supporting workload diversity.

Parcerisa et al. [34] partition the front-end of a conventional clustered architecture to improve clock frequency. The front-end is designed to fetch from a single thread: parallel, evolving, or multiprogrammed workloads are not discussed and reconfiguration is not considered. The branch predictor is interleaved on high-order bits, which may result in underutilized space unless the code is very large and sparse in memory. Mechanisms for keeping consistent global histroy across different branch predictor banks are not discussed.

Chaparro et. al. [15] propose to distribute the rename map and the reorder buffer to obtain temperature reductions. Fetch and steering are centralized. Their distributed ROB expands each entry with a pointer to the next ROB entry (possibly remote) of the next dynamic instruction in program order. Committing involves pointer chasing across multiple ROBs to determine the right set of instructions. In core fusion, we depart from fundamentally independent cores and add the necessary mechanisms to achieve fusion. In particular, we also fully distribute our ROB, but without requiring expensive pointer chasing mechanisms across cores.

7.3 Other Related Work

Trace Processors [37] overcome the complexity limitations of conventional out-of-order processors by distributing instructions to processing units at the granularity of traces. The goal is the complexity-effective exploitation of ILP in sequential applications. Supporting other types of workloads (e.g., parallel codes) is not a design goal. The centralized front-end is designed to fetch from a single thread. MultiScalar processors [39] rely on compiler support to exploit ILP with distributed processing elements. The involvement of the compiler is prevalent in this approach (e.g., for register communication, task extraction, and marking potential successors of a task). On the contrary, core fusion does not require specialized compiler support. Neither multiscalar nor trace processors address the issue of accommodating workload diversity in CMPs or facilitating incremental software parallelization, which is a key focus of our work.

8 CONCLUSIONS

We have presented *core fusion*, an architectural technique that allows homogeneous CMP cores to adaptively and dynamically fuse into larger, more powerful processors to aggressively exploit ILP. Through this mechanism, several cores can be morphed into a single large execution engine when running sequential applications or regions. On the other hand, they can be used as distinct processing elements for executing parallel threads. Core fusion gracefully accommodates workload diversity and software evolution in CMPs. It requires no additional programming effort or specialized compiler support, maintains ISA compatibility, and keeps both hardware and software complexity manageable.

Our evaluation of core fusion running a mix of sequential, multiprogrammed, parallel, and evolving parallel workloads shows that core fusion effectively adapts to the diverse and changing needs of these workloads, closely tracking ideally-suited static CMP configurations, and significantly outperforming ill-suited ones at each operating point. In particular, we show that core fusion exhibits overall superior behavior with respect to static coarse-grain, fine-grain, and asymmetric CMP alternatives of similar silicon budget.

REFERENCES

[1] A. Aggarwal and M. Franklin. An empirical study of the scalability aspects of instruction distribution algorithms for clustered processors. In *International Symposium on Performance Analysis of Systems and Software*, Tucson, AZ, November 2001.

[2] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, pages 403–410, 1990.

[3] V. Aslot and R. Eigenmann. Quantitative performance analysis of the SPEC OMPM2001 benchmarks. *Scientific Programming*, 11(2):105–124, 2003.

[4] P. Bai, C. Auth, S. Balakrishnan, M. Bost, R. Brain, V. Chikarmane, R. Heussner, M. Hussein, J. Hwang, D. Ingerly, R. James, J. Jeong, C. Kenyon, E. Lee, S-H. Lee, N. Lindert, M. Liu, Z. Ma, T. Marieb, A. Murthy, R. Nagisetty, S. Natarajan, J. Neirynck, A. Ott, C. Parker, J. Sebastian, R. Shadeed, S. Sivakumar, J. Steigerwald, S. Tyagi, C. Weber, B. Woolery, A. Yeoh, K. Zhang, and M. Bohr. A 65nm logic technology featuring 35nm gate length, enhanced channel strain, 8 cu interconnect layers, low-k ild and $0.57\mu m^2$ sram cell. In *IEEE International Electron Devices Meeting*, Washington, DC, December 2005.

[5] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *International Symposium on Computer Architecture*, pages 506–517, Madison, Wisconsin, June 2005.

[6] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Dynamically managing the communication-parallelism trade-off in future clustered processors. In *International Symposium on Computer Architecture*, pages 275–287, San Diego, CA, June 2003.

[7] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *International Symposium on Microarchitecture*, pages 337–347, Monterey, CA, December 2000.

[8] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Berghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *International Symposium on Computer Architecture*, pages 282–293, Vancouver, Canada, June 2000.

[9] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser. Correlated load-address predictors. In *International Symposium on Computer Architecture*, pages 54–63, Atlanta, GA, May 1999.

[10] R. Bhargava and L. K. John. Improving dynamic cluster assignment for clustered trace cache processors. In *International Symposium on Computer Architecture*, pages 264–274, San Diego, CA, June 2003.

[11] J. Burns and J.-L. Gaudiot. Area and system clock effects on SMT/CMP processors. In *International Conference on Parallel Architectures and Compilation Techniques*, page 211, Barcelona, Spain, September 2001.

[12] R. Canal, J.-M. Parcerisa, and A. González. A cost-effective clustered architecture. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 160–168, Newport Beach, CA, October 1999.

[13] R. Canal, J. M. Parcerisa, and A. González. Dynamic cluster assignment mechanisms. In *International Symposium on High-Performance Computer Architecture*, pages 132–142, Toulouse, France, January 2000.

[14] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, San Francisco, CA, 2001.

[15] P. Chaparro, G. Magklis, J. González, and A. González. Distributing the frontend for temperature reduction. In *International Symposium on High-Performance Computer Architecture*, pages 61–70, San Francisco, CA, February 2005.

[16] G. Chrysos and J. Emer. Memory dependence prediction using store sets. In *International Symposium on Computer Architecture*, pages 142–153, Barcelona, Spain, June–July 1998.

[17] J. D. Collins and D. M. Tullsen. Clustered multithreaded architectures - pursuing both ipc and cycle time. In *International Parallel and Distributed Processing Symposium*, Santa Fe, New Mexico, April 2004.

[18] J. Cong, A. Jagannathan, G. Reinman, and Y. Tamir. A communication-centric approach to instruction steering for future clustered processors. In *Watson Conference on the Interaction between Architecture, Circuits, and Compilers*, Yorktown Heights, NY, October 2004.

[19] J. D. Davis, J. Laudon, and K. Olukotun. Maximizing cmp throughput with mediocre cores. In *International Conference on Parallel Architectures and Compilation Techniques*, Saint Louis, MO, September 2005.

[20] A. E.-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas. Partitioning multi-threaded processors with a large number of threads. In *International Symposium on Performance Analysis of Systems and Software*, pages 112–123, Austin, TX, March 2005.

[21] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The Multicluster architecture: Reducing cycle time through partitioning. In *International Symposium on Microarchitecture*, pages 149–159, Research Triangle Park, NC, December 1997.

[22] J. González, F. Latorre, and A. González. Cache organizations for clustered microarchitectures. In *Workshop on Memory Performance Issues*, pages 46–55, Munich, Germany, June 2004.

[23] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[24] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 9(2):24–36, March 1999.

[25] A. KleinOsowski and D. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[26] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *International Symposium on Microarchitecture*, pages 81–92, San Diego, CA, December 2003.

[27] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *International Symposium on Computer Architecture*, pages 64–75, München, Germany, June 2004.

[28] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *International Symposium on Computer Architecture*, pages 408–419, Madison, Wisconsin, June 2005.

[29] F. Latorre, J. González, and A. González. Back-end assignment schemes for clustered multithreaded processors. In *International Conference on Supercomputing*, pages 316–325, Malo, France, June–July 2004.

[30] R. Lawrence, G. Almasi, and H. Rushmeier. A scalable parallel algorithm for self-organizing maps with applications to sparse data mining problems. Technical report, IBM, January 1998.

[31] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: a modular reconfigurable architecture. In *International Symposium on Computer Architecture*, pages 161–171, Vancouver, Canada, June 2000.

[32] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, Cambridge, MA, October 1996.

[33] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *International Symposium on Computer Architecture*, pages 206–218, Denver, CO, June 1997.

[34] J. M. Parcesira. *Design of Clustered Superscalar Microarchitectures*. Ph.D. dissertation, Univ. Politècnica de Catalunya, 2004.

[35] J. Pisharath, Y. Liu, W.-K. Liao, A. Choudhary, G. Memik, and J. Parhi. NU-MineBench 2.0. Technical Report CUCIS-2005-08-01, Center for Ultra-Scale Computing and Information Security, Northwestern University, August 2005.

[36] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. `http://sesc.sourceforge.net`.

[37] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. Trace processors. In *International Symposium on Microarchitecture*, pages 138–148, Research Triangle Park, NC, December 1997.

[38] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *International Symposium on Computer Architecture*, pages 422–433, San Diego, CA, June 2003.

[39] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *International Symposium on Computer Architecture*, pages 414–425, Santa Margherita Ligure, Italy, June 1995.

[40] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[41] V. V. Zyuban and P. M. Kogge. Inherently lower-power high-performance superscalar architectures. *IEEE Transactions on Computers*, 50(3):268–285, March 2001.

Designing Hardware that Supports Cycle-Accurate Deterministic Replay

Brian Greskamp, Smruti R. Sarangi, and Josep Torrellas
Department of Computer Science, University of Illinois
http://iacoma.cs.uiuc.edu

Abstract

Most computer hardware today is nondeterministic, *meaning that two executions of a program will not be cycle-for-cycle identical at the microarchitectural level even if they start from the same microarchitectural state. Due to uninitialized state elements, I/O, and timing variations on high-speed buses, the microarchitectural states of the two executions will evolve differently.*

Such nondeterminism complicates system verification and makes hardware faults detected during bringup more difficult to reproduce and analyze. Consequently, we believe that board-level computer hardware should be designed in a way that supports cycle-accurate deterministic replay. In this paper, we outline the hardware required to provide this capability. We argue that the resulting hardware complexity is minimal, providing a net savings in bringup time and cost. We also show that potential applications of deterministic hardware extend far beyond hardware verification.

1. Introduction and Motivation

We propose the *Cycle-Accurate Deterministic REplay* (CADRE) architecture, which cost-effectively makes a board-level computer cycle-deterministic — including processors, buses, memory, chipset, and I/O devices. CADRE uses checkpoints, logs, and certain hardware extensions to enable replayed executions that match the microarchitectural state of the original execution cycle-for-cycle. For example, assume that one of the processors in the computer observes a bus signal transition A at internal cycle a and initiates an ALU operation B at cycle b. These events will recur at exactly the same internal cycles during the re-execution. Further, the microarchitectural states of the multiple processors, memory controllers, and other components will evolve exactly as they did during the original execution.

Cycle-accurate determinism has many applications, but one of the most obvious is in system bringup — the verification phase when engineers begin running programs on first silicon. Since the real processor is so much faster than the simulators used in earlier verification phases, longer and more detailed tests, such as booting a full operating system, can finally be executed. The bringup tests quickly reveal many previously unknown bugs, which must be characterized. The characterization process typically begins with finding a way to reliably reproduce an error. The engineer can then employ "iterative debugging" — replaying the error and examining system state before and after to gain a full understanding of the problem. With typical hardware, finding a test that reliably reproduces the error is difficult or impossible, but with CADRE, it is trivial. With CADRE, an engineer can replay a failing test over and over, with the assurance that at each cycle, the signal and state transitions will exactly match those of the original execution. He can then stop the machine at different points and examine the internal state through a test access port or read out the complete system state at any point and transfer it to an RTL simulator for detailed analysis.

Deterministic hardware is also easier to test than nondeterministic hardware. Already, automatic testers are encountering problems with nondeterminism [5]. These testers operate by presenting test vectors at the chip's input pins and observing the response vectors on the output pins. In a nondeterministic system, response vectors may not arrive at the tester at the expected time, or even in the expected order. In extreme cases, the data in the response vectors could differ from the expected values. Cycle-accurate deterministic hardware does not present these problems.

CADRE is not just for verification and test; it can be deployed in the field, providing hardware vendors with a powerful tool to debug customer-site failures. After the customer identifies what he believes to be a hardware error, he could send the vendor a checkpoint preceding the crash. The vendor would then be able to reproduce the fault exactly using in-house hardware and simulators. The idea is similar to the current use of software crash feedback agents that help software developers identify bugs in deployed software.

Cycle determinism also has less obvious applications. For example, a cycle-deterministic system is easier to incorporate into an n-way modular redundancy system. Traditional NMR systems, such as HP's NonStop server [2], require custom buffering and synchronization logic between the processors and voters because each processor may slowly slip behind or ahead of the others. Cycle-deterministic components do not require such compensation logic in NMR configurations, as cycle determinism ensures that as long as the components have the same inputs, they will continue in lock step.

Finally, hardware deterministic replay subsumes previous proposals for software determinism to debug parallel programs [11]. In a cycle-accurate deterministic system, the interleaving of replayed memory accesses is guaranteed to match the original, since all access occur at exactly the same cycle as during the original execution. Furthermore, as we will show later, interrupts and I/O events will also recur exactly where they should. One issue with this approach is that special measures are needed to allow a debugger to run on the target machine without interfering with hardware determinism during replay. A solution to this problem is to run the debugger on another machine attached to the target's front side bus or test access port.

2. Sources of Nondeterminism

A system supporting deterministic replay must have two key properties: (1) A deterministic execution interval must begin at cycle 0 with each state-holding element initialized to a known state. (2) A component must receive a signal at the nth edge of its local clock during replay iff it received the same signal at the nth edge during the original execution. The first condition is the base case, requiring that the original and replay executions start at exactly the same state, and the second is the inductive step, ensuring that they experience the same state transitions at the same cycles.

All nondeterminism is then traceable to one of two causes: (1) incomplete initialization, in which some state-holding elements are in an incorrect or unknown state at the start of replay, or (2) changes in the arrival times of signals, possibly due to environmental factors. Below, we discuss the two causes as they apply to each component of the system.

CPUs Modern processors contain millions of bits of state contained in registers, pipeline latches, SRAMs, and counters. Some state bits, like those in the branch predictor tables, have no architecturally-visible effect. Consequently, processors usually do not provide any ISA-level means of resetting them, violating condition 1. Additionally, dynamic power and temperature management techniques such as clock duty cycle modulation and voltage-frequency scaling (DVFS) are dependent on the environment (die temperature). As a result, the timing of power and temperature events is uncertain and condition 2 is violated.

Memory Systems The memory controller is the component responsible for scheduling memory read, write, refresh, and scrubbing operations. The Itanium-2 verification engineers [4] reported that the memory refresh and scrubbing operations are a source of nondeterminism because the scrubber and refresh walkers will be working on different lines during the re-execution than in the original. Therefore, scrubs and refreshes line up with the program's read and write accesses differently during replay. Due to contention, the timing of all memory operations will change.

I/O and Interrupts The timing of I/O operations and interrupts is notoriously unpredictable. For example, hard disks have mechanical components that introduce non-deterministic seek times and rotational delays. The timing of events from human-interface devices and network interfaces is equally unpredictable.

Buses The buses that cross clock domains in a computer, for example as they connect different chips, are a major source of nondeterminism. These buses are often source-synchronous [1], which means that the transmitter generates and transmits a clock signal that travels with the data to the receiver. One popular example is HyperTransport [3]. In these buses, receiving a message occurs in two steps (Figure 1). First, the rising edge of the transmitter clock signal latches the data into a holding queue in the bus interface of the receiver. We refer to this event as the *arrival* of the message. Some time later, normally on the next rising edge of the receiver's core clock, the receiver removes the data from the queue and submits it for processing. We refer to this event as the *processing* of the message.

Unfortunately, the exact arrival time is nondeterministic because all clock and data pulses that traverse the bus are affected by physical and electrical processes such as temperature variations, voltage variations, channel cross talk, and inter-symbol interference [1, 3, 8]. As a result, these signals experience a random but bounded delay on the bus and could arrive at any time during a certain window. For example, the HyperTransport specification assumes an uncertainty interval of one cycle even for very short buses [3]. Clearly, this is a violation of condition 2.

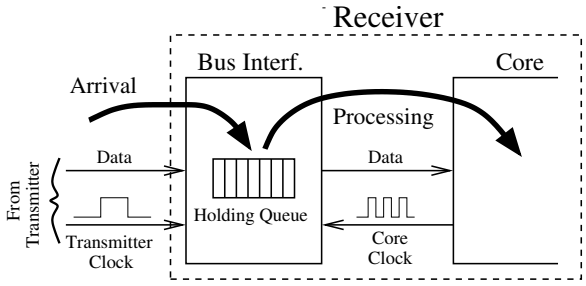

Figure 1. Arrival and processing of a message at the receiver.

Figure 2 illustrates how uncertainty in the arrival time of the transmitter clock can give rise to nondeterminism at the receiver. The receiver may see the rising edge of the transmitter clock arrive anywhere in the hatched interval. If the receiver processes the message on the first rising edge of the core clock after arrival, then the processing time is nondeterministic because it depends on the arrival time.

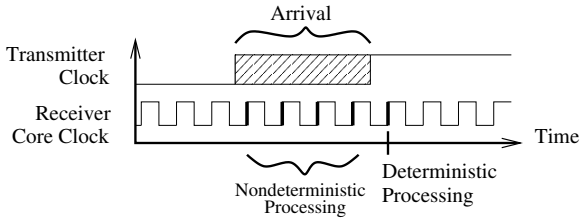

Figure 2. Nondeterministic and deterministic processing.

3. Ensuring Cycle Determinism in Buses

To make bus transfers fully cycle-deterministic, we propose to delay the *processing* of a message at the receiver until the last possible core clock cycle at which the message could have arrived. The correct processing time is shown in Figure 2 as "deterministic processing". The cost of this approach is a small increase in latency for some messages.

Our scheme works in any system where the ratio of the frequencies in the transmitter T and receiver R is constant, although the relative phase of the clocks may change with time (within bounds) due to physical and electrical effects. However, for simplicity, this paper will assume that T and R operate at the same frequency.

CADRE adds a *domain-clock* counter — an up-counter driven by the local clock signal — to both the transmitter and the receiver. At periodic global machine checkpoints (once per second), a broadcast signal resets all domain-clock counters. At any time, the difference between the transmitter's and receiver's domain-clock counts is bounded by $[p, q]$. Additionally, the transmission delay of a message on the bus (measured in domain-clock counts) is bounded by $[d_1, d_2]$. The constants d_1, d_2, p, and q are known at design time. As a result, if the transmitter sends a message at count x_T of its

domain-clock counter, the message will *arrive* at the receiver at count y_R of the receiver's domain-clock counter, as given by:

$$y_R = x_T + [d_1 + p, d_2 + q] = x_T + [\theta_1, \theta_2] \quad (1)$$

We call $\theta_2 - \theta_1$ the *Uncertainty Interval*.

Our scheme to enforce bus determinism is detailed in [7]. It requires that the transmitter include in every message a short tag ρ (typically 1 or 2 bits) that allows the receiver to determine when the message was sent (x_T). Then, the receiver simply computes $z_R = x_T + \theta_2$ and delays the *processing* of the message until z_R, ensuring determinism.

The hardware required is shown in Figure 3. Our scheme adds a *Synchronizer* module to the bus interface of the receiver. This hardware processes the tag ρ arriving with the data, and uses it to determine at what cycle z_R to process the data. Full details are in [7].

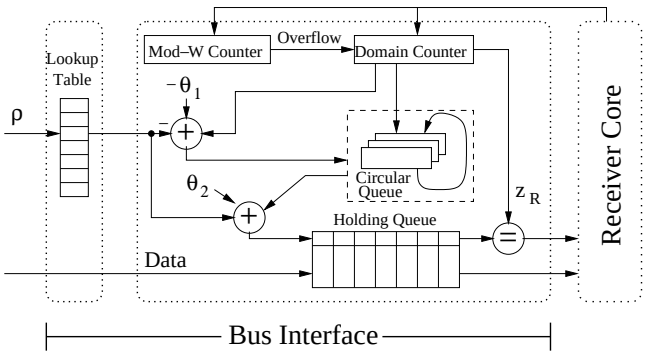

Figure 3. Synchronizer module added to the bus interface of the receiver.

4. Overall Deterministic System

To design a CADRE system, we build on the checkpointing and logging mechanisms from ReVive [6] or SafetyNet [10]. The idea is as follows. Periodically — say, once per second — processors write back their caches to memory and invalidate caches and TLBs. They then save their registers and completely re-initialize all internal state-holding elements to a known state. As execution proceeds after the checkpoint, when a main-memory location is about to be over-written for the first time since the checkpoint, the old value of that location is saved in a Memory Log. This is done in hardware by the memory controller. As discussed in [6, 10], this support enables memory state rollback.

To make each CPU deterministic, we introduce the `DETRST` instruction, which initializes all the state elements in the processor. `DETRST` is executed after every checkpoint. Moreover, each CPU is augmented with a CPU Log that records a variety of events, such as (i) clock duty cycle modulation, (ii) voltage-frequency scaling, and (iii) nondeterministic interrupts and exceptions generated inside the processor chip. Examples of the latter are thermal emergencies and ECC failures due to soft errors. During re-execution, events in the log are replayed to reproduce the events in the original execution.

To make the memory deterministic, we make two changes to the memory controller. First, the controller makes memory refresh deterministic by resetting the refresh logic at each checkpoint. In this case, if all of the inputs to the memory controller are deterministic, the refresh logic will generate deterministic outputs. As long as the checkpoint interval is long enough to allow at least one refresh operation to complete per memory location, the DRAM will not lose data. Moreover, to circumvent nondeterminism from memory scrubbing, the controller includes in the checkpoint the register that indexes the currently scrubbed line. When restoring the checkpoint, the register is restored, enabling scrubbing to resume from exactly where it was in the original execution.

Since I/O devices are inherently nondeterministic, CADRE uses a logging-based solution. Specifically, CADRE places a buffering module in the memory controller called the Input Log. The Input Log records all the messages arriving from the I/O devices and the interrupts that the I/O devices deliver. When replaying an execution, the I/O devices can simply be suspended by gating their clock and disconnecting them temporarily from the data bus. The Input Log will reproduce all of the signals that the I/O devices generated during the original execution.

Finally, to enforce determinism in source-synchronous buses, we use the module shown in Figure 3 at the receiver side of each bus. If a bus is bidirectional, CADRE places one such module at each end of the bus.

5. Feasibility

Possible concerns about CADRE include the added chip area, design complexity, storage overhead, and performance overhead. The area overhead of the added CADRE logic is very small; the bus synchronizers comprise fewer than a thousand gates each, and the memory and IO log controllers are also tiny. Only the Input Log, which is implemented in SRAM, consumes significant space. As for complexity, we feel that any additional design effort for a CADRE system is more than offset by the improvements in verification efficiency that cycle-accurate determinism provides.

The storage overhead is composed of the Input Log, CPU Logs, and the ReVive/SafetyNet Memory Log. The latter is estimated to be around 50 MB/s per processor in [11]. Although the size of the Input Log varies with the application, we found it to be quite low for a set of workloads that include SPECint, SPECfp, SPEComp, SPECjbb, and SPECweb. While some applications may require up to 100 MB/s during periods of high activity, no application exceeded 1 MB/s of input log bandwidth in the steady state. Finally, the storage cost of the CPU Log is negligible since frequency scaling and thermal events are rare in current processors.

The main contributor to performance overhead is the increased memory latency introduced by the bus synchronizers that make the path from the processor to memory deterministic. Other costs, such as flushing caches at checkpoints, are negligible with checkpoint intervals of one second or longer. So, assume that the memory controller is on a different chip than the processor and that the interconnection of the processor, memory controller, and memory modules is through HyperTransport links. Given current technology [3], the bus synchronizer on each link will add one cycle to each message in the worst case. We therefore consider a worst case of four additional bus cycles for each memory access. Our simulation results

show that the resulting slowdowns on SPECjbb, SPEComp, SPECcpu, and SPECweb are all less than 1%.

To summarize, extending a four-way CMP server with CADRE hardware supporting a one second checkpoint interval would have a storage cost of about 200 MB of DRAM plus a few MB of SRAM (for the Input Log), a performance overhead of 1%, and a small area cost. For that price, we obtain the ability to "rewind" execution to a checkpoint one second in the past and re-execute deterministically cycle-for-cycle.

6. Related Work

The state of the art in deterministic replay for hardware debugging is Golan, a hardware testbed used to debug the Pentium-M processor [9]. Golan attaches a logic analyzer to the pins of the processor chip. Every input signal arriving at the pins is logged. This includes data to satisfy cache misses. Like CADRE, Golan takes a periodic checkpoint, which involves invalidating caches and TLBs, saving the processor registers, and resetting the processor state. Upon detection of a failure, Golan restores a checkpoint and restarts execution while replaying the logic analyzer log.

A shortcoming of the Golan approach is that the checkpoint interval (and therefore replay distance) is much shorter and storage overhead is much greater than in CADRE. Additionally, the probes that attach the logic analyzer to the processor pins present a difficult electrical design problem because the pins cycle at high frequency and no nondeterminism in the connection can be tolerated. Although Golan was highly successful in speeding Pentium-M bringup, it is not suitable for field deployment.

7. Conclusions

We have proposed that hardware that enforces cycle-accurate determinism be included in commodity computer systems to ease verification and testing, and for other purposes. We have explained what the main sources of nondeterminism are and how they can be circumvented. Finally, we have argued that the area, complexity, storage, and performance cost of the required hardware is minimal.

References

[1] W. J. Dally and J. W. Poulton. *Digital Systems Engineering*. Cambridge University Press, 1998.

[2] D. Bernick et al. NonStop advanced architecture. In *DSN*, pages 12–21, 2005.

[3] HyperTransport Technology Consortium. HyperTransport I/O link specification revision 2.00b, 2005.

[4] D. D. Josephson, S. Poehhnan, and V. Govan. Debug methodology for the McKinley processor. In *ITC*, pages 451–460, 2001.

[5] K. Mohanram and N. A. Touba. Eliminating non-determinism during test of high-speed source synchronous differential buses. In *VTS*, pages 121–127, 2003.

[6] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *ISCA*, pages 111–122, 2002.

[7] S. R. Sarangi, B. Greskamp, and J. Torrellas. CADRE: Cycle-accurate deterministic replay for hardware debugging. In *DSN*, 2006.

[8] L. Sartori and B. G. West. The path to one-picosecond accuracy. In *ITC*, pages 619–627, 2000.

[9] I. Silas et al. System level validation of the Intel Pentium-M processor. *Intel Technology Journal*, 7(2), May 2003.

[10] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA*, pages 123–134, 2002.

[11] M. Xu, R. Bodík, and M. D. Hill. A "Flight Data Recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

Leveraging Bloom Filters for Smart Search Within NUCA Caches

Robert Ricci, Steve Barrus, Dan Gebhardt, and Rajeev Balasubramonian
School of Computing, University of Utah
{ricci,sbarrus,gebhardt,rajeev}@cs.utah.edu

Abstract

On-chip wire delays are becoming increasingly problematic in modern microprocessors. To alleviate the negative effect of wire delays, architects have considered splitting up large L2/L3 caches into several banks, with each bank having a different access latency depending on its physical proximity to the core. In particular, several recent papers have considered dynamic non-uniform cache architectures (D-NUCA) for chip multi-processors. These caches are dynamic in the sense that cache lines may migrate towards the cores that access them most frequently. In order to realize the benefits of data migration, however, a "smart search" mechanism for finding the location of a given cache line is necessary. These papers assume an oracle and leave the smart search for future work. Existing search mechanisms either entail high performance overheads or inordinate storage overheads. In this paper, we propose a smart search mechanism, based on Bloom filters. Our approach is complexity-effective: it has the potential to reduce the performance and storage overheads of D-NUCA implementations. Also, Bloom filters are simple structures that incur little design complexity. We present the results of our initial explorations, showing the promise of our novel search mechanism.

1 Introduction

It is well-known that on-chip wire delays are emerging as a major bottleneck in the design of high-performance microprocessor chips. As feature sizes are reduced, wire delays do not scale down at the same rate as logic delays [1, 5]. It has been projected that at 35nm technologies, less than 1% of the total chip area will be reachable in a single cycle [1]. Communication between distant modules on a chip will therefore cost tens of cycles and will negatively impact performance.

On-chip cache hierarchies bear the brunt of growing wire delays as they occupy a large fraction of chip area in modern microprocessors. For example, more than two-thirds of the chip area in Intel's Montecito [11, 12] can be attributed to L3 caches that have a capacity of 24MB. Such large cache structures are typically organized as numerous banks to help reduce latency and power consumption [14]. Given an input address, the request is routed to a subset of banks that then service the request. The latency for any cache access is a function of the distance between the bank that contains the requested data and the cache controller. This observation motivated the proposal by Kim *et al.* [7] of a non-uniform cache architecture (NUCA). Within a NUCA organization, the latency for a cache access may be as little as a handful of cycles if the data is located close to the cache controller, or up to 60 cycles if the data is located in a distant bank. This architecture is unlike a conventional cache organization where the cache latency is uniform and determined by the worst-case delay to access any block. Recent proposals have extended NUCA designs to also handle chip multiprocessors [2, 4, 6].

NUCA organizations have been classified as static (S-NUCA) and dynamic (D-NUCA) in the literature [7]. In static-NUCA, an address is mapped to a unique cache bank. Given an address, the cache controller sends the request to a single bank (typically determined by examining the address index bits). While such a mechanism is simple, it does not take advantage of locality. The L2 or L3 cache latency for a data structure is set as soon as it is allocated in the physical memory address space. Dynamic-NUCA attempts to improve performance by leveraging locality and moving recently accessed blocks to banks that are close to

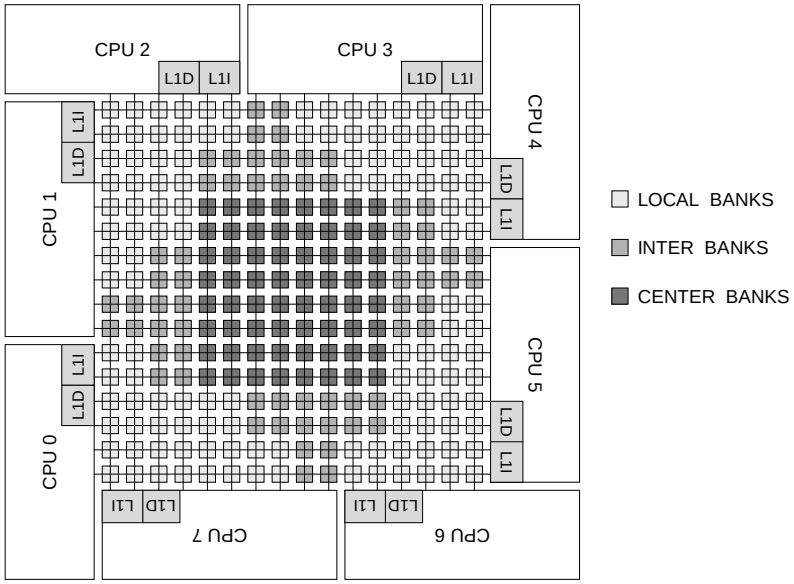

Figure 1. Baseline CMP with 8 cores that share a NUCA L2 cache. The L2 is partitioned into 256 banks and each block is allowed to reside in one of 16 possible banks.

the cache controller. A block is now allowed to reside in different cache banks at different times. Given an address, the cache controller identifies a number of candidate banks to which the request can be sent. In one approach, the banks can be sequentially probed until the data is located – this can significantly increase cache access latency. In a second approach, the banks can be probed in parallel – this can increase contention cycles because of the increased bandwidth pressure on the inter-bank network. Hybrids of the two search approaches have also been proposed [2, 7].

A recent paper explores block migration policies for a D-NUCA organization in a CMP [2]. The authors show that D-NUCA can improve performance, relative to an S-NUCA organization, provided there exists an oracle to identify the bank that stores the data. If a realistic hybrid data search mechanism is incorporated, performance is actually worse than that of S-NUCA. Hence, for D-NUCA to be effective, the cache controller must identify a small subset of banks to probe, with high accuracy. Kim *et al.* [7] propose one mechanism for such a smart search for a single-core chip. The cache controller maintains a partial tag array that stores six bits of the tag for each cache line. The cache controller then forwards the request to only those banks that have tags that match the address. While this mechanism has high accuracy, Beckmann and Wood [2] point out that such partial tag arrays can lead to extremely high storage overheads. In an 8-core CMP with 16MB of cache and block size of 64 bytes, the total storage for 6-bit partial tags is as high as 1.5MB (roughly 72M transistors). The efficient design of a "D-NUCA smart search mechanism" is considered an open problem and an important factor in the success of block migration policies [2].

This paper presents a complexity-effective solution to the smart search problem. It takes advantage of Bloom filters to identify candidate cache banks with high accuracy. Compared to partial tag arrays, it reduces storage requirements by an order of magnitude. Power consumption can also be reduced by avoiding access to large tag arrays and by avoiding transmission of tags on the inter-bank network. This reduction in network traffic decreases routing congestion and can improve latency for all L2 transactions. Bloom filter updates and look-ups are achieved with simple indexing functions.

Section 2 provides background on Bloom filters and the baseline NUCA organization. Section 3 details our proposed smart search mechanism. We present a preliminary analysis of our approach in Section 4 and draw conclusions in Section 5.

2 Background and Related Work

2.1 Non-Uniform Cache Architectures

The baseline processor organization that we use is similar to that of Beckmann and Wood [2] and is illustrated in Figure 1. Each processor core (including L1 data and instruction caches) is placed on the chip boundary and eight such cores surround a shared L2 cache. The L2 is partitioned into 256 banks and connected with a mesh interconnection network. Each core has a cache controller that routes the core's requests to appropriate cache banks.

In a static-NUCA organization, eight bits of the block's physical address can be used to identify the unique bank that the block maps to. Each bank will have to be set-associative to reduce conflict misses.

In an alternative static-NUCA organization, each bank can accommodate a single way. If the L2 cache is 16-way set-associative, a given block can map to 16 possible banks and four bits of the block's physical address are used to identify this subset of banks. When a block is brought into the cache, LRU (or even the block's address) can determine where the block is placed and the block remains there until it is evicted. This S-NUCA organization is no better than the organization described in the previous paragraph (in terms of performance) and requires a "smart search mechanism" to identify the bank that contains the block. We describe it here because it forms the basis for dynamic-NUCA organizations.

In dynamic-NUCA, LRU/block address determines where the block is initially placed. Access counters associated with each cache line keep track of the processor cores that request the block. The block is gradually moved to a bank that best reflects the "center of gravity of processor requests". Of the 16 candidate cache banks (ways) for a block, eight are in close proximity to each of the eight cores (referred to as the *local* banks), four are in the *center* region, and the remaining four are in the *intermediate* region (shown in Figure 1 by different colors). These 16 banks are classified as a single *bankset* and there exist 16 such banksets. The L2 cache is partitioned into 16 *bankclusters*. Each bankcluster contains one bank from every bankset. Block migration causes a block to move between bankclusters, or stated alternatively, between banks within a bankset.

Given a block address, the cache controller must potentially forward the request to 16 different banks in order to locate the block. The smart search mechanism proposed in this paper will be employed in this context. Beckmann and Wood adopt the following mechanism: the core's local, intermediate, and four center banks are searched in parallel; if the block is not located, the other ten candidate banks are searched in parallel. Such a mechanism entails high performance overheads and usually negates any performance improvements from block migration. The search mechanism of Kim *et al.* [7] maintains partial tag arrays at the cache controller to identify a small subset of banks that can be probed in parallel. Such a mechanism, extended to CMPs, would entail marginal performance overheads, but incur non-trivial storage overheads. Our proposed mechanism has the potential to incur marginal performance and storage overheads.

Block migration incurs other complexities as well (regardless of the search mechanism). Firstly, access counters must be maintained for every cache line (or blocks can be aggressively migrated on every access). Block migration cannot happen simultaneously with block look-up, else there is the potential to signal a false miss (the request fails to see the block in transit and triggers an L2 miss). A four-phase protocol is required to implement migration correctly: (i) cores are informed of the migration so that accesses to that block are temporarily disabled, (ii) acknowledgments are received from the cores, (iii) migration is effected, (iv) cores are informed after migration so that accesses can be enabled again. Block migration requires that the search hints at each core be updated. The search mechanism may dictate the amount of network traffic required to update search hints.

2.2 Bloom Filters

We base our smart search algorithm on the concept of Bloom filters [3]. Here, we describe general Bloom filters. The particular variant that we use is detailed in Section 3.3.

A Bloom filter is a structure for maintaining probabilistic set membership. It trades off a small chance of false positives for a very compact representation. A general Bloom filter cannot return false negatives.

A Bloom filter consists of an array A of m one-bit entries and k hash functions, $\{h_1, h_2, ...h_k\}$. In an empty filter, all bits in A are zero. To add item i to

the filter, $h_1(i)$ is computed. This value is then used to index into A, and $A[h_1(i)]$ is set to one. This process is repeated with each hash function, up to $h_k(i)$.

Testing for set membership is straightforward. To test for the presence of i', we examine $A[h_1(i')]$. If it is zero, then we can tell that i' is not present in the set. If it is one, we continue to check $A[h_2(i')]$ and so on, up to $A[h_k(i')]$. If all such bits are one, the set membership test returns $true$, but if any are zero, the membership test returns $false$.

In set membership tests, false negatives are impossible—if item i has been added to the filter, then we are guaranteed that:

$$\forall j\{1 <= j <= k : A[h_j(i)] = 1\}$$

False positives, however, are possible; any or all of the bits in A could have been set to one by the insertion of a different item. Assuming "good" hash functions, the probability of getting a false positive from a Bloom filter is approximately $1 - e^{-kn/m}$ [3], where n is the number of items inserted into the set, k is the number of hash functions, and m is the size of array A. For example, for $k = 5, m = 2048$, and $n = 256$, the probability of a false positive is 2.1%. Such a filter requires only 2048 bits of space, while an equivalent table with 256 entries for 32-bit numbers would require 8192 bits of space.

It is not possible to remove elements from a Bloom filter. If we were to try to do so, setting any of the bits in A to zero could interfere with another element in the set, causing a false negative. Removal requires a "counting" Bloom filter, in which we keep counts of how many elements in the set require a particular bit in A to be set. Such filters, however, have higher storage requirements, because each entry in A must now hold a counter rather than a single bit. Thus, we do not use counting Bloom filters in this paper.

3 Smart Search Design

Our smart search algorithm is based on the idea of keeping *approximate* information about the contents of L2 bankclusters. Kim *et al.*'s [7] partial tag array uses a similar strategy, though in the context of single-core chips. Such arrays would be prohibitively large for processors with many cores and many bankclusters. Thus, we turn to Bloom filters as a compact approximation.

3.1 Overall Design

At each core on the CMP, we maintain an approximation of the cache lines present in each L2 bankcluster. For our baseline processor (Figure 1), each of the 8 cores maintains 16 filters, one for each L2 bankcluster. While the number of filters is large, as we shall see shortly, each filter is a relatively small structure.

The filters are used to direct L1 misses to the L2 bank or banks that seem likely to contain the requested cache line. The search proceeds in a similar manner to Beckmann and Wood's [2]; when a core's memory access misses in L1, the core's cache controller looks for the cache line in the L2 bankclusters. First, it consults the core's local L2 bankcluster, the Center, and the Inter bankclusters. If migration has had its intended effect, these are the most likely places to find the line. If none of those bankclusters hit, the core tries the other cores' local bankclusters before going to main memory. In our search, we filter out request messages by consulting, in parallel, the Bloom filters for each bankcluster, and only sending messages to those that hit. Due to the possibility of false negatives, discussed in the next section, if no Bloom filters hit, or all of the bankclusters consulted turn out to be false positives, we send messages to all bankclusters skipped in the first pass.

3.2 Managing the Filters

We next deal with content management: when are items inserted into filter, and when are they removed? The design principle we follow here is to keep complexity low by minimizing our changes to the baseline cache design. Thus, rather than adding new messages for filter management, we use the messages already sent by the cache. When one of the L2 banks inserts a new line, we do not broadcast this event to all filters. Instead, we only insert addresses into our filters on-demand; that is, when a core receives a line from some L2 bank. The first time a cache line is accessed by a given core, therefore, there is a compulsory miss in the filters. For this reason, our filters can return a false negative. We also see similar false negatives when a cache line has migrated from one bankcluster to another. Because the probability of false positives returned by a Bloom filter is a function of the number of items inserted into it, inserting no more items than necessary will help keep the false positive rate low.

All filters start empty, with the bits in their arrays cleared. The first time core c accesses line l, it will have no entry for l in any of its filters. Assuming no false positives, all filters will miss, and c will fall back to searching all banks for l. If l is present in bankcluster b, b will send the line to c. Upon receipt of the line, c inserts l into its filter for bankcluster b. Thus, the next time that c misses on l, it will know that b has at some point held line l.

As time goes on, the filters will tend to return more false positives for two reasons. First, due to the fundamental properties of Bloom filters, as more items are inserted into a filter, its false positive rate rises. Second, as time passes, the information in the filters becomes stale; some of the entries they have stored will no longer be accurate because lines may have migrated or been otherwise evicted from their original bankclusters.

We clearly must have a mechanism in place for removing filter entries to avoid a constantly-rising false positive rate. Recall that in a simple Bloom filter, entries cannot be removed. In addition, if we were to track migrations or evictions, this would require new messages between bankclusters and the filters.

In order to keep the complexity of our design down, strategy we adopt is to clear an entire filter when its false positive rate becomes too high. This decision can be made locally by each filter, without requiring global co-ordination. Filters can, for example, maintain a simple n-bit saturating counter that is incremented on each true positive and decremented on each false positive. This provides an approximation of the false positive rate over the last several memory accesses. For our initial implementation, we clear a filter when its ratio of false positives to true positives goes above one—that is, when the false positive rate reaches 50%. There is clearly a tradeoff between clearing the filters frequently to reduce false positives and clearing them infrequently to minimize the number of compulsory misses thus incurred. This is a tradeoff we will explore in future work.

3.3 Filter Design

The filter we chose is similar to the partitioned-address Bloom filter used by Peir *et al.* [13] for predicting cache misses.

The bit array, A is divided into k slices, $A_1, A_2, ...A_k$. Each hash function indexes into its corresponding slice. For example, h_1 indexes into A_1, h_2 indexes into A_2, and so forth. Each slice can be implemented as a separate structure, keeping the time required to access it small. All slices can be accessed in parallel, meaning that the lookup time is independent of the number of slices.

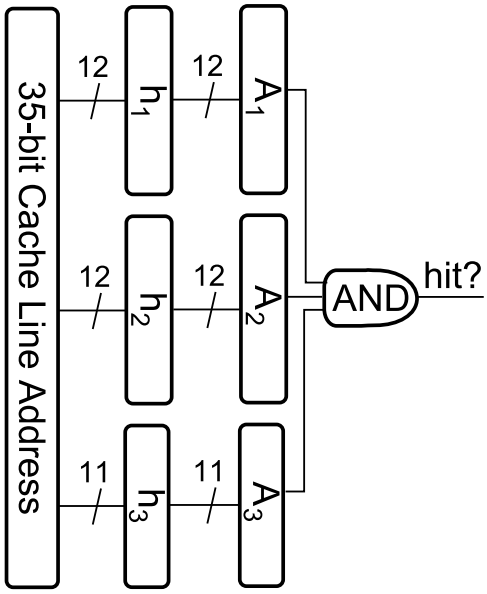

Figure 2. Design of our Bloom filter, showing hash functions h_1 through h_3, which are used as indexes into bit array slices A_1 through A_3. A hit is detected when ones are returned from all array slices.

We use the simplest hash function possible, the identity function. We split up the address into several sets of bits, then use each to index into one of the array slices.

The layout of our filter is depicted in Figure 2. We assume a physical address width of 41 bits—this is one bit wider than the current AMD Athlon 64 and Sun Niagara [8] processors, and can accommodate two terabytes of memory. We also assume 64 byte wide cache lines, giving 6-bit offsets. This leaves us with 35-bit cache line addresses. We divide these bits into two groups of 12 and one group of 11, and use each group to index into an array slice. Our design can easily be adapted to other address widths by changing the number and size of the array slices.

Our filters are reasonably sized structures; there are two arrays of 4096 bits each, and one array of 2048 bits. The total size for one filter is thus 10 Kb. Since

each core needs 16 filters, one for each bankcluster, each core requires 160 Kb, and all eight cores together require 1280 Kb, or 160 KB, of storage. Assuming 6-transistor SRAM cells, the total filter size is approximately 7.68 million transistors.

In comparison, if we were to extrapolate the 6-bit partial tag array used as a filter by Kim *et al.* [7] to an 8-way CMP with 16 banksets, it would require 1.5MB of storage (approximately 72 million transistors). The physical layout for both of these approaches can be organized as a roughly square memory structure. Our structure requires proportionally more space for the decoders and other RAM components, but we estimate they should not be more than 20% of the storage transistors. Thus, our design has much lower overhead, at least eight times smaller, than the existing work in this area.

Filter sizes and hash functions different from the ones we use in this paper may result in better performance or smaller filters—we leave a careful study of filter sizes and hash functions to future work.

4 Results

We now present the results of our initial exploration. We have implemented our Bloom filter smart search using the GEMS 1.2 [10] toolset. The *Ruby* cache model of GEMS includes the D-NUCA design we have described as our baseline. GEMS interfaces with the full-system functional simulator, Simics [9]. For evaluation, we used 12 benchmarks from the SPLASH-2 [15] parallel multithreaded program set. We start with a cold cache in order to take into account compulsory misses in the filters. All programs ran for at least one billion instructions, and at least 500 thousand L2 accesses.

Our initial implementation tracks the management of the Bloom filters and records their accuracy, but does not yet modify the search itself. Thus, the results we present here reflect the accuracy of the filters, and not the IPC or power improvements. Overall, we find that the filter accuracy is high enough to indicate that we have identified a promising technique for smart search. Our results merit further examination, including quantifying the cycles and power saved. We leave these evaluations to future work.

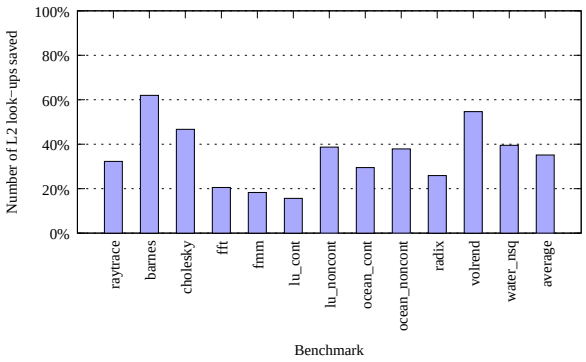

Figure 3. Percent of messages saved by our smart search per L2 access.

4.1 Messages Saved

The key metric for evaluating our smart search is how many messages it saves compared to the baseline search. A saved message is a block request sent to an L2 bankcluster by the baseline search, but filtered out by our search. The effect of fewer messages is decreased traffic on the L2 routing network, resulting in lower power and fewer contention cycles. Power is also saved because fewer bankclusters have to check their tag arrays.

Figure 3 shows the average messages saved per L2 access. The average savings across all benchmarks is 35%, and two are better than 50%.

4.2 Transistor Efficiency

We now evaluate our work in terms of its complexity effectiveness. To do so, we look at the number of messages saved on each L2 access per million transistors. We compare our filter with an idealized 6-bit partial tag array, much like the one used in prior work [7]. Each core has one tag array per bankset, just as we have one bloom filter per core per bankset. We do not model synchronization messages for the tag arrays, assuming that the arrays are perfectly synchronized with the corresponding bankset. Thus, the tag arrays in this test performs better than would a real implementation. As previously calculated, our filters are assumed to require 7.68 million transistors, and the partial tag arrays are assumed to require 72 million transistors.

Figure 4 shows the the results of this test. The number of messages saved for both structures is similar. The accuracy of the idealized partial tag array

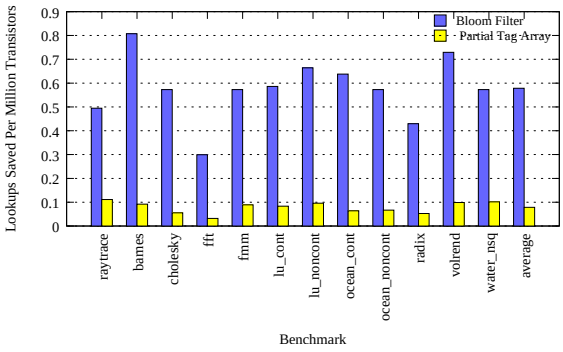

Figure 4. Average messages saved on each L2 access, per million transistors. We compare our filters with an idealized 6-bit partial tag array.

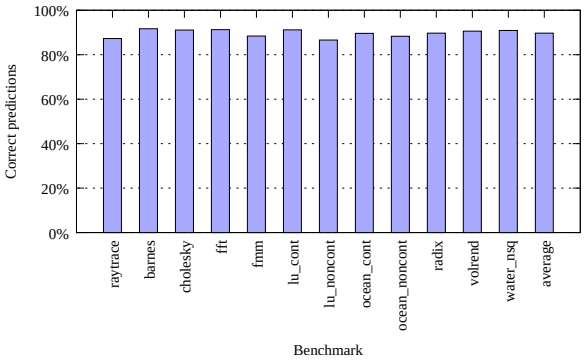

Figure 5. Overall filter accuracy.

is slightly higher—it saves on average 1.2 more messages per lookup. However, the Bloom filters, because of their much smaller size, make more efficient use of transistors. On average, they are able to save more than 6 times as many messages per million transistors.

4.3 Filter Accuracy

We now look at filter accuracy in finer detail, shown in Figure 5. This graph shows the number of true positives and true negatives over all filter lookups. As we can see, our filter accuracy is quite good overall.

The dominant cause of inaccuracy is false positives. This is due, in large part, to migration. When a cache line migrates, our filters learn the new location, but still remember the old location. We believe that the primary way to improve our filter accuracy will be to handle these migrations, removing line addresses from their old bankclusters.

To illustrate this point, we consider a modified version of D-NUCA that does not migrate blocks—it

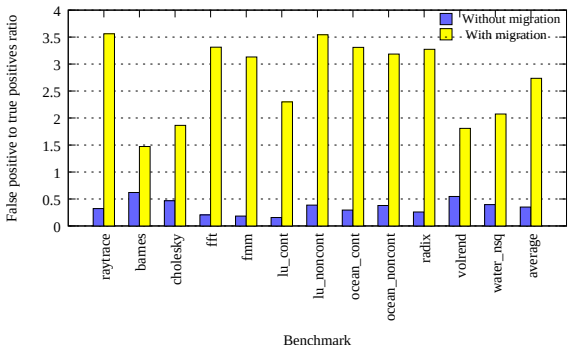

Figure 6. False positive to true positive ratio for D-NUCA caches with and without migration.

brings them into the cache according to D-NUCA polices, but does not move them thereafter. We do not claim that this cache policy is desirable, we use it only to isolate the effects of migration on our false positive rate.

The ratio of false positives to true positives with and without migration are shown in Figure 6. This ratio serves to isolate the percentage of false positives that result from migration, and we can see in this figure that the effect is significant. If we can, in future work, find a way to address migration, the accuracy of our filters will improve. Counting Bloom filters, which allow removal of set elements, may be of help, but it remains to be seen whether their higher storage requirements would be worth the benefit.

4.4 Per-L2 access Statistics

Finally, we examine the "per-L2 access" characteristics of the Bloom filters. We consider each L2 access (resulting from an L1 miss) as a whole. Thus, if *any* of a core's sixteen filters correctly predicts a hit in a particular bankcluster, we consider the whole access to have hit, regardless of the predictions from other filters. If there are no true positives, but one or more false positives, then the whole access is considered a false positive. If there are no positives returned, we classify the access as a true or false negative. Results are aggregated across all eight cores.

Figure 7 shows the Bloom filter accuracy for the SPLASH-2 benchmarks. This shows the percentage of L2 accesses for which our filters return the correct prediction (either a true positive or a true negative) on

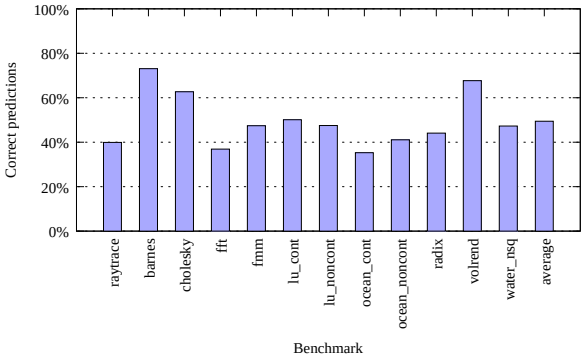

Figure 7. Filter accuracy per L2 access.

the first pass. We believe that the primary way to improve this accuracy will be to deal with migrations, as discussed in Section 4.3.

5 Conclusion

In this paper, we have explored the use of Bloom filters to create a smart search algorithm for CMPs with D-NUCA caches. Our results are very promising, showing that such filters can have very high accuracy, which results in a reduction of block requests. Our filters are complexity effective—they make efficient use of transistors, and do not make changes to the baseline coherency protocols.

Our initial explorations leave room for future work. Of most immediate importance, we will move on to quantify the cycle and power savings that result from our smart search. As shown in Section 4.3, if we can address the false positives that linger after block migration, very large improvements will likely result. Counting filters may help address this problem, but using them in this context is not straightforward.

Acknowledgements

We thank Liqun Cheng for helping us with Simics and benchmark suites. Virtutech AB for provided us with the Simics licenses necessary for our simulations. We also thank Wisconsin's Multifacet group, especially Bradford Beckmann and Mike Marty, for their help with GEMS.

References

[1] 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.

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

[3] B. Bloom. Space/time trade-offs in hash coding with allowable errors, July 1970.

[4] Z. Chishti, M. Powell, and T. Vijaykumar. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *Proceedings of ISCA-32*, June 2005.

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

[6] J. Huh, C. Kim, H. Shafi , L. Zhang, D. Burger, and S. Keckler. A NUCA Substrate for Flexible CMP Cache Sharing. In *Proceedings of ICS-19*, June 2005.

[7] C. Kim, D. Burger, and S. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Dominated On-Chip Caches. In *Proceedings of ASPLOS-X*, October 2002.

[8] P. Kongetira. A 32-Way Multithreaded SPARC Processor. In *Proceedings of Hot Chips 16*, 2004. (`http://www.hotchips.org/archives/`).

[9] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[10] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 2005.

[11] C. McNairy and R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *IEEE Micro*, 25(2), March/April 2005.

[12] S. Naffziger, B. Stackhouse, T. Grutkowski, D. Josephson, J. Desai, E. Alon, and M. Horowitz. The Implementation of a 2-Core Multi-Threaded Itanium Family Processor. *IEEE Journal of Solid-State Circuits*, 41(1), January 2006.

[13] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai. Bloom fi ltering cache misses for accurate data speculation and prefetching. In *Proceedings of Int'l Conference on Supercomputing (ICS)*, pages 189–198, June 2002.

[14] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. Technical Report TN-2001/2, Compaq Western Research Laboratory, August 2001.

[15] 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.

RegionTracker: Using Dual-Grain Tracking for Energy Efficient Cache Lookup

Jason Zebchuk and Andreas Moshovos
Department of Electrical and Computer Engineering
University of Toronto
{zebchuk, moshovos}@eecg.toronto.edu

Abstract

This work proposes energy efficient memory hierarchy lookup structures aimed primarily at relatively large, higher-level on-chip caches. The mechanisms proposed provide location information for a large fraction of cache references and eliminate the corresponding accesses to a larger, slower and less energy efficient tag array. A key contribution of this work is the concept of dual-grain tracking *where a two-level, two-grain approach is used to dynamically focus a set of few tracking resources on high-payoff memory areas. A coarse-grain tracking structure uses imprecise information to identify accesses to new regions of memory and then directs the allocation of a precise, fine-grain tracking structure. We propose* RegionTracker, *a simple implementation of dual-grain tracking which can be easily partitioned for optimizing its power and latency, and which does not use cascaded lookups or impose any restrictions on cache placement. We demonstrate that RegionTracker can significantly reduce lookup energy for various L2 caches. For example, we show that a RegionTracker that uses just 6.9% of the storage used by a conventional tag array and that can track just 128 8Kbyte regions, is able to reduce L2 lookup energy by 35% on the average for a 4MB L2 cache. We also demonstrate that RegionTracker can complement conventional, demand-driven tag set buffers and that it provides better energy savings.*

1 Introduction

This work proposes simple mechanisms to reduce cache lookup energy in higher level (L2 and L3) on-chip caches while targeting applications with relatively large memory footprints. A number of application, semiconductor technology and microarchitectural trends suggest that the contribution of tag lookup power to overall processor power will increase in the future. Tag energy will increase as higher level caches become larger and are accessed more frequently.

The size of higher level caches will increase as a result of application and semiconductor technology trends. Historically, application memory footprints and working sets for "typical" applications have grown and evolved. At the same time, the gap between processor and memory speeds has also grown. Larger on-chip caches help reduce the combined effects of these two trends.

Other semiconductor and microarchitecture trends will result in an increased number of accesses to higher level caches. Specifically, although semiconductor technology improvements have lead to smaller and faster transistors, corresponding increases in processing speeds have limited the amount of SRAM storage that can be accessed within a reasonable number of clock cycles. This combines with the low latency requirement of first level caches to limit the size of L1 caches. This limitation suggests that higher level caches will be accessed more often. Recent trends towards simultaneous and fine-grain multithreaded cores, as well as towards chip-multiprocessors have also resulted in larger high level, on-chip caches with increased traffic. Hardware and software prefetching further increase the demand for cache bandwidth. Finally, this increased cache traffic is becoming unbalanced as some requests, such as many coherence and prefetching requests, only access the tag arrays.

This works proposes *RegionTracker*, a complexity effective mechanism for increasing the energy efficiency of cache lookups. RegionTracker supplements the tag array, providing the same information for many lookups and thus eliminate man tag array accesses. RegionTracker uses two simple structures. The first structure tracks which coarse grain regions currently have blocks cached. It uses this information to detect the first access into newly touched regions. A second structure maintains fine-grain location information for individual blocks within regions (i.e., where the blocks are cached), but only for a small number of regions, as instructed by the coarse-grain tracking structure. As we explain in more detail in Section 2, typical application behavior is such that these two structures can be used to locate the blocks referenced by many cache accesses, exploiting the same behavior that makes small translation look-aside buffers effective.

As Section 3 explains, the implementation of RegionTracker is straightforward and imposes no additional restrictions on what can be cached simultaneously; it also requires no changes to existing cache implementations and avoids associative lookups

and updates. Imprecise information is used by the coarse-grain tracking structure, allowing a simple implementation at the price of capturing most relevant requests but not all. Key to the energy efficiency of RegionTracker is that each access precisely addresses a very small portion of the RegionTracker structures. Although RegionTracker can also potentially reduce lookup latency and improve performance, this work focuses on RegionTracker's ability to increase lookup energy efficiency.

RegionTracker is an example of mechanisms that rely on the concept of *dual-grain tracking*, or DGT for short. DGT mechanisms track block residency information at two levels of granularity so that a relatively small structure can efficiently satisfy many cache lookups.

This work makes the following contributions: (1) it introduces the concept of DGT; (2) it proposes *RegionTracker*, an energy efficient implementation of DGT, and demonstrates that practically sized RegionTrackers can reduce energy significantly (e.g., 35% of lookup energy for a 4Mbyte cache with a RegionTracker that requires just 6.9% of the resources required by a conventional tag array); (3) finally, it shows that RegionTracker provides better energy reduction than tag set buffers.

The rest of this paper is organized as follows: In Section 2 we introduce the DGT concept and briefly discuss how lookup energy can be reduced. Section 3 presents the RegionTracker implementation. We review related work in Section 4. In Section 5 we demonstrate RegionTracker's utility, and compare it to an existing technique for tag lookup energy reduction. Finally, in Section 6 we summarize this work.

For clarity, and without the loss of generality, we will use the term *tags* for conventional memory hierarchy lookup structures. The techniques we discuss, however, are applicable to other recently proposed lookup structures such as the centralized lookup arrays of the NuRapid memory hierarchy [10]. We also restrict our attention to level-two caches, however, the methods proposed should be directly applicable to even higher cache levels. Finally, all L2 caches used in this study are 8-way set-associative because through experimentation we found that our techniques are not noticeably sensitive to associativity. We also assume that the L2 uses 128-byte blocks (a commonly used size today).

2 Dual-Grain Tracking

RegionTracker (the implementation details are given in Section 3) achieves high energy efficiency via a two-level, dual-grain tracking (DGT) approach where the first level uses coarse-grain tracking and the second level uses fine-grain tracking for only a few, large memory regions. A *region* is a large continuous, aligned memory area of power of two size.

The coarse-grain level aims at detecting newly touched regions that have no blocks currently cached. It does so by detecting *first misses*. An access for block B within region R sent to cache C is a first miss if and only if no block within region R, including B, is currently cached within C. Once a first miss is detected, the fine-grain tracking level starts tracking the location of all blocks within the region. This is done by recording whether or not a block is cached, and if so, in which data way it is cached. This requires only a few bits per block, as opposed to a full tag. It is important to observe that when a first miss is detected, complete location information is also detected for the whole region since none of its blocks are currently cached. Thus, a single access uncovers information for many blocks, allowing the fine-grain tracking level to track all blocks as they accessed. This property eliminates the need for an initial search of the L2, and makes DGT effective despite a lack of substantial temporal locality in the L2 stream (temporal locality is typically absorbed by the L1 cache).

RegionTracker was designed primarily to exploit a behavior that is typical of many applications with large memory footprints. Specifically, although these applications access a very large set of regions over their lifetime, they typically operate on a few memory regions at any given time. The first time an application accesses a region, it incurs a first miss, giving RegionTracker an opportunity to track subsequent references within that region. Assuming that only a few regions are accessed at a time, RegionTracker should be able to track them all successfully using few resources. Much later, after many regions have been touched, a region may be accessed again. Given that the application has a large memory footprint, it is likely that all previously accessed blocks within the region have since been evicted from the cache as a result of capacity and, to a lesser extent, conflict misses. Accordingly, another first miss will occur and RegionTracker again has the opportunity to detect it and start tracking the region.

2.1 Reducing Lookup Energy with DGT

RegionTracker impacts energy and latency as summarized in Table 1. Prior to accessing the tag array for each cache lookup, RegionTracker is examined. Ideally, RegionTracker provides sufficient information to completely avoid the tag access. This assumes an in-series tag and data array organization for the L2 where the RegionTracker is accessed first and then the tags are accessed only if needed, and finally a single data way is accessed. This is the norm in commercial designs because it reduces power, e.g., [6]. We define a *hit* in RegionTracker as an access which indicates definitively

that the requested block is either not in the cache, or is located in a specific cache way. In the first case, only a single way of the tag array needs to be accessed in addition to replacement tracking information to determine and update the tag that will be replaced. In the latter case, there is no need to access the tag array at all. A lookup *miss* occurs when RegionTracker provides no precise information. In this case, more energy is used as the tag array has to be accessed after the RegionTracker.

Table 1. How RegionTracker (DGT column) impacts power and latency and which parts of the conventional L2 tag array have to be accessed.

DGT	*L2*	*Energy*	*Latency*	*L2 Tag Access*
miss	*miss*	increased	increased	all ways
hit	*miss*	decreased	decreased	single way + replacement information
miss	*hit*	increased	increased	all ways
hit	*hit*	decreased	decreased	status bits for one way as needed

3 RegionTracker Design and Application

The RegionTracker implementation of DGT studied in this work consist of two structures: (1) the *Cached Region Hash* or *CRH,* and (2) the *Cached Block Vector,* or *CBV*. The CRH is used to detect the first miss into a region and is identical to the CRH proposed in [24]. The CBV tracks the location of all the blocks within the few regions that are currently fine-grain tracked. The organization of both structures is shown in Figure 1. Both structures are indexed using parts of the incoming address. Without loss of generality, in this section we assume a 2Mbyte L2 cache, 42-bit physical addresses and 8Kbyte regions. The relevant parts of the incoming address are a unique *region number* (bits 41 through 13), and the *block offset* within the region (bits 12 through 7). The lower seven bits are the byte offset within a block and are not used by any RegionTracker structures. We assume only physical addresses are used with RegionTracker.

3.1 Cached Region Hash

The CRH keeps track of those regions that have blocks currently cached. We opt for a simple Bloom-like filter [5] which provides an imprecise representation of the set of regions that are currently cached. It consists of a table of counts which are incremented on each block allocation, and decremented on each eviction. The CRH is indexed using the region number of the block being allocated or evicted. In this work, the index is simply computed as a sufficient number of bits starting from the least significant bit in the region number (e.g., bits 13 through 22 for a 1K entry CRH); however, other indexing functions could be used. This simple, imprecise implementation allows us to use a very small, and hence energy and latency efficient structure to capture *most* first misses. Specifically, the CRH represents a *superset* of all regions that currently have blocks cached. The CRH can also easily be partitioned to further improve energy efficiency.

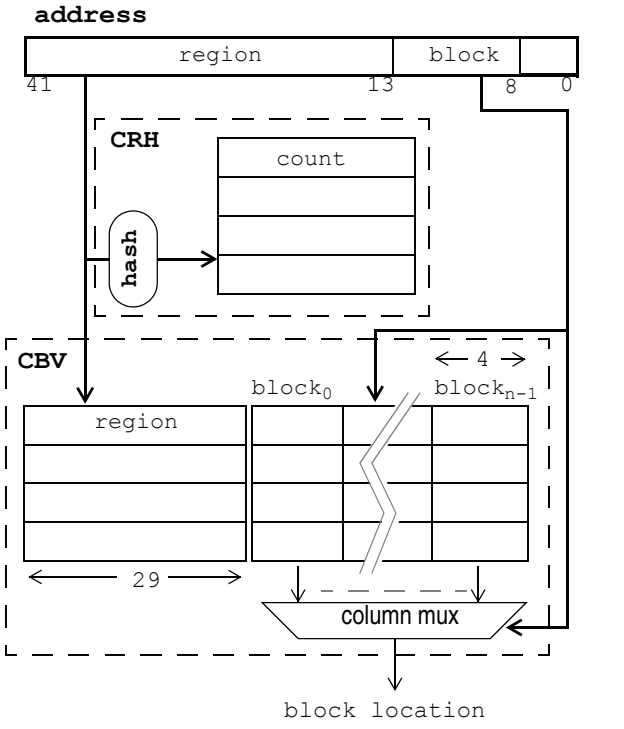

Figure 1: *(a) CRH and CBV organization for 8Kbyte regions, 42-bit physical addresses, 128-byte blocks and an 8-way set-associative cache.(b) An alternative fully-associative CBV implementation that results in lower power and latency.*

When a CRH counter is read, there are two possible outcomes. A counter value of zero indicates a first miss to a region. A non-zero counter value indicates that some portion of that region *may* be cached. The uncertainty results from potential aliasing of different regions onto the same CRH entry. When a first miss is detected, RegionTracker allocates a CBV entry for the region and starts fine-grain tracking of the location of all blocks in that region.

3.2 Cached Block Vector

The CBV is a table where each entry comprises a region tag and a set of information bits for each block within the region. For example, with 8Kbyte regions and 128-byte cache blocks, each CBV entry contains 64 block information fields. In the configurations considered in this work, the information fields encode whether or not the block is cached and where. For an 8-way set-associative cache, four bits are sufficient per block to encode the nine possible states: “not cached” or “cached in way N” where N ranges from 0 to 7. Depending on the cache organization, other information may also be stored in the information fields. For example, the CBV might store status or coherence information, or in a NuRapid memory hierarchy [10], the exact sub-array index can be

stored in the CBV. In the implementations considered in this work, status information is *not* stored in the CBV.

To access the CBV, the region number is compared with the region tags. If a matching entry is found, the information contained in the corresponding block field can be used to access the appropriate data array. The CBV is updated when blocks being tracked are allocated or evicted from the cache so that the CBV block information remains coherent. CBV entries are evicted when space is exhausted and a new entry has to be allocated following the detection of a first miss. Various replacement policies are possible, but this work uses an LRU replacement algorithm. While Figure 1 shows a fully-associative CBV, other organizations are possible as the CBV can be partitioned both vertically and horizontally to reduce energy and latency. Although we do not present the results here, for the programs we studied an 8-way set-associative CBV achieves coverage very close to a fully-associative CBV (within 2%). Further CBV optimizations are possible to reduce energy and latency, but the details are beyond the scope of this work. We note that only four bits need to be read out of the CBV array in Figure 1. Accordingly, only four bitlines are discharged during reads.

3.3 Energy and Storage Requirements

Since each RegionTracker access uses less energy than a conventional tag array access, using RegionTracker can reduce the total lookup energy. Section 5.3 describes the models used to calculate energy used by tag arrays and the RegionTracker structures. The various RegionTracker configurations presented in this paper use between 12% and 16% of the energy of the tag array for each access.

The low energy consumption results from two factors: (1) small size, and (2) small number of bits accessed. The RegionTracker configurations used in this work require between 0.8% and 18% of the storage of the L2 tag array. Appendix A provides a detailed discussion of the storage requirements. In addition to its small size, RegionTracker also benefits from only accessing a few bits on each access. For the configurations studied in this work, each CRH entry is only 10 bits, and only 4 bits in the CBV entry need to be read for each access. This compares to 184 tag bits that need to be read and compared for a 4MB, 8-way set-associative L2 cache with 128-byte blocks and 42-bit addresses.

3.4 RegionTracker Complexity

RegionTracker successfully reduces cache lookup energy with a minimal increase in hardware complexity. Since RegionTracker places no restrictions on how the cache operates, it does not introduce any new complexity in the implementation of the cache itself. Meanwhile, the RegionTracker structures are small and simple, and should lend themselves to a low complexity implementation. Finally, relatively little information needs to be communicated between the cache and RegionTracker; thus, adding RegionTracker to the cache access path should not significantly increase the overall complexity.

4 Related Work

Given the proportion of chip area devoted to caches, many contributions have been made to reducing cache power. However, most existing proposals target level one caches. The filter cache [16], consisting of a small cache placed in front of the L1 cache, can service a large fraction of L1 accesses, but misses to the filter cache incur an increased latency. A similar mechanism has been proposed for increasing L1 bandwidth [35], and [13] explored the idea of using these line-buffers in front of the L2 tag and data arrays to reduce power. Park *et al.,* [26] proposed a simple modification to this scheme which increased its effectiveness. These techniques exploit temporal and fine-grain spatial locality. As shown in Section 5.4, RegionTracker complements these techniques by filtering many L2 accesses that would only be caught by a larger TSB. Specifically, we show that a tiny (two entry) TSB combined with a RegionTracker outperforms a TSB with as many as 128 entries, and we also demonstrate that RegionTracker is more energy efficient.

A number of techniques have also been proposed for reducing the area and power of tag arrays. Decoupled sectored caches [32] and Caching Address Tags [34] are two techniques which reduce the tag array area by sharing tags amongst multiple cache blocks. The resulting structure has fewer tags than cache blocks. This exploits the same spatial locality as RegionTracker. However, since these techniques rely on a reduced number of tags, a single cache miss could require the invalidation of multiple cache blocks because their corresponding tag has been evicted. This incurs not only an initial latency penalty on such a miss, but also a possibly higher overall miss rate which can indirectly impact overall power and performance. RegionTracker does not affect L2 miss rate and, as we report in Section 5.5, with straightforward tuning it never hurts overall performance and hence power. Finally, implementing these techniques requires changing the L2 cache controller, something avoided by the simple RegionTracker implementation.

Other techniques which address tag array power include way prediction [12,14,28] and memoization [20], as well as techniques which attempt to optimize tag search energy using multi-stage tag lookup [8,9]. The former techniques, as well as [4] and [25] apply mostly to the L1 instruction cache, while the latter techniques were demonstrated for the L1 data cache. It is not clear if these techniques will scale well to larger L2 caches with higher

associativities. Additional work has incorporated compiler support for reducing cache power [1,2], and much work has been done which relies on cache partitioning, layout and circuit level techniques to realize energy reduction in caches, including [11, 17, 18, 19, 33].

Bloom filters similar to the CRH have been previously proposed for avoiding snoop-induced tag lookups [23] or snoop broadcasts [24], for L1 hit/miss prediction [27], load/store queue complexity reduction [30] and for miss prediction [22]. Whereas the Bloom filters previously proposed cannot track the locations of individual cache blocks, RegionTracker overcomes this short-coming by combining a bloom-like filter with a fine-grain tracking structure which can track and service most L2 requests.

5 Evaluation

This section is organized as follows: In Section 5.1 we describe our experimental methodology. In Section 5.2 we demonstrate the effectiveness of RegionTracker at servicing lookup requests. In Section 5.3 we report energy savings compared to a standalone, conventional tag array. In Section 5.4, we compare RegionTracker with tag set buffers. Finally, in Section 5.5 we summarize our findings about overall power and performance.

5.1 Methodology

We used Simplescalar v3.0 [7] to simulate the processor detailed in Table 2. Amongst several modifications, we modified the macros for the NOP instruction to not generate memory references (the NOP is a load to register zero and the hardware is supposed to ignore this load) and added support for modelling contention in the memory system. We compiled the SPEC CPU 2000 benchmarks for the Alpha 21264 architecture using HP's compilers and for the Digital Unix V4.0F using the SPEC suggested default flags for peak optimization. All benchmarks were run using a reference input data set. It was not possible to simulate a few benchmarks due to insufficient memory resources. Table 3 presents a list of the benchmarks as well as their memory footprints. Most of these footprints greatly exceed the L2 capacity, thus a reasonable RegionTracker cannot trivially track all blocks for an application.

To obtain reasonable simulation times, samples were taken for one billion committed instructions after skipping the first 100 billion committed instructions. For art and parser we only skipped 20 billion instructions prior to collecting measurements. We experimented with several other one billion instruction samples and with longer samples of up to 40 billion instructions and observed that results did not vary significantly for the different samples. A continuous instruction sample is important for our measurements as RegionTracker structures have to be kept coherent throughout execution. Unless otherwise noted we used timing simulation to measure the overall performance and power impact of RegionTracker. As shown in Table 2, the memory system comprises split level one data and instruction caches, a unified second level cache and a main memory. We studied L2 caches in the range of 2Mbytes to 16Mbytes. In the interest of space and clarity we use an *A/B* naming scheme for RegionTracker configurations where A is the number of CRH entries and B is the number of CBV entries. In all experiments we use an 8Kbyte region size. Section 5.3 describes the details of our power modeling methodology.

Table 2. Base processor configuration

Branch Predictor	**Fetch Unit**
16k GShare +16K bi-modal 16K selector 2 branches per cycle	Up to 6 instr. per cycle 64-entry Fetch Buffer Non-blocking I-Cache
Issue/Decode/Commit	**Scheduler**
any 6 instr./cycle	128-entry/64-entry LSQ
FU Latencies	**Main Memory**
same as MIPS R10000	Infinite, 300 cycles
L1D/L1I Geometry	**UL2 Geometry**
32KBytes, 2-way set-associative with 64-byte blocks	2Mbytes to 16Mbytes, 8-way set-associative with 128-byte blocks
L1D/L1I/L2 Latencies	**Cache Replacement**
3/3/16 cycles	LRU

Table 3. Total simulated memory bytes allocated per application during our simulation interval.

Benchmark	Memory Footprint	Benchmark	Memory Footprint
ammp	27M	***gcc***	133M
applu	186M	***gzip***	185M
apsi	196M	***lucas***	189M
art	89M	***mcf***	186M
bzip2	188M	***mesa***	10M
crafty	2M	***mgrid***	57M
eon	2M	***parser***	62M
equake	50M	***swim***	196M
facerec	17M	***twolf***	3M
fma3d	107M	***vortex***	70M
galgel	45M	***vpr***	51M
gap	193	***wupwise***	181M

5.2 Coverage with Practical RegionTrackers

We first report *coverage* results with RegionTracker. Coverage is the percentage of cache accesses for which RegionTracker provides precise location information, indicating either that the block is not cached, or cached in a specific cache way. We used functional simulation in order to evaluate a wide range of RegionTracker configurations. Figure 2 presents the results of timing simulations for the most promising RegionTracker configurations, showing the average coverage each configuration. Each curve represents a separate configuration, while the x-axis indicates the size of the L2 cache. As expected, coverage increases with larger RegionTrackers and for a fixed RegionTracker

configuration coverage decreases with L2 size. This result implies that on the average RegionTracker is well behaved and can be easily tuned to each cache configuration. Overall coverage varies from as high as 61% to as low as 15%. The results also demonstrate that if the size of the RegionTracker relative to the L2 cache is kept constant, then coverage remains roughly constant as the cache size increases. Consider, for example, a 2K/64 RegionTracker with a 4MB L2 cache, which achieves 45% coverage. Doubling the size of the cache and RegionTracker results in 46% coverage for the 4K/128 RegionTracker with the 8MB cache.

We next demonstrate that although RegionTracker coverage varies significantly across programs, it is high for most. Figure 3 presents the coverage for a 4K/128 RegionTracker for each benchmark with a 4MByte L2 cache. On average, this configuration achieves 55% coverage, with a minimum of 9.5% coverage for vortex and a maximum of 97.6% coverage for gzip. Those programs with low coverage generally access a large number of regions and require larger CBVs to obtain better coverage.

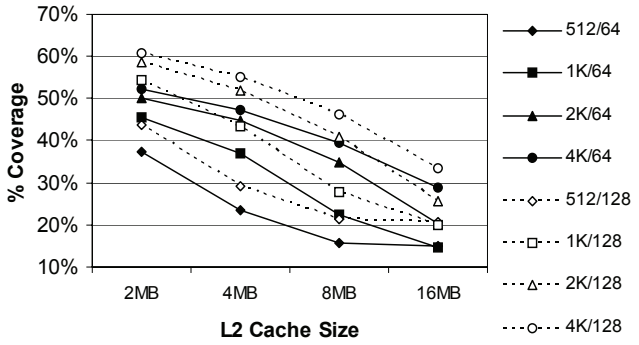

Figure 2: *Average coverage achieved by various RegionTracker configurations (different curves) for various L2 cache sizes (x-axis).*

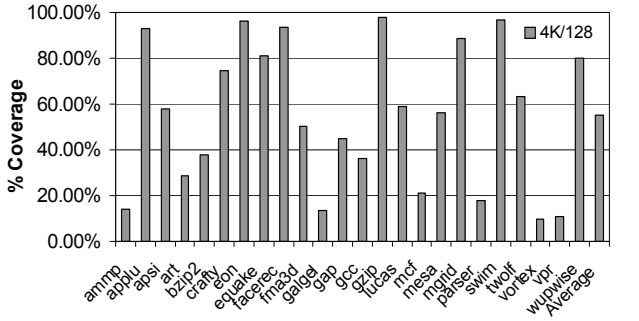

Figure 3: *Per benchmark coverage for a 4K/128 RegionTracker with a 4MB L2 cache.*

5.3 Energy Savings Compared to Conventional Tags

This section demonstrates that significant energy savings are possible with practical RegionTrackers for various L2 caches. We used CACTI 3.2 [29] to model the energy used by the L2 tag arrays and the RegionTracker structures. All structures were modeled in a 65nm technology. We modeled L2 caches from 2MB to 16MB in size. We selected a sub-bank size of 512KB[1], and each cache was divided into the appropriate number of sub-banks. The tag energy was computed as the sum of the tag decode, wordline, bitline, sense amp, and compare energy as reported by CACTI. For the RegionTracker configurations, the CRH was modeled as a direct mapped cache, but the contribution of the tag array energy was ignored as the CRH is an un-tagged structure. The CBV was modeled as a combination of a direct mapped cache and a fully associative CAM structure.

In modelling the various structures, we observed that the sense amps were contributing a significant portion of the energy, especially for the cache tag arrays. Previous work suggests that a power optimized sense amplifier would use about one third of the power used by the sense amplifier modeled by CACTI [21]. We thus scaled sense amp power accordingly. Note that this adjustment reduces the benefits of RegionTracker.

The energy savings were calculated based on the following assumptions:

- It is possible to access a single way in the tag array.
- On an L2 miss, only a single tag way is accessed to write the updated tag information.
- An L2 hit requires a full tag array lookup, and an L2 miss requires (1+1/A) tag array lookups, where A is the L2 associativity (i.e., initial access + single way access to update info).
- Each L2 hit caught by RegionTracker avoids a tag lookup.
- Each L2 miss caught by RegionTracker avoids the initial full tag set access and replaces it with a single tag way access (i.e., 2/A accesses are now required instead of (1+1/A))
- Each L2 access reads both the CRH and CBV.
- Each L2 miss causes a write to the CRH.
- Each L2 miss caught by RegionTracker also writes to the CBV.

These assumptions were used in combination with statistics from the simulations to calculate the lookup energy saved, as a percentage of the lookup energy consumed by a conventional L2 tag array.

Figure 4 reports average energy savings for different RegionTracker configurations (with a CRH with between 512 and 4k entries, and either 64 or 128 CBV entries), represented by different curves, and for L2 cache sizes from 2MB to 16MB, varied along the x-axis. The highest

[1] We evaluated various sub-bank sizes using CACTI and found that 512KB sub-banks minimized the energy-delay product.

savings of 44% is observed for the 4K/128 RegionTracker and the 2Mbyte L2 cache. This RegionTracker produces savings of 38%, 29% and 16% for the 4Mbyte, 8Mbyte and 16Mbyte caches respectively.

Since cache sub-bank size remains constant for all cache sizes, the tag energy remains almost constant; thus, the energy savings are reduced for larger caches according to the reduction in coverage shown in Figure 2. However, as the L2 cache size increases we can also increase the RegionTracker size to maintain the coverage and energy savings. It should be emphasized that as cache capacity increases, larger RegionTrackers become practical as their storage requirements become a smaller fraction of the L2 tag arrays.

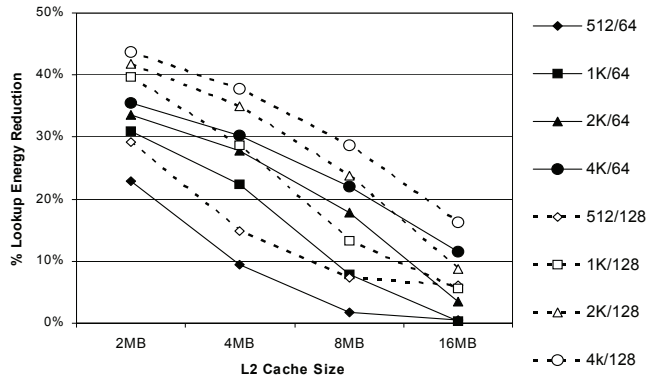

Figure 4: *Average energy savings expressed as fraction over the energy of a conventional L2 tag array. Shown are RegionTrackers with various CRH entry counts and a 64 or 128-entry CBVs. Results are shown for L2 caches of 2Mbytes to up to 16Mbytes (x-axis). Each curve corresponds to a different RegionTracker, labeled CRH/CBV. All RegionTrackers are 8-way set-associative.*

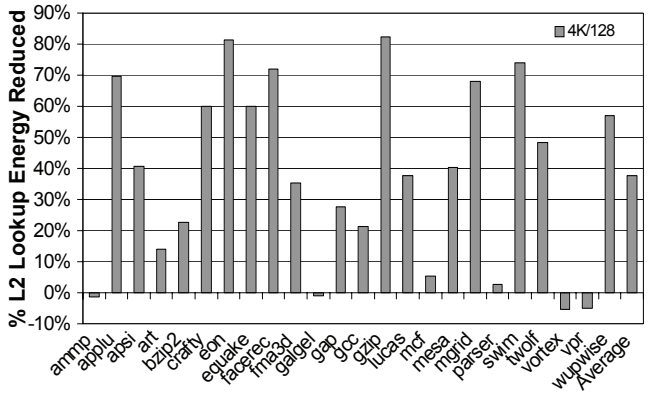

Figure 5: *Per program relative energy savings with a 4K/128 RegionTracker and a 4MB L2 cache, expressed as a fraction over the conventional L2 tag array power.*

Figure 5 indicates that while RegionTracker is robust and provides significant energy reductions for most programs, there are a few programs which exhibit an increase in energy consumption. This figure reports per program energy changes for a 4K/128 RegionTracker with a 4MB L2 cache. For a few programs, RegionTracker increases the lookup energy slightly, with the largest increase of 5% being observed for vortex and vpr. This compares with an average reduction of 38% and a maximum savings of 82% for gzip.

5.4 Comparing with Conventional Tag Set Caching

As we discussed in Section 4, a number of existing proposals for reducing L2 tag power rely either on efficient encoding or on keeping a small cache of recently accessed tags. In this section, we compare RegionTracker with tag set buffers (TSBs), or line buffers as they are often referred to, which have sizes less than or approximately equal to the size of the RegionTracker structure. We compare the two approaches using two metrics, coverage and energy savings. As we explain, TSB coverage rivals that of RegionTracker, however, energy savings does not.

A TSB is a small cache of recently accessed tag sets. For example, for an 8-way set-associative cache, each tag set entry will hold eight tags. Entries are allocated on demand as accesses probe the conventional tag array. Each access first probes the TSB, and if the set it maps to is found in the TSB, then there is no need to access the tags. If the set is not found in the TSB, then it is brought into the TSB after being read from the tag array.

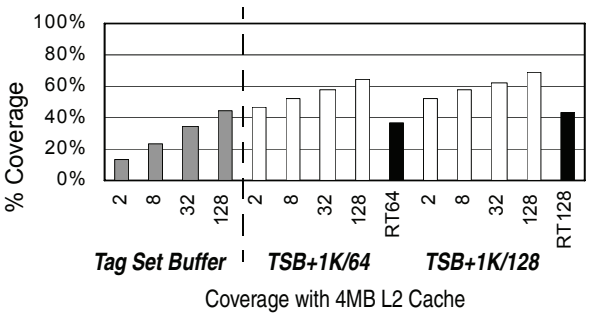

Figure 6: *Comparing Tag Set Buffers of various sizes to RegionTracker in terms of coverage for a 4MB cache. The four grey bars correspond to fully-associative tag buffers of two through 128 sets. The next four white bars are for a 1K/64 RegionTracker combined with Tag Set Buffers with the number of entries reported along the x-axis (two through 128). The coverage of the RegionTracker alone is shown by the next dark bar (RT64). Finally, we double the number of RegionTracker entries to 128.*

Figure 6 reports average coverage for various fully-associative TSBs (range of two to 128 entries), standalone 1K/64 and 1K/128 8-way set-associative RegionTrackers and combinations of the aforementioned TSBs and RegionTrackers. All results in Figure 6 are for a 4MB L2 cache. The grey bars report coverage for TSBs of the corresponding size (listed along the x-axis). The white bars report coverage for hybrid RegionTracker and

TSB organizations. The TSB entry count is listed along the x-axis. The first four are for the 1K/64 RegionTracker and the next four white bars are for the 1K/128 RegionTracker. Finally, the two black bars report coverage with just the 1K/64 (left) and the 1K/128 (right) RegionTrackers. Table 4 reports the storage requirements in bits of the TSBs and the two RegionTrackers as a fraction of the L2 tags. While it appears that a tag buffer can achieve coverage comparable to, or better than, a similarly sized RegionTracker, Section 5.4.1 shows that RegionTracker obtains better energy reduction.

Table 4. Comparing the storage requirements of tag buffers and RegionTrackers. Storage requirements are measured in bits and reported as a fraction over that of a conventional tag array for a 2Mbyte cache.

Tag Set Buffer	*Storage Requirements (bits)*
2	< 1%
4	< 1%
8	< 1%
16	< 1%
32	1.6%
64	3.3%
128	6.5%
CBV 64 + 1K CRH	6.6%
CBV 128 + 1K CRH	10.9%

5.4.1 Tag Set Buffer vs. RegionTracker trade-offs

A number of factors suggest that RegionTracker might offer a number of advantages over TSB. The designs of RegionTracker and TSBs are drasctically different. We used CACTI to model various TSBs, and while TSB may be conceptually simpler, each TSB access utilizes more energy than each RegionTracker access. Thus, a TSB with a given coverage will likely use more energy than a RegionTracker configuration that acheives similar coverage.

Figure 7 shows the average energy reduction for tag set buffers with 16, 32, 64 and 128 sets (different curves) for L2 caches of various sizes (x-axis). Only two configurations actually reduce energy on average, and the maximum reduction is only 3% for a 128 set TSB with a 16MB cache. Contrary to RegionTracker, TSB energy savings generally increase with cache size as the number of tag bits decreases for larger caches, and TSB coverage remains roughly constant as the cache size increases.

RegionTracker has an additional set of potential advantages over tag set buffers. These include the increased flexibility of RegionTracker implementations. Most accesses to RegionTracker involve a very small number of bits compared to the 180 or so tag bits involved in a TSB access. This leads to a flexibility in how the RegionTracker structures can be implemented, and where they are located. Also, RegionTracker would be relatively easy to port to novel or unconventional cache architectures such as NuRapid [10] or skewed

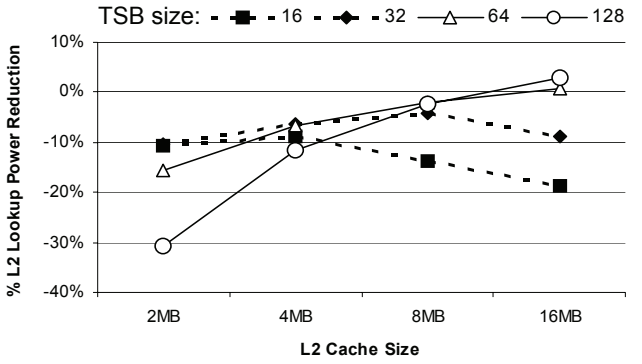

Figure 7: *L2 lookup power reduction (increase) with tag buffers of various sizes.*

associative caches [31]. An investigation of these issues is beyond the scope of this paper. The few results presented indicate that while both TSBs and RegionTracker can achieve comparable coverage, RegionTracker provides a much larger energy reduction than TSBs.

5.5 Performance and Overall Power

As mentioned above, RegionTracker affects L2 latency, and thus impacts both overall performance and power. We have measured the overall performance impact of RegionTracker configurations with 64 or 128 CBV entries and 512, 1K, 2K, and 4K CRH entries, assuming that it decreases L2 access latency by two cycles on a RegionTracker hit while it increases it by one cycle for a RegionTracker miss. These assumption were validated using an analytical latency model based on CACTI [29]. We studied caches of 2Mbytes and 4Mbytes. On average, overall performance increased less than 1% with RegionTracker. In the best case of twolf, performance increased by 2% with a 128/4k RegionTracker and a 2MB cache. Only a few benchmarks suffered from decreased performance, with the worst case being fma3d which had a slowdown of 0.02% with a 128/512 RegionTracker and a 4MB L2 cache. Correspondingly, overall processor power decreased slightly on average with the RegionTracker configurations we studied, although a few benchmarks saw increases of less than 0.1% for some configurations.

6 Summary

We proposed RegionTracker as an area, power and latency efficient implementation of memory hierarchy lookup structures aimed primarily at higher-level, relatively large, on-chip caches.

RegionTracker implements the concept of dual-grain tracking, using a simple Bloom-like filter (CRH) to track coarse-grain regions, combined with a small table of fine-grained region tracking entries (CBV). A key result was the demonstration that using a dual-grain tracking

approach provides significantly more potential than simple, demand-based allocation of fine-grained tracking resources. We demonstrated the utility of RegionTracker for reducing power and latency for L2 tag lookups. A 2k/128 RegionTracker saves 35% of the tag lookup power for a 4Mbyte L2 cache, while requiring less than 7% of the resources required for the conventional tag array. RegionTracker can be complemented by adding a tiny tag set buffer to achieve better coverage than either RegionTracker or tag set buffers can provide on their own. Other potential applications of RegionTracker include increased tag lookup bandwidth for aggressive prefetching, or increasing L1 tag port bandwidth and lookup latency, although the latter application would involve complex scheduling and latency issues.

Acknowledgements

The authors would like to thank Patrick Akl, Ioana Burcea, Davor Capalija, Elham Safi, and the anonymous reviewers for their valuable comments. Jason Zebchuk is supported in part by an NSERC Canada Graduate Scholarship (CGS-M). This work was supported by Semiconductor Research Corporation under contract #901.001, an NSERC Discovery Grant, a Canada Foundation for Innovation equipment grant, an Intel equipment donation and an Intel research grant.

7 References

[1] D. H. Albonesi. *Selective cache ways.* In the Proc. of the 32nd Annual International Symposium on Microarchitecture, Nov. 1999.

[2] R. Ashok, S. Chheda, C. A. Moritz, *Cool-Mem: Combining Statically Speculative Memory Accessing with Selective Address Translation for Energy Efficiency,* In the Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 2002

[3] B. Bateman, C. Freeman, J. Halbert, K. Hose, and E. Reese. *A 450Mhz 512KB Second-Level Cache with a 3.6GB/S Data Bandwidth.* In the Proc. of the IEEE International Solid-State Circuits Conference, 1998.

[4] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis. A*rchitectural and compiler support for energy reduction in the memory hierarchy of high performance processors.* In the Proc. of the International Symposium on Low Power Electronics and Design, Aug. 1998.

[5] B. Bloom. *Space/time trade-offs in hash coding with allowable errors.* Communications of ACM, pages 13(7):422-426, July 1970.

[6] W.J. Bowhill et al. *Circuit Implementation of a 300Mhz 64-bit Second Generation Alpha CPU.* Digital Journal vol. 7., 1995.

[7] D. Burger and T. Austin. The Simplescalar Tool Set v2.0, *Technical Report UW-CS-97-1342. Computer Sciences Department, University of Wisconsin-Madison*, June 1997.

[8] D. Brooks, V. Tiwari M. Martonosi. *Wattch: A Framework for Architectural-Level Power Analysis and Optimization*. In the Proc. of the 27th Annual International Symposium on Computer Architecture, June 2000.

[9] Y.-J. Chang, S.-J. Ruan and F. Lai, *Design and analysis of low-power cache using two-level filter scheme,* IEEE Transactions on VLSI, vol 11, no. 4, Aug. 2003.

[10] Z. Chishti, M. D. Powell and T. N. Vijaykumar, *Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures,* In the Proc. of the 36th Annual International Symposium on Microarchitecture, Dec. 2003.

[11] M. Huang, J. Renau, S.-M. Yoo and J. Torellas. *L1 Data Cache Decomposition for Energy Efficiency*. In the Proc. of the International Symposium on Low-Power Electronics and Design, Aug. 2001.

[12] K. Inoue, T. Ishihare and K. Murakami. *Way-Predicting Set-Associative Cache for High Performance and Low Energy Consumption,* In the Proc. of the International Symposium on Low-Power Electronic Design, August 1999.

[13] M.B. Kamble and K. Ghose. *Reducing Power in Superscalar Processors using subbanking, multiple line buffers and bit-line segmentation.* In the Proceedings of the International Symposium on Low Power Electronics and Design, 1999.

[14] R. E. Kessler, R. Joss, A. Lebeck, and M. D. Hill, *Inexpensive Implementations of Set-Associativity,* In the Proc. 16th Annual International Symposium on Computer Architecture, June 1989.

[15] C. Kim, D. Burger and S. W. Keckler, An Adaptive, *Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches,* In the Proc. of the 10. International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 2002.

[16] J. Kin, M. Gupta, and W. Mangione-Smith. *The Filter Cache: An Energy Efficient Memory Structure*. In the Proc. of the 30th International Symposium on Microarchitecture, pages 184-193, Nov. 1997.

[17] U. Ko, P. T. Balsara, and A. K. Nanda. *Energy Optimization of Multi-Level Processor Cache Architectures.* In the Proc. of the International Symposium on Lower Power Design, Aug. 1995.

[18] U. Ko, P. T. Balsara, and A. K. Nanda. *Energy Optimization of Multilevel Cache Architectures for RISC and CISC Processors.* In IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 6(2), Jun. 1998.

[19] H.S. Lee and G. S. Tyson. *Region-Based Caching: an energy-delay efficient memory architecture for embedded processors.* In the Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, Nov. 2000.

[20] A. Ma, M. Zhang, and K. Asanovic, *Way Memoization to Reduce Fetch Energy in Instruction Cache,* Workshop on Complexity-Effective Design, held in conjunction with the 28th Annual International Symposium on Computer Architecture, June 2001.

[21] M. Margala. *Low-power SRAM circuit design*. Memory Technology, Design and Testing, 1999. Records of the 1999 IEEE International Workshop on 9-10 Aug. 1999.

[22] G. Memik, G. Reinman, W. H. Mangione-Smith, *Just Say No: Benefits of Early Cache Miss Determination,* In the Proc. of the 9th International Symposium on High-Performance Computer Architecture, Feb. 2003.

[23] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. *JETTY: Filtering snoops for reduced energy consumption in SMP servers.* In the Proc. of the 7th International Symposium on High- Performance Computer Architecture, January 2001.

[24] A. Moshovos, *RegionScout: Exploiting Coarse-Grain Sharing in Snoop Coherence,* In the Proc. 32nd Annual International Symposium on Computer Architecture, June 2005.

[25] R. Panwar and D. Rennels. *Reducing the frequency of tag compares for low power I-Cache design.* In the Proceedings of the International Symposium on Low Power Electronics and Design, Aug. 1995.

[26] W.-H. Park, A. Moshovos, and B. Falsafi. *ReCast: Boosting Tag Line Buffer Coverage in Low-Power High-Level Caches 'for Free',* In the Proc. of the 2005 IEEE International Conference on Computer Design, Oct. 2005.27, Nov. 2000.

[27] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark and K. Lai, *Bloom filtering cache misses for accurate data speculation and prefetching,* In the Proc. of the 16th International Conference on Supercomputing, June 2002.

[28] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B Falsafi and K. Roy, *Reducing Set-Associative Cache Energy via Way-Prediction and*

Selective Direct-Mapping, In the Proc. of the 34th Annual Symposium on Microarchitecture, Dec. 2001.

[29] G. Reinman and N.P. Jouppi. *An Integrated Cache Timing and Power Model. Technical report,* COMPAQ Western Research Lab, 1999.

[30] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore and S. W. Keckler, *Scalable Hardware Memory Disambiguation for High-ILP Processors,* In the Proc. 36th Annual International Symposium on Microarchitecture, Nov. 2003.

[31] A. Seznec. *A Case for Two-Way Skewed-associative Caches.* In the Proc. of the 20th Annual International Symposium on Computer Architecture, May 1993.

[32] A. Seznec. *Decoupled sectored caches: Conciliating low tag implementation cost and low miss ratio*. In the Proc. of the 21st Annual International Symposium on Computer Architecture, June 1994.

[33] C. Su and A. Despain. *Cache Designs for Energy Efficiency.* In the Proceedings of the 28th Annual Hawaii International Conference on System Sciences, pages 306-315, 1995.

[34] H. Wang, T. Sun, and Q. Yang. *CAT – caching address tags: A technique for reducing area cost of on-chip caches.* In the Proc. of the 22nd Annual International Symposium on Computer Architecture, June 1995.

[35] K. M. Wilson, K. Olukotun, and M. Rosenblum. *Increasing cache port efficiency for dynamic superscalar microprocessors*. In the Proc. of the 23rd Annual International Symposium on Computer Architecture, May 1996.

Appendix A RegionTracker Relative Storage Requirements

Since RegionTracker acts to supplement a conventional tag array to reduce energy, it should require minimal overhead in terms of on-chip area. Tables 5 and 6 report the storage requirements (total bit count) of various CRH and CBV structures respectively, demonstrating that reasonably sized RegionTracker structures are much smaller than conventional tag arrays. The storage requirement of each structure is expressed as a fraction of the storage requirement of the tag array of a 2Mbyte L2 cache. This provides a first-order approximation of the area cost. The overall bit requirement is meaningful as an area estimate as it remains constant regardless of implementation details, such as partitioning into separate banks or sub-arrays.

Table 5 shows CRH requirements for entry counts of 512 through 4K. The size of each CRH entry depends on the cache configuration and region size. In general, each block in the cache could map to the same CRH entry as a result of aliasing. However, with the simple indexing function used in this work, only a fixed number of cache sets will can map to any CRH entry. Thus the number of bits required for each entry is only *N x lg(L2 Associativity x (Region size / block size))*, where *N* is the number of CRH entries.

Table 5. CRH storage requirements as a fraction of the bits required by the tag array of a 2Mbyte 8-way set-associative L2 cache with 128-byte blocks

CRH entries	*Storage*
512	1.2%
1K	2.4%
2K	4.8%
4K	9.6%

Table 6. Eight-way set-associative CBV storage requirements as a fraction of the bits required by the tag array of a 2Mbyte, 8-way set-associative L2 cache with 128-byte blocks. Ratios are shown for different CBV entry counts and region sizes. We assume 42-bit physical addresses and two status bits per tag entry (fractions will improve if additional status bits were used).

CBV Entries	*Region Size in Bytes*						
	512	**1K**	**2K**	**4K**	**8K**	**16K**	**32K**
16	<1%	<1%	<1%	<1%	**1.1%**	2.0%	3.9%
32	<1%	<1%	<1%	1.2%	**2.1%**	4.0%	7.9%
64	<1%	1.0%	1.4%	2.3%	**4.3%**	8.1%	15.8%
128	1.4%	1.8%	2.7%	4.6%	**8.4%**	16.1%	31.5%
256	2.6%	3.5%	5.4%	9.2%	**16.8%**	32.2%	62.8%
512	5.2%	7.0%	10.7%	18.3%	**33.5%**	64.2%	125.6%

The CBV storage requirements are primarily proportional to the number of CBV entries, the number of blocks within the region and the number of L2 ways. Larger regions or smaller blocks results in more block information fields in each CBV, and the L2 associativity determines the size of each field. The size of the region tags has only a small effect on the total CBV size. As shown in Table 6, a 128-entry CBV with 8Kbyte regions requires less than 9% of the bits needed by the conventional tag array. The percentages shown in Table 6 can also be used to estimate the relative cost of RegionTracker for larger caches since CBV requirements are not directly affected by cache size. For example, as the cache size doubles, the tag array approximately doubles in size as well, thus halving the relative storage requirements of the CBV. This work considers caches in the range of 2MB to 16MB, so while a 512 entry CBV requires 125% of the storage of a 2MB L2 tag array, it requires only 4.76% of the bits required by a conventional tag array for a 16MB cache.

Using Speculation Cost Predictability in Low-Power Cost-Aware Branch Prediction

Ehsan Atoofian, Amirali Baniasadi, and Farzad Khosrow-Khavar
Electrical and Computer Engineering Department
University of Victoria, Victoria, Canada
{eatoofian, amirali, fkhosrow}@ece.uvic.ca

Abstract

Branch prediction is essential in modern high-performance processors. Unfortunately, mispredictions are inevitable and cost energy.

We study branch mispredictions and show that there are a few high-cost mispredictions that account for most of the cost. We show that a low-cost branch instruction tends to remain low-cost in future reappearances. We exploit this predictability and introduce a simple yet efficient cost predictor that identifies low-cost branch instructions with an accuracy above 95%.

We use our findings and introduce cost-aware branch prediction. In cost-aware branch prediction we exploit a simple predictor for low-cost branch instructions and leave the full-blown predictor for branch instructions with higher costs. On average, we reduce branch predictor access frequency up to 50% with a maximum performance loss of 0.1%. This results in up to 22% branch predictor energy reduction.

1. INTRODUCTION

Speculation is an essential part of high-performance processors. Modern processors rely on branch outcome speculation to keep the pipeline full and to enhance performance. To achieve this, previous work has extensively studied branch instruction behavior and has suggested a variety of techniques to predict branch outcome. Despite steady progress in developing more accurate predictors, predictors still make (and are expected to continue making) mispredictions. While they are well-known mechanisms to recover from such mispredictions and to discard the mistakenly executed instructions, the associated energy cost is inevitable.

Reducing this cost has been the goal of several previous studies. Such efforts have focused on either exploiting more accurate branch predictors, effectively reducing the misprediction frequency (*e.g.*, [3,5]), or reducing the cost associated with the occurring mispredictions(*e.g.*,[2,11]). However, although some mispredictions waste more energy compared to others, such techniques favour uniformity, *i.e.*, they assume that all mispredictions are equally costly, performing the same actions per misprediction.

In this work we show that mispredictions are not equally important. In fact, a few (about 30%) of mispredictions account for the major part (more than half) of the misprediction cost.

We also investigate whether misspeculation cost can be estimated in advance and show that the speculation cost associated with a branch instruction can be predicted with high accuracy. In other words, low-cost branch instructions tend to remain low-cost. We identify low-cost branch instructions with an accuracy above 95% by using a simple cost-predictor.

We exploit misprediction cost variation and predictability and introduce cost-aware branch prediction. Conventional branch predictors access several power hungry structures to predict branch instruction outcome. This would be efficient (from the energy point of view) only if the energy overhead associated with such predictions does not exceed the energy cost of a misprediction. Accordingly, in cost-aware branch prediction we use a full-blow, energy hungry branch predictor for branch instructions with high mispredictions costs. For low-cost branch instructions, we use a small, simple and energy efficient branch predictor.

Speculation cost prediction can have other potential applications in areas such as checkpointing (*e.g.*, saving energy by selectively allocating checkpoints for high-cost branch instructions) or in multithreaded cores (*e.g.*, improving performance by giving higher priority to fetching instructions from threads with less high-cost branches). However, due to space limitations, in this work we do not consider these applications.

We make the following contributions:

- We show that speculation cost a) is not uniformly distributed among mispredictions and b) is predictable using a low-overhead, simple and effective cost predictor.
- We introduce *cost-aware branch prediction* to reduce power dissipation in the branch predictor. We do so by avoiding using a full-blown predictor for low-cost branch instructions.

Branch mispredictions result in wasted energy (consumed by flushed instructions) and time (cycles spent on executing mispredicted instructions). In this work we use the number of mispredicted (and therefore flushed) instructions to estimate cost. Our study shows that similar conclusions can be reached if the number of cycles spent on mispredicted instructions is used to estimate cost.

The rest of the paper is organized as follows. In Section 2 we study speculation cost. In Section 3 we introduce and study cost prediction. In Section 4 we present cost-aware branch prediction. In Section 5 we present our methodology and evaluate cost-aware branch prediction. In Section 6 we review related work, Finally, in Section 7 we offer concluding remarks.

2. SPECULATION COST

A previous study shows that a high percentage of fetched instructions are flushed due to mispredictions [2]. The flushed instructions are referred to as *extra work*. Extra work is not equally distributed among all mispredictions. While some mispredictions result in flushing a large number of instructions, others do not flush that many.

In Figure 1(a) we report misprediction cost distribution for the subset of SPEC CPU 2000 benchmarks studied here and the base processor presented in Table 1 (see Section 5 for the methodology). We classify mispredictions based on their cost, *i.e.*, how many instructions they flush. For example, the 1_7 category represents the percentage of mispredictions flushing less than 8 instructions.

Individual program behavior varies. For most programs, mispredictions with a cost of 32 or higher are relatively frequent (*e.g., 176.gcc*) or the majority (*e.g., 175.vpr)*. In other programs most mispredictions have relatively small cost (*e.g., 253.perlbmk*). On the average, about 67% of the mispredictions have a cost of less than 32.

Figure 1(b) shows the contribution of each misprediction category to the overall cost. Whereas in Figure 1(a) each misprediction counted the same independently of its cost, here misprediction is weighted by its actual cost. For most applications, more than half of the overall cost is the result of mispredictions with a cost higher than 32 instructions. On the average, about 53% of the overall cost is the result of such mispredictions.

We conclude from Figure 1 that mispredictions are not equally costly: less frequent high-cost mispredictions account for a large share of total cost, while more frequent low-cost mispredictions, account for a lower share.

3. COST PREDICTION

Our study shows that a mispredicted branch flushing low number of instructions is likely to flush low number of instructions next time mispredicted too. We exploit this

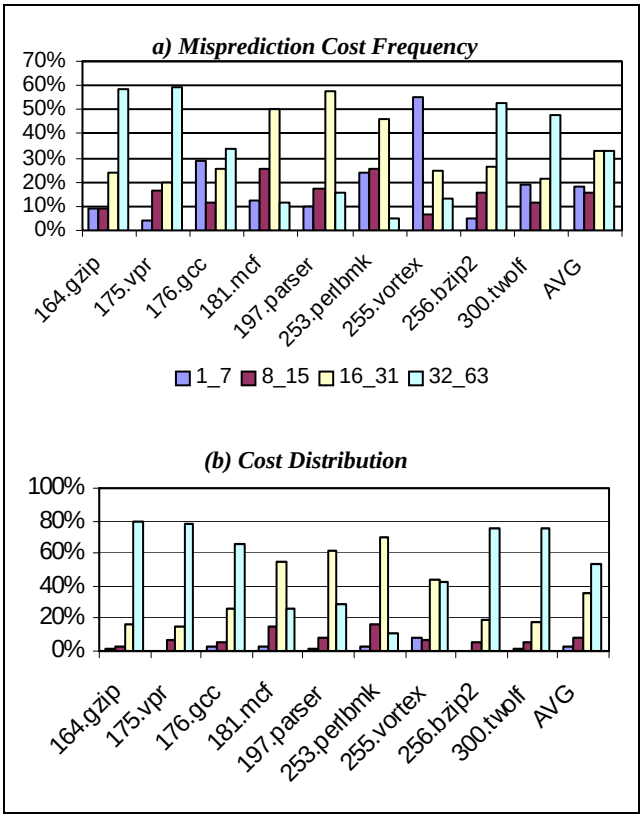

Figure 1: Branch mispredictions cost class: a) frequency b) share of total misprediction cost.

behavior and suggest using branch cost history to estimate possible future speculation cost.

To speculate misprediction cost, we use a table of saturating counters accessed by xoring the instruction PC and the global branch instruction history. The saturating counter records how frequently the branch has been low-cost in the past.

To differentiate between a low-cost misprediction and other mispredictions we use a pre-decided low-cost threshold (LCT).

Upon a low-cost misprediction (*i.e.*, a misprediction with a cost less than LCT), we increment the corresponding table counter. On a high-cost misprediction we reset the counter. At fetch we probe this table and mark a branch as low-cost if the corresponding counter is saturated to the maximum value.

In Figure 2 we report how accurately low-cost branch instructions can be predicted by various cost predictors. We study how changing design parameters impacts predictor accuracy for a highly accurate and low-overhead cost predictor. We vary the LCT in part (a), the counter width in part (b) and the number of entries in part (c).

In Figure 2(a) we report accuracy, *i.e.*, how often a branch instruction predicted to be low-cost turns out to be one, for different LCTs and for a cost predictor equipped

with a 128-entry table of 3-bit counters. In general, accuracy improves as LCT increases. Average accuracy varies between 96.7% and 98.8% for different LCTs. In Figure 2(b) we report accuracy for different counter sizes. We assume a 128-entry table and an LCT of 16. Accuracy is higher for larger counters. Average accuracy varies between 97% and 98.6% for different counter sizes. In Figure 2(c) we report accuracy for different table sizes. We assume 3-bit counters and an LCT of 16. In general, larger tables show better accuracy. Average accuracy varies between 97.6% and 98.4% for different table sizes.

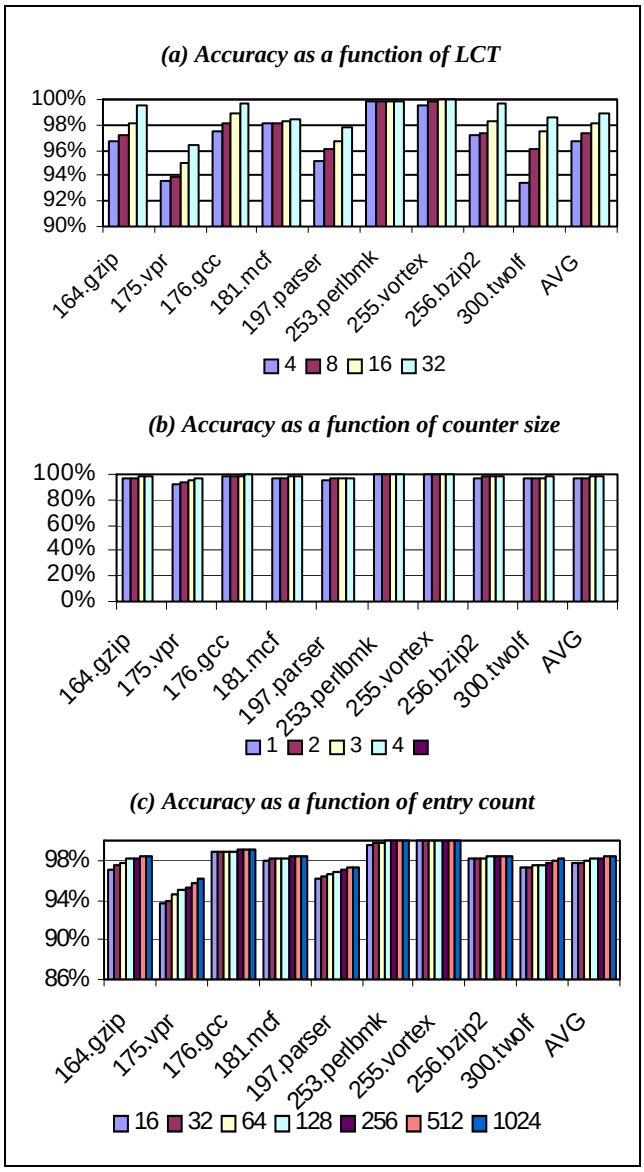

Figure 2: Prediction accuracy: a) for a cost predictor using 128-entry of 3-bit counters for different LCTs (i.e., 4, 8, 16, 32) b) for a 128-entry cost predictor for saturating counters with different sizes (1, 2, 3, and 4-bit counters). LCT is 16. c) for different predictor sizes (i.e., 16, 32, 64, 128, 256, 512, 1024 entries). LCT is 16 and each entry has a 3-bit counter.

An accurate evaluation of the cost predictors studied here would require measuring predictor coverage in addition to accuracy. Therefore, in Figure 3 we report coverage (*i.e.*, the percentage of low-cost branch instructions identified) for various cost predictors. Similar to Figure 2, we vary the LCT in part (a), the counter width in part (b) and the number of entries in part (c).

In Figure 3(a) we report coverage for different LCTs and for a cost predictor equipped with a 128-entry table of 3-bit counters. In general, coverage improves as LCT increases. Average coverage varies between 74.7% and 95.2% for different LCTs. In Figure 3(b) we report coverage for different counter sizes. We assume a 128-entry table and an LCT of 16. Coverage is lower for larger counters. Average coverage varies between 76.8% and 96.1% for different counter sizes. In Figure 3(c) we report coverage for different table sizes. We assume 3-bit counters and an LCT of 16. In general, larger tables show better coverage. Average coverage varies between 83.1% and 86.8% for different table sizes.

4. COST-AWARE BRANCH PREDICTION

Developing alternative power-aware branch predictors for high-performance processors is important due to two reasons.

First, conventional high-performance designs access the predictor aggressively and frequently. This requires using multi-ported structures and can result in high temperatures (possibly resulting in faults) and higher leakage power.

Second, the branch predictor is an energy hungry unit and consumes a considerable share of the processor's energy budget [7].

In this study we use the combined predictor [3]. This predictor accurately captures the behavior of many branches.

Cost-aware branch prediction (CAP) exploits variations in cost and predictability in order to reduce predictor power. This is achieved by selectively using a simpler and smaller predictor for branch instructions which are both well behaved and low-cost.

Accordingly, CAP uses a small filter, referred to as the CAP-filter, to identify low-cost, high-confidence branch instructions. This filter is an extension of the cost predictor introduced earlier and includes information regarding cost and confidence. We use this filter to guess the branch outcome for low-cost branches.

Figure 4 shows the organization of the CAP-filter structure. In this work we use a 128-entry direct-mapped CAP-filter after testing several alternatives. We have observed that storing 128 entries provides adequate information with affordable overhead. While smaller filters tend to miss energy saving opportunities for some applications, larger filters results in unjustifiable overhead.

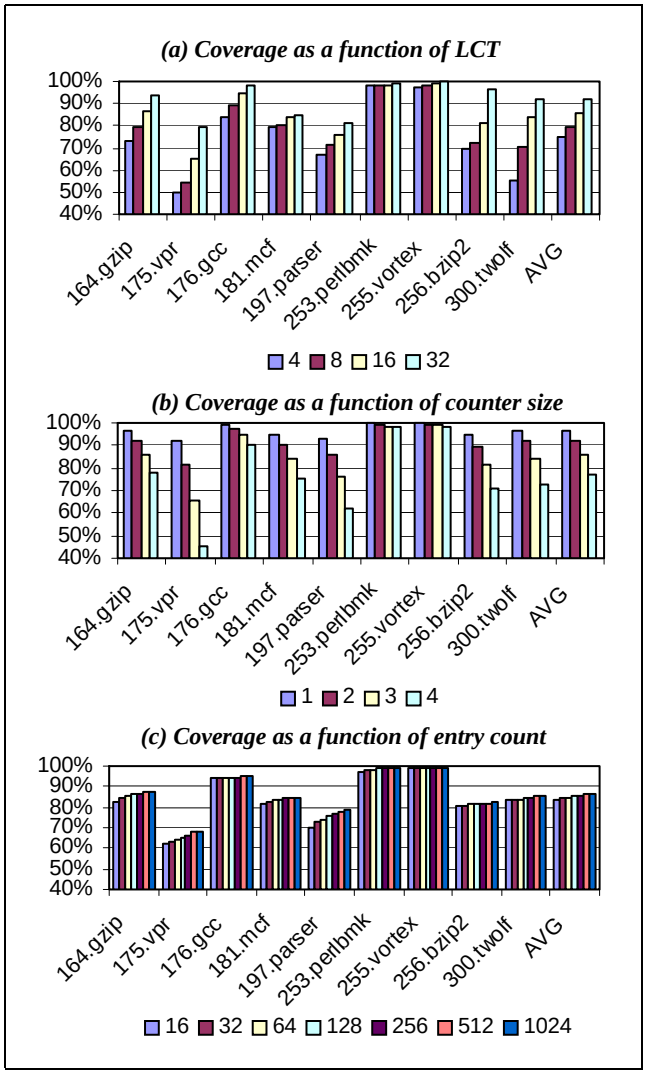

Figure 3: Prediction coverage: a) for a cost predictor using 128-entry of 3-bit counters for different LCTs (i.e., 4, 8, 16, 32) b) for a 128-entry cost predictor for saturating counters with different sizes (1, 2, 3, and 4-bit counters). LCT is 16. c) for different predictor sizes (i.e., 16, 32, 64, 128, 256, 512, 1024 entries). LCT is 16 and each entry has a 3-bit counter.

Every CAP-filter entry has 3 fields: tag, a 3-bit counter to record cost, and valid. High confidence branch instructions are stored in the filter. Branches are known to be high confidence if the corresponding saturating counters in the bimodal and the gshare predictors are saturated to strong taken. We focus on taken branches as our study shows that they account for a larger share of high confidence branch instructions. The 3-bit saturating counter records how frequently the branch has been low-cost in the past. We increment the corresponding table counter for low-cost branches and reset the counter for high-cost (flushing more than 16 instructions) ones.

The last field, *"valid"*, is initially (*i.e,* when the branch is stored in the filter) set to zero. An accurately predicted branch sets the "valid" bit to one. If the "valid" bit associated with the branch is 0, the original predictor is accessed. If a branch is mispredicted we reset the valid bit. We only trust the CAP-filter if the valid bit is set.

It is easier to reasons about CAP if we assume that for each dynamic branch instruction, we access the CAP-filter prior to the original predictor (we may access them in parallel to avoid increasing latency). Accessing CAP-filter allows us to decide if the branch is a low-cost branch instruction. If so, we save power by not accessing the branch predictor. If the branch is not present in the CAP-filter we access the original predictor. A branch is removed from CAP-filter either because of limited space or because it is mispredicted.

Provided that sufficient number of low-cost branches are accurately identified, CAP can potentially reduce branch prediction energy consumption. However, it introduces energy overhead and can, in principle, increase overall power dissipation if the necessary behavior is not there. We take into account this overhead in our study and show that for the programs we studied CAP is robust.

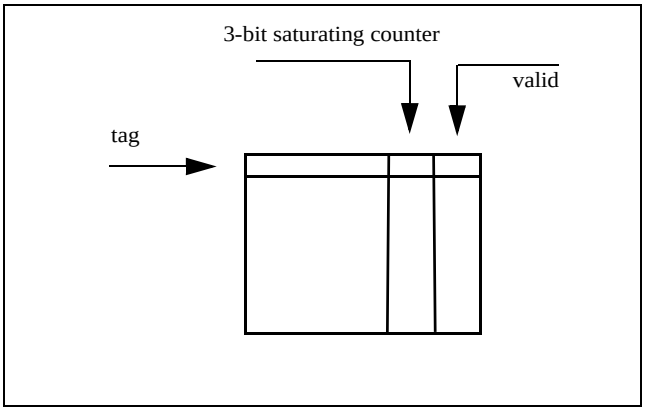

Figure 4: CAP-filter

5. METHODOLOGY AND RESULTS

To evaluate CAP we report performance, predictor access frequency and energy reduction.

We use a 128-entry CAP-filter. Using Wattch [14] we measured the power overhead of the CAP-filter. The energy consumed by accessing the 128-entry predictors is less than 2% of that consumed by the 16k-entry combined predictor.As for the timing overhead, even under the worst case timing scenario, it should be possible to access the cost predictor and the CAP-filter in parallel to the branch predictor. By using CACTI [15] we have estimated that under this assumption we can abort a 16k-entry predictor access after the decode. This saves the energy used by the wordlines, bitlines, and sense amplifiers. This would maintain front-end latency but will cut our savings by 40%. Therefore if we must avoid increasing front-end latency at all costs, we

can still reduce power by terminating some accesses early enough.

We used integer programs from SPEC'2k compiled for the MIPS-like architecture used by the Simplescalar v3.0 simulation tool set [6]. We run each simulation starting at the "early single" SimPoint [13] and run for 500M instructions.We used GNU's *gcc* compiler. The main architectural parameters are shown in Table 1

.

Table 1: Base configuration details.

Fetch/Issue/ Decode/Commit	any 4 instructions
Scheduler	128 entries, RUU-like
Load/Store Queue	64 entries, 4 loads/stores per cycle
L1 - I/D Caches	64K, 4-way SA, 32B blks, 3C hit latency
Unified L2	256K, 4-way SA, 64B blks, 16C hit latency
Branch Predictor	16K GShare, bi-modal, selector
FU. Latency	MIPS R10000
Memory	Infinite,100 cycle

5.1. Performance, Access frequency and Energy

In Figure 5(a) we report performance cost for CAP. Performance slowdown is below 0.1% across all programs for different LCTs. Average performance loss is below 0.04% for the applications studied here. This is a significant result as it shows that CAP maintains predictor accuracy and therefore performance at a competitive level.

In Figure 5(b) we report branch predictor access reduction. We report access reduction as it provides an implementation and technology independent measurement of potential energy savings. Access reduction reaches a maximum of 50% (average:~27% for different LCTs).

In Figure 5(c) we report branch predictor energy reduction as measured by wattch [14]. Energy reduction reaches a maximum of 22% (average:~9% for different LCTs).

Our study shows that by using CAP we also reduce processor overall energy consumption up to a maximum of 1.7% (average:0.8%). Overall energy savings may not appear considerable in absolute terms. However, when weighted against the insignificant performance cost, the savings are worthwhile and higher than that achieveable using techniques such as dynamic voltage scaling.

.

6. RELATED WORK

Skadron *et al*. introduced a new taxonomy for branch misprediction [12]. They show that wrong history mispredictions are the main source of mispredictions. They introduced alloyed predictors mixing both local and global history to index the pattern history table (PHT).

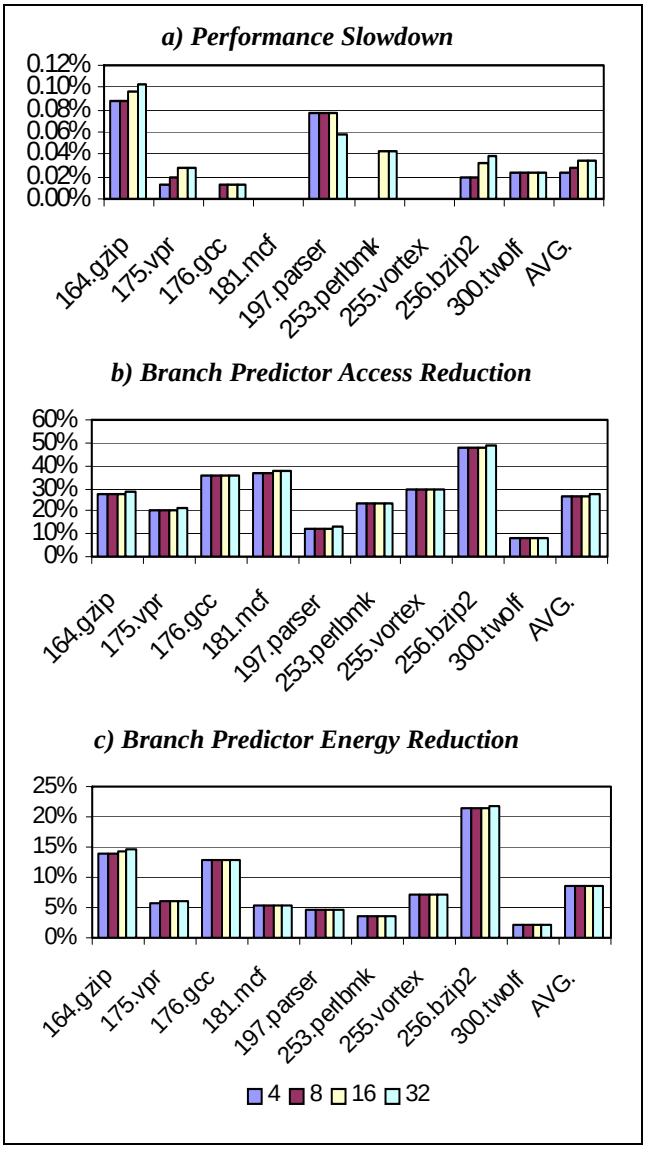

Figure 5: CAP: a) Performance slowdown b) Branch predictor access reduction c) Branch predictor energy reduction. Bars from left to right report for LCTs of 4, 8, 16 and 32 respectively.

Parikh *et al.* explored the effects of branch predictors on processor power dissipation. They also introduced banking and prediction probe detector (PPD) to reduce predictor or BTB energy consumption. Banking reduces the active portion of the predictor. PPD identifies when a cache line has no branches so that a lookup in the predictor buffer/BTB can be avoided [7].

Baniasadi and Moshovos introduced Branch Predictor Prediction (BPP) [10] and Selective Predictor Access (SEPAS) [9] to reduce branch predictor energy consumption. BPP stores information regarding the sub-predictors accessed by the most recent branch instructions executed and avoids accessing all three underlying structures.

SEPAS selectively accesses a small filter to avoid unnecessary lookups or updates to the branch predictor.

Huang *et al*. used profiling to reduce power dissipation in the branch predictors [8]. They disable tables that do not improve accuracy and reduce BTB size for applications with low number of static branches.

Our work is different from all above studies as it relies on speculating misprediction cost.

Several previous studies have suggested exploiting filters to reduce either branch predictor complexity and delay or table interference and destructive aliasing [5,16,4]. Our work's main difference with these studies is that we look at the power aspect of exploiting such filters.

Chang *et. al* [5], suggested identifying easily predictable branches and inhibiting the pattern history table for these branches to reduce table interference.

Eden *et. al* [4] introduced YAGS to reduce aliasing in the pattern history table.

Jimenez *et. al* [16] suggested using an overriding branch predictor which provides two predictions, one faster and one slower and less accurate consecutively, to reduce delay.

Using saturating counters as branch confidence estimators was first suggested by Smith [1]. We use saturating counters to estimate branch misprediction cost.

Manne *et. al*, [2] suggested the "Both Strong" estimation method which marks a branch as high confidence if the saturating counters for both gshare and bimodal are saturated and have the same predicted direction. We also use "Both Strong" to identify high-confidence branch instructions. However, we focus on a subset (*i.e.*, low-cost branches) of high-confidence branch instructions in CAP.

7. CONCLUSION

We studied misprediction cost from the energy point of view. We showed that misprediction cost is a) not distributed uniformly across branch mispredictions and b) predictable using a small predictor. We also investigated how changes in design parameters for the cost predictor impacts accuracy.

We used our findings and introduced cost-aware branch prediction. In cost-aware branch prediction we exploit variations in cost and predictability and reduce predictor power dissipation. We achieve this by selectively using a simpler and smaller predictor for branch instructions which are both well behaved and low-cost. By using the cost-aware branch predictor we reduced power while maintaining performance.

8. ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada, Discovery Grants Program and Canada Foundation for Innovation, New Opportunities Fund.

References

[1]E. Jacobsen, E. Rotenberg, J. Smith: Assigning Confidence to Conditional Branch Predictions. MICRO 1996: 142-152

[2]S. Manne, A. Klauser, D. Grunwald: Pipeline Gating: Speculation Control for Energy Reduction. ISCA 1998: 132-141S.

[3]S. McFarling. Combining Branch Predictors. Tech. Note TN-36,DECWRL, Jun. 1993.

[4]A.N.Eden and T.Mudge, The YAGS Branch Prediction Scheme, *In Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, Nov.1998.

[5]P.Chang, M.Evers and Y.N. Patt, Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference. PACT. 1996.

[6]D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report Computer Sciences Tech. Report #1342, University of Wisconsin-Madison, June 1997.

[7]D. Parikh, K. Skadron, Y. Zhang, M. Barcella, M. Stan: Power Issues Related to Branch Prediction. HPCA 2002.

[8]M.C Huang, D. Chaver, L. Pinuel, M. Prieto, F. Tirado. Customizing The Branch Predictor To Reduce Complexity And Energy Consumption, In IEEE micro. Sep. 2003

[9]A. Baniasadi, A. Moshovos: SEPAS: a highly accurate energy-efficient branch predictor. ISLPED 2004: 38-43

[10]A. Baniasadi, A. Moshovos: Branch Predictor Prediction: A Power-Aware Branch Predictor for High-Performance Processors. ICCD 2002: 458-461

[11]A. Gandhi, H. Akkary, S. Srinivasan: Reducing Branch Misprediction Penalty via Selective Branch Recovery. HPCA 2004: 254-264

[12]K. Skadron, M. Martonosi, D. Clark: A Taxonomy of Branch Mispredictions, and Alloyed Prediction as a Robust Solution to Wrong-History Mispredictions. PACT 2000: 199-206

[13]T. Sherwood, E. Perelman, G. Hamerly, B. Calder. Automatically Characterizing Large Scale Program Behavior, ASPLOS 2002. San Jose, California.

[14]D. Brooks, V. Tiwari, M. Martonosi: Wattch: a framework for architectural-level power analysis and optimizations. ISCA 2000: 83-94

[15]S. Wilton and N. Jouppi. "An Enhanced Access and Cycle Time Model for On-chip Caches." In WRL Research Report 93/5, DEC Western Research Laboratory, 1994.

[16] D.A.Jimenez, S.W.Keckler, C. Lin, The Impact of Delay on the Design of Branch Predictors. *In Proceedings of International Symposium on Microarchitecture*, December 2000.