UNIVERSITY OF CALIFORNIA

Los Angeles

# Factored Multi-core Architectures

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

**Anahita Shayesteh**

2006

The dissertation of Anahita Shayesteh is approved.

William Kaiser

Jens Palsberg

Yuval Tamir

Glenn Reinman, Committee Chair

University of California, Los Angeles

2006

*To my parents*

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ACKNOWLEDGMENTS

I would like to extend my gratitude and appreciation to the many people in my academic and personal life who made this dissertation possible. First and foremost, I would like to thank my research advisor, Professor Glenn Reinman for his support, encouragement, and guidance. I could not have asked for a better advisor. It has been a rare privilege to work with him these past four years. His deep knowledge in this field, his enthusiasm for research, and his trust and patience have been the guiding forces behind my research.

I would also like to thank professors Tim Sherwood, Suleyman Sair, Gokhan Memik and Norm Jouppi for providing their invaluable guidance. I am grateful for the opportunity to collaborate with them. I would also like to acknowledge and thank professors Yval Tamir, Bill Mangione-Smith, Jens Palsberg and William Kaiser who served on my committee and provided many insightful comments.

It has been a pleasure working with so many talented and dedicated people at UCLA. I wish to express my special thanks to Eren Kursun and Yongxiang Lui with whom I collaborated on this research. I would also like to extend my appreciation to Thomas Yeh, Adam Kaplan, and Kanit Therdsteerasukdi and acknowledge their help and support. I want to express my thanks also to Osvaldo Colavin for his support and guidance during my summer internship at STMicroelectronics.

Many individuals have made my graduate school experience a memorable one. I am thankful to all of them, especially Shiva Navab, Aida Varzaghani, Soheil Ghiasi, Eli Bozorgzadeh, Mozhgan Mansuri, and Parvaneh Mohaddes who provided their unwavering support and friendship. Also I would like to thank Dr. Mohammad Navab and Dr. Robert Newman for their non-stop support and unexpected kindness towards

me. I would also like to express my special thanks to our department's graduate stduent advisor, Verra Morgan who is always there for students. I am very grateful for her help and support.

My special love and gratitude goes out to my family for their unconditional support, understanding, and sacrifice without which I could not have accomplished this effort. I offer special thanks to my sisters Kati and Elnaz who have been my closest friends since youth. We share a special bond as only siblings can. Last, and most especially, I want to thank my parents Teyomour and Sedigheh who gave me the gift of love and served as examples of what hard work, love and wonderment can bring to one's life. I dedicate this thesis to both of them.

# VITA

| | |
|---|---|
| 1977 | Born, Mashad, Iran. |
| 1988–1995 | NODET (National Organization for Development of Exceptional Talents) Middle/High School. |
| 1995–1999 | B.S. (Electrical Engineering), Sharif University of Technology, Tehran, Iran. |
| 2000–2002 | M.S. (Computer Science), UCLA, Los Angeles, California. |
| 2001 | Teaching Assistant, Computer Science Department, UCLA. CS152A Digital Design Laboratory |
| 2004, 2005 | Teaching Assistant, Computer Science Department, UCLA. CS151B Computer Architecture |
| 2005 | Summer Intern, Microprocessor Research Group, ST Microelectronics, San Diego, California. |
| 2000–present | Graduate Research Assistant, Computer Science Department, UCLA. |

# PUBLICATIONS

*An Evaluation of Deeply Decouple Cores* E. Kursun, A. Shayesteh, S. Sair, T. Sherwood, G.Reinman, Journal of Instruction Level Parallelism (JILP) , vol 8. Feb 2006

*Dynamically Configurable Shared CMP Helper Engines for Improved Performance* A Shayesteh, T. Sherwood, S. Sair, N. Jouppi, G. Reinman, ACM Workshop on Design, Analysis, and Simulation of Chip Multiprocessors (dasCMP 05), November 2005, published in SIGARCH Computer Architecture News, vol 33, December 2005

*Reducing the Latency and Area Cost of Core Swapping through Shared Helper Engines* A. Shayesteh, E. Kursun, T. Sherwood, S. Sair, G. Reinman, International Conference of Computer Design(ICCD), Oct 2005

*Tornado Warning: the Perils of Selective Replay in Multithreaded Processors* Y. Liu, A. Shayesteh, G. Memik, G. Reinman, International Conference on Supercomputing(ICS), June 2005

*Low-Overhead Core Swapping for Thermal Management* E. Kursun, G. Reinman, S. Sair, A. Shayesteh, T. Sherwood, IEEE International Symposium on Microarchitecture, Workshop on Power-Aware Computer Systems, Portland, OR, 2004

*The Calm Before the storm: Reducing Replays in the Cyclone Scheduler* Y. Liu, A. Shayesteh, G. Memik, G. Reinman, IBM T.J. Watson Conference on Interaction between Architecture, Circuits, and Compilers, October 2004

*Scaling the Issue Window with Look-Ahead Latency Prediction* Y. Liu, A. Shayesteh, G. Memik, G. Reinman, International Conference on Supercomputing(ICS), June 2004

ABSTRACT OF THE DISSERTATION

# Factored Multi-core Architectures

by

**Anahita Shayesteh**

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2006

Professor Glenn Reinman, Chair

Technology scaling trends have forced designers to consider alternatives to deeply pipelining aggressive cores with large amounts of performance accelerating hardware. One alternative is to *factor out* or *decouple* large structures from critical pipeline loops. In this work, we combine prior techniques in factoring into a cohesive framework and extend this paradigm to more of the processor core. We propose an architecture where the large structures and latency tolerant performance accelerators are factored out of the processor core into helpers and the small and fast $\mu$-core can be augmented with these latency tolerant helpers. This design will reduce the number of accesses to the large, power hungry structures and hence provide power savings with minimal impact on performance. Also this architecture allows the use of slower, more power efficient circuit designs for helpers. As the demands placed on the processor core varies between applications, and even between phases of an application, the benefit seen from any set of helpers will vary tremendously. If there is a single core, these auxiliary structures can be turned on and off dynamically to tune the energy/performance of the machine to the needs of the running application. This is achieved by taking advantage of the dynamic reconfigurability or polymorphism of helpers and allowing a core to adapt to changing applications, workloads, or phases.

As more of the processor is broken down into helpers, and additional cores are added to a single chip, the $\mu$-core design provides a unique opportunity of sharing helpers among the cores. This can be considered a middle point between two extremes of Simultaneous Multiprocessing (SMT) where most processor resources are shared among multiple threads running simultaneously and Chip Multiprocessing (CMP) where each thread is running on a separate core, possibly sharing only the second level on-chip cache.

With the opportunity of sharing helpers among multiple cores, the decisions that are made about these structures become increasingly important. Which resources should be shared, how should they be allocated and how can we efficiently manage their power? To answer these questions at run-time, we need a set of shared helper management policies that can adaptively allocate resources in a way that takes into consideration the needs of each executing workload.

In this work we describe the need for methods that effectively manage these helpers. Along with selectively enabling different helpers, our techniques determine whether to assign exclusive or shared ownership of helpers across multiple cores. By intelligently correlating run-time observations with past measured benefits, our management schemes enable dynamic selection of customized configurations which increase performance and reduce power consumption.

Finally we evaluate the thermal efficiency of the $\mu$-core architecture. We investigate activity migration (core swapping) as a means of controlling the thermal profile of the chip. Specifically, the $\mu$-core architecture presents an ideal platform for core swapping thanks to helpers that maintain the state of each process in a shared fabric surrounding the cores. This results in significantly reduced migration overhead, enabling seamless swapping of cores. Furthermore, we evaluate alternative approaches to spending the area overhead of the additional microcore, including larger $\mu$-cores, CMP cores, and

SMT cores running two-threaded workloads. We evaluate our design compared to different thermal management techniques such as global clock gating and frequency scaling.

# CHAPTER 1

# Introduction

## 1.1 Introduction and Motivation

For more than twenty years, microarchitecture research has emphasized instruction level parallelism (ILP), improving performance by increasing the number of instructions per cycle. In striving for higher ILP, processors have moved towards wider instruction fetch, higher instruction issue rates, larger instruction windows, and increasing use of prediction and speculation. There has been an increase in size of the branch prediction structures to help hide the overall pipeline latency, the cache structures to hide instruction and data memory latency, the physical register file to support a large instruction window and provide more ILP and the other speculative structure to further boost ILP or hide memory latency. However, wire latency has been projected [AHK00] to impact large pipeline structures more than smaller structures, increasing the latency of critical processor loops. These structures force extra access latency as well as latency from routing complexity [SC02]. One approach to provide aggressive cycle time in the face of this latency is to deeply pipeline all aspects of the processor. However deep pipeline increases latency to critical processor loops [BTM02] and leads to very complex, hardware-intensive processors. Larger structures also contribute to the growing power density and thermal problems facing modern processor designs.

One alternative approach to pipelining, is decoupling or factoring out large structures from the critical pipeline loops. Some structures can be broken in two: a small

structure that can hold enough states to provide low-latency performance, and a larger second level structure that can hold sufficient capacity to keep performance high. Other structures can be completely factored from the processor core and have inherent latency tolerance that allows them to avoid becoming critical loops in the processor. Such factoring not only helps to reduce critical loop latency, but can also simplify routing on the critical path and help reduce power from critical path structures.

Another alternative for factoring is multiclustering [FCJ97]. In this approach the structures of the execution core are split into different execution engines. These execution clusters help to scale the instruction scheduling window and the register file. However the performance can be impacted by cross cluster communication of register values and poor cluster utilization.

In [KS02], Smith proposes a processor implementation that consists of several distributed functional units, each fairly simple and with a very high frequency clock. These units communicate via point-to-point interconnections that have short transmission delays.

Yet another approach to gain high throughput, especially in today's multi-programmed compute environment is exploiting thread level parallelism (TLP). This objective has been addressed by adding features to monolithic superscalar processors to support simultaneous multi-threading (SMT)[TEL95] [SMT] or building chip multiprocessors (CMP) [HNO97] [Pow].

SMT processors augment wide (issuing many instructions at once) superscalar processors with hardware that allows the processor to execute instructions from multiple threads of control concurrently when possible, dynamically selecting and executing instructions from many active threads simultaneously. This promotes much higher utilization of the processor's execution resources and provides latency tolerance in case a thread stalls due to cache misses or data dependencies. However misschedulings from

one thread can occupy a significant portion of the processor issue bandwidth, effectively starving other threads [LSM05]. When multiple threads are not available, the SMT simply looks like a conventional wide-issue superscalar.

Traditionally CMPs use relatively simple single-threaded processor cores to exploit only moderate amounts of parallelism within any one thread, while executing multiple threads in parallel across multiple processor cores. If an application cannot be effectively decomposed into threads, CMPs will be underutilized.

From a purely architectural point of view the SMT processor's flexibility makes it superior. However, the need to limit the effects of interconnect delay, which are becoming much slower than transistor gate delays as well as power intensity and its thermal effects will also drive the billion-transistor chip design to be partitioned into small, localized processing elements. For this reason the CMP is much more promising because it is already partitioned into individual processing cores. If Moore's law continues to apply in the chip multiprocessor era, we can expect to see a geometrically increasing number of cores with each advance in feature size. Intel, AMD, IBM, and Sun Microsystems have all introduced their multicore microprocessors.

A critical question in CMPs is the size and strength of the replicated core. Many server applications focus primarily on throughput per cost and power. Ideally a CMP targeted for those applications would use a large number of small low-power cores. However, desktop users are more interested in the performance of a single application of a few applications at a given time. A CMP designed for desktop users would more likely be focused on a smaller number of larger, higher-power cores with better single-thread performance. The challenge is to choose between these conflicting requirements in core complexity. The question becomes even more interesting considering varying behavior among different applications and even different phases of application execution.

With this in mind, Kumar et al. [KJR03] [KTR04] proposed a CMP design comprised of a heterogeneous set of processor cores all of which can execute the same ISA. The heterogeneity of the cores comes from differences in their raw execution bandwidth, cache sizes, and other fundamental characteristics. In a more recent study Kumar et al. [KJT04] propose *conjoined cores* where adjacent cores share over-provisioned structures by requiring only minor modifications to the floorplan. They investigate the possible sharing of floating-point units, crossbar ports, instruction caches, and data caches and detail the area savings that each kind of sharing entails.

To address the same challenge, we propose an alternative design point between SMT and CMP, where multiple simple execution cores (*microcores*) share a set of helpers. The $\mu$-core can be simple enough to allow high clock rates, while allowing single thread to get performance comparable to that of a complex processor by making use of appropriate helpers. This architecture can address several challenges in today's microarchitecture design. Similar to CMP, many high performance execution cores with tightly integrated data caches allow higher operation frequencies and shorter pipelines, and similar to SMT, resources can be shared between threads and allow higher utilization.

This architecture allows further dynamic power/ performance optimizations based on application behavior and phase changes. Best suited helpers can be assigned to the core to meet application demands, resulting in better performance while less used helpers are turned off for more power efficiency. At the same time this architecture provides better area-efficient coverage of the whole spectrum of workload demands that may be seen in a real machine, from low thread-level parallelism (assigning more helpers to each core) to high thread-level parallelism (helpers are shared between higher number of cores). Tuning helpers to application specific needs our design can provide heterogeneous cores similar to [KJR03].

Another aspect of the $\mu$-core in a CMP design, is the opportunity of buffering state in shared helpers for seamless swapping of process among $\mu$-cores. Activity migration (core swapping) has been proposed as an efficient technique for power density reduction [HBA03] and thermal management of the chip. Core swapping on a dual $\mu$-core architecture reduces the performance overhead by buffering state in shared helpers as well as area overhead by merely duplicating the small $\mu$-core.

## 1.2 Dissertation Outline

In this work I propose and study factored architectures both in single and multi-threaded workloads. I investigate performance, power and area efficiency of such architecture and address some of the challenges in the area as well as optimization opportunities provided by this design.

In Chapter 2 I introduce our factored architecture, a $\mu$-core enhanced by a number of different helper engines. I address some challenges in factoring larger structures out of the critical processor loop and evaluate IPC and power performance of this architecture compared with a modern monolithic architecture with similar parameters. Further I look at more power optimizations provided by tuning helper engines to application and phase specific needs.

Chapter 3 extends the study on $\mu$-cores by focusing on policies to manage the helpers. In this chapter I present a novel complexity effective counter guided approach to help the allocation of resources for power efficient design without compromising performance. Furthermore I show how this mechanism can be extended to guide the sharing of resources between multiple cores on a chip. Finally this chapter presents benefit from constructively sharing helpers among multiple cores running the same application.

Chapter 4 focuses on a thermal management opportunity provided by our factored architectures. In this chapter I present and investigate the advantages of activity migration (core swapping) on $\mu$-core architecture as a means for controlling the thermal profile of the chip. Additionally, I study the area overhead of dual $\mu$-core architecture and evaluate different architectures with comparable area including SMT and CMP cores for a two application workload.

Chapter 5 focuses on a case study of two of the helpers used throughout this work: Value prediction and Prefetching. In this chapter I study scalable predictors and evaluate impact of predictor size on performance provided by these structures. Further I explore the interaction between these helpers and other parts of the processor.

Finally Chapter 6 will conclude the dissertation and highlight some future directions for interested reader.

# CHAPTER 2

# An Evaluation of Deeply Decoupled Cores

Despite increases in the transistor count available in the design of future generation processors, emerging technology trends including poor wire latency scaling, increased power density, and reduced transistor reliability threaten to limit processor performance. Future scaling trends challenge microarchitects to improve both clock rate and IPC [AHK00].

Prior work aimed at improving single-threaded IPC involves the discovery and exploitation of instruction level parallelism (ILP). However, wire latency has been projected [AHK00] to impact large pipeline structures more than smaller structures, increasing the latency of critical processor loops. This will impact the size of the branch prediction structures that help hide the overall pipeline latency, the cache structures that hide instruction and data memory latency, the physical register file that supports a large instruction window to provide more ILP, and the other auxiliary speculative structures that further boost ILP or hide memory latency. One approach to providing aggressive cycle times in the face of increasing latency is to deeply pipeline all aspects of the processor, from the branch predictor to the instruction wakeup logic to the cache memories.

While deep pipelining has been effective at increasing operating frequency, one of the reasons that performance lags behind that of the trends set by frequency is the increased latency to critical processor loops [BTM02] from deeply pipelined structures. Furthermore, there is extra latency from routing complexity due to large struc-

tures [SC02]. Larger structures also contribute to the growing power density and thermal problems facing modern processor design through greater power dissipation and longer clock wires [BCB00].

One alternative approach is to *decouple* or *factor out* large structures from critical pipeline loops. Some structures can be broken in two: a small structure that can hold enough state to provide low-latency performance, and a larger second level structure that can hold sufficient capacity to keep performance high. Other structures can be completely factored from the processor core and have inherent latency tolerance that allows them to avoid becoming critical loops in the processor. Such factoring not only helps to reduce critical loop latency, but can also simplify routing on the critical path and help reduce power from critical path structures.

In this study, we combine prior techniques in factoring into a cohesive framework and extend this paradigm to more of the processor core. In [Smi01], the structures that are factored from the main processor pipeline are called *helper engines*. We maintain this naming convention, and refer to the part that remains after factoring as the *$\mu$-core*. A conventional architecture featuring larger structures on the critical path is referred to as a *monolithic* architecture.

While there may be a performance penalty in dividing the processor into a $\mu$-core and helper engines, as measured by instructions committed per cycle, there are many advantages that outweigh these penalties. The primary advantage of such a technique is that the performance critical parts are factored from the larger, more latency tolerant structures. This allows us to target each of these two different domains with different circuit, architectural, or application level techniques. For example, we can target the $\mu$-core with very aggressive, high-power,large transistors to achieve the best cycle time possible. On the other hand, we can use slower, less power-hungry, and more dense transistors for the helper engines.

Power consumption has become a critical design constraint for microprocessors in recent years [Mud00]. As the number of transistors on the next generation microprocessors nears one billion, ever shrinking feature sizes and exponential rises in cooling costs will further complicate this problem. We believe that our $\mu$-core architecture can provide significant power reduction opportunities, not only as the promising initial results of this study illustrate, but also by the flexibility the factored architecture paradigm creates for other more sophisticated power optimization techniques.

Prior reseach [KS02, KGM97, RAC99, BDA01] has explored factoring or decoupling certain parts of the processor pipeline. Our contributions over prior work include:

- An extension of the factoring paradigm to the entire processor pipeline: $\mu$-core architecture
- Analysis of which structures are suitable for factoring - and the individual challenges in factoring these structures.
- An exploration of how far different structures can be factored.
- A detailed power and performance modeling to analyze the benefits of the factored $\mu$-core over an aggressively pipelined conventional core.

The rest of this chapter is organized as follows. In Section 2.1, prior work on the helper engine paradigm is discussed. Section 2.2 describes our factored $\mu$-core architecture. Simulation methodology and benchmark descriptions can be found in Section 2.3, finally Section 2.4 presents performance and power results.

## 2.1 Prior Work on Decoupling and Factoring

Our $\mu$-core design is built upon many past techniques in computer architecture, including past designs that have shown how to partition various architectural structures, and

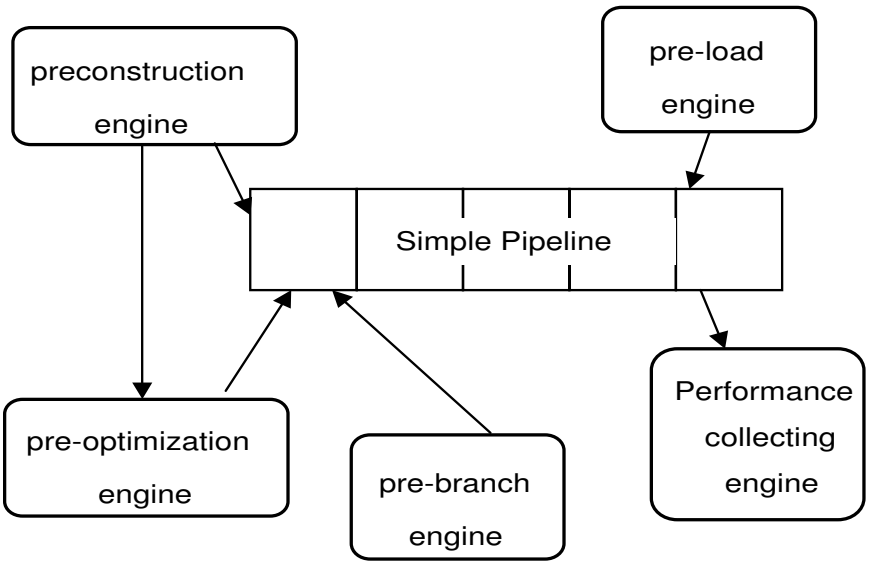


Figure 2.1: Smith's heterogeneous ILDP chip architecture.

that have dynamically change the size of various structures at run time.

In this section we discuss the relationship between our design and the most relevant prior work. In Section 2.2, we will detail the prior work for each individual helper engine when discussing the implementation of that helper engine.

The work most related to our is [Smi01]. In [Smi01], Smith proposes a processor implementation that consists of several distributed functional units, each fairly simple and with a very high frequency clock. These units communicate via point-to-point interconnections that have short transmission delays. He then describes how surrounding this simple core pipeline with *helper engines* that perform speculative tasks off the critical path results in enhanced overall performance 2.1. Examples of helper engines iclude the pre-load egnine of Roth and Sohi [RS99] where pointer chasing can be performed by a special processing unit. Another is the branch engine of Reinman et al [RAC99]. An even more advanced helper engine is the instruction co-processor described by Chou and Shen [CS00]. Helper engines have also been proposed for garbage

collection [HS00] and dynamic correctness checking [Aus99](figure 2.1. Since the helper engines are off the critical path, they can use slower transistors to reduce static power consumption. This is also the motivation behind our factored design, where the speculative structures are shrunk to a bare minimum size to support near ILP but they are duplicated in larger sizes outside of the critical path for extracting distant ILP. On a follow-up paper [KS02], Kim and Smith discuss the microarchitecture and ISA that implements this distributed processing paradigm, which utilizes hierarchical register files and with a global register file to hold global state.

Another type of factoring is multiclustering [FCJ97]. In this approach, the structures of the execution core are split into different execution engines - much like the distributed functional units of [Smi01]. These execution clusters help to scale the instruction scheduling window (including the overall issue bandwidth) and the register file. However, performance can be impacted by cross cluster communication of register values and poor cluster utilization.

Both of these approaches ([KS02][FCJ97]) could be used to further factor the structures of our $\mu$-core, and we will consider this enhancement for future work. However, we consider factoring the structures of the entire processor core including some popular performance accelerating helper engines, and explore the impact that this factoring has on both performance and power for an aggressive clock.

Related work discussion continues in Section 2.2. While presenting the individual blocks in our microcore architecture, we also discuss the comparison with related studies.

## 2.2 Factored Architectures

The classic example of a structure being split into smaller pieces is the cache memory. Because a single monolithic cache would either be too slow to hide latency or too small to have a reasonable hit-rate, caches are typically divided into a hierarchy of memories, with small fast memories near the processor and large slow memories near (in terms of access ordering) off chip storage. This hierarchical design methodology is common knowledge among architects because cache memories have been a part of the design of modern processors for many years and thus have been subject to much study. However, as processors sink into deep-sub micron technologies, we believe that other structures might benefit from a similar transformation as well.

The main idea behind factored architectures is to move a set of larger structures out of the regular processor core, resulting in a tiny core with only the necessary components included.

Figure 2.2 shows a conventional(monolithic) core that utilizes a variety of architectural features commonly found in modern microprocessors. These features are necessary to extract as much ILP from the application as possible. However, some of these units such as register file, first level caches, and branch predictor, lie on the critical path of the processor and can therefore have a significant impact on cycle time. Figure 2.3 illustrates our factored architecture, with most of the larger blocks moved out of the critical loops. We demonstrate three different types of factored structures:

- Hierarchical extensions: Caches and branch predictor (shown in light gray)
- Complete factorization: Value predictor and data prefetcher (shown in dark gray)
- Hybrid factorization: Register file and ROB (shown with gray stripes)

Hierarchical extensions remove larger structures from the processor core and leave

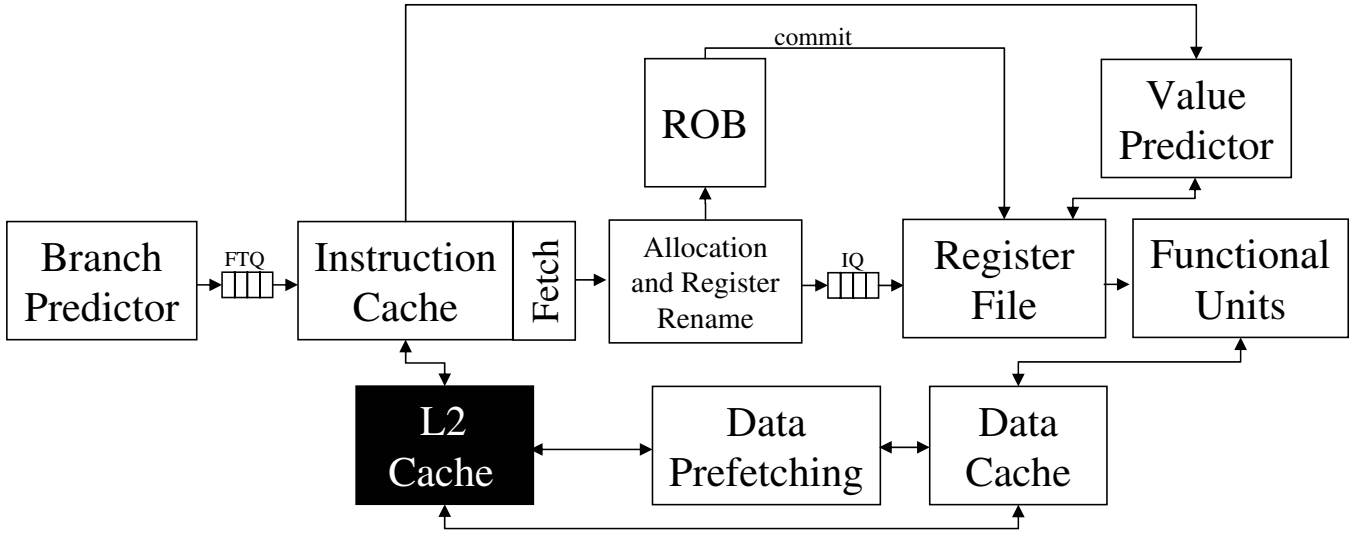


Figure 2.2: The monolithic core

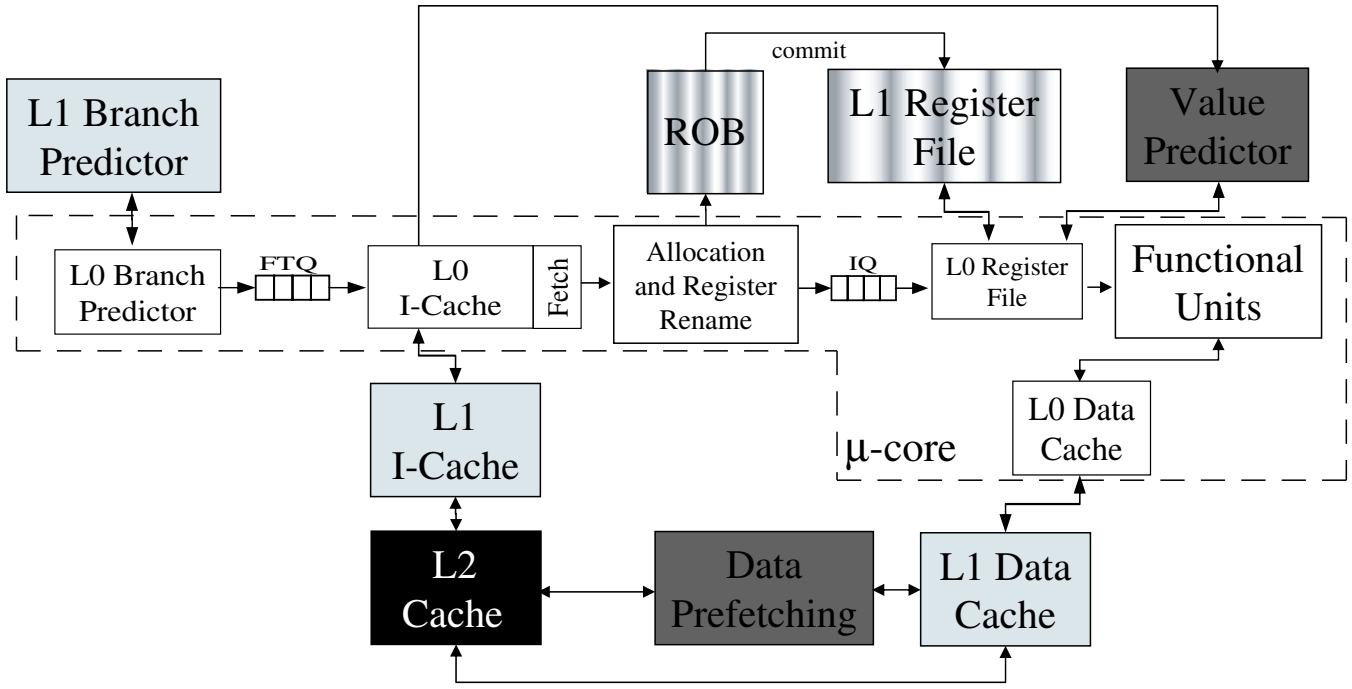


Figure 2.3: The factored $\mu$-core architecture

a much smaller version of the original structure in the core. This benefits structures that are tightly integrated with the processor core for high performance (such as the data cache), particularly those with high port counts (such as the register file). Complete factorization is suitable for structures with flexible functionality and high latency tolerance that can be configured to require little or no interaction with the core. Some architectural features can use a hybrid of these two approaches (hybrid factorization) - completely factoring some components (e.g. structures of the commit stage) and hierarchically extending others (e.g. the register file).

There have been a number of previous studies on independently factoring individual parts of a processor, as will be introduced in this Section. Yet these studies have not explored what happens when this principle of factoring is applied to more than one structure of the core. We believe our study is unique in synthesizing all of these different ideas together and studying their impact on overall system power.

While structures such as caches are fairly easy to factor, other structures require more extensive analysis and work. In this section we describe how each of the major structures may be factored, and how we model that in our processor simulator.

### 2.2.1 Data Cache

We move the L1 data cache out of the core processor pipeline and replace it with a smaller L0 cache, as in [KGM97]. The L0 extends the cache hierarchy, therefore we access the L1 data cache on an L0 miss.

### 2.2.2 Instruction Fetch

Similar to the data cache, we insert a smaller L0 instruction cache and move the L1 instruction cache out of the $\mu$-core. To compensate for the smaller cache size, we use

out-of-order instruction fetch as described in [SRP97]. In this scheme, a placeholder is used in the instruction fetch queue (IFQ) to maintain program order - and the execution core stalls if the next entry to be consumed from the IFQ is still in flight. We model the complexity this brings to the IFQ by implementing the equivalent of an MSHR [Kro81] for the instruction cache. An MSHR tracks cache misses while they are being satisfied to allow the cache to continue operation even in the face of a miss (i.e. a nonblocking cache). In the IFQ, we keep the index of the MSHR that will hold the missing instruction cache block along with the instruction's offset inside the cache block. Subsequently, when we see an index to the MSHR in the next IFQ entry to be consumed, we access the MSHR directly to get the required instruction word. The MSHR has a valid bit indicating when the decoding of the instruction can commence. For the case that MSHR runs out of indices or IFQ entries, we stall instruction fetch. However, we found this to be extremely rare without an observable impact execution performance. Out-of-order fetch is also used in our baseline monolithic case, as it has the additional benefit of reducing instruction cache miss stalls.

### 2.2.3 Data Prefetch

We model a stream buffer architecture [Jou90] guided by a stride-filtered Markov predictor as proposed in [SSC00]. In the monolithic microprocessor, the address predictors guide prefetches to the stream buffer, which is accessed in parallel with the data cache. For the case of factorization, the address predictors and stream buffers are moved further away from the core pipeline. The stream buffers of the factored prefetch engine are *only* accessed on L0 misses. We allow a single prediction and a single prefetch per cycle, guided by the address predictor trained on the L0 miss stream.

### 2.2.4 Value Prediction

Value prediction [LS96a] is one approach to break true data dependencies and create more instruction level parallelism in an application. We use a hybrid value predictor [WF97a], yet we only predict load instructions. This structure can be accessed early in the pipeline as we only need the PC of the instruction to make the prediction. Our predictor makes a single prediction per cycle. We make use of an extra bit associated with each instruction in the instruction cache to dynamically mark instructions for value prediction. In this study, we only mark instructions that are loads and that can be confidently predicted by our value predictor.

One way of factoring the value predictor is to store the predicted load value in the register allocated to the load instruction we are predicting. When the data access completes, it overwrites the predicted value in the destination register with the actual load value. If the predicted value and the actual value do not match, a checker engine similar to the ARB [FS96] detects the misprediction and squashes the mispredicted result and its dependents with the same hardware that is used in a branch misprediction.

### 2.2.5 Branch Address Prediction

Our architecture makes use of a basic block target buffer (BBTB) [YP92], a branch address predictor that predicts an entire basic block each cycle. The PC at the head of the basic block serves as an index to the predictor, which returns a target address, a fallthrough address, a branch type, and two per-branch prediction counters: one to make per-branch direction predictions, and one to arbitrate between the per-branch prediction and the global branch direction prediction. In the $\mu$-core design, we reduce the size of the BBTB in the core pipeline and add a second level BBTB as done in [RAC99]. Similarly we decouple branch prediction from the instruction cache using

fetch target queue (FTQ). On a first level BBTB miss, the second level BBTB is probed and fetch stalls until a response is received from the second level. If the second level misses, we guess a fixed fetch block size and continue fetching until a misprediction is detected.

### 2.2.6 Register File and Commit

In the factored architecture we use a multi-level register file similar to the one proposed in [BDA01]. The basic differences are that we model an inclusive register file hierarchy where the second level register file (RF1) includes all the state contained in the first level register file (RF0). Since commit is a vital part of the processor pipeline, this helper engine is never disabled, but the second level register file that can be dynamically configured with either 128 or 512 entries at runtime depending on the needs of the application. On a branch misprediction, the second level register file recovers the state of the first level register file.

The first level register file releases physical registers when they are no longer the most recent version of a given logical register and when there are no more in-flight instructions that are waiting to consume their value. The second level register file releases a given physical register *pr* that is mapped to a logical register *lr* only after the instruction defining *pr* has committed and a subsequent instruction remapping *lr* has committed. This ensures that the value will not be required for a branch misprediction or to provide precise exceptions. We leverage the fact that the second level register file is only used for commit and branch recovery by factoring the second level register file, reorder buffer (ROB), and commit hardware into a helper engine, similar to a future file [SP85]. However, our ROB only holds status information for instructions in the pipeline – all result values are stored in the register file, as in the Pentium 4 [HSU01]. Moreover, unlike the future file, our first level register file holds a subset of the total

physical registers rather than only the most recent versions of logical registers (effectively holding more state than the future file). In the core pipeline, a simple tag allocator handles ROB allocation, with a tag implicitly mapping to an ROB entry and providing a means to hook up dependent instructions. The register map remains in the core pipeline.

This is a hybrid of complete factorization and hierarchical extension, as the register file is extended with a second level structure, but the commit hardware and ROB are completely factored, with only tag allocation in the ROB impacting the core timing.

## 2.3 Methodology

The simulator used in this study is derived from the SimpleScalar/Alpha 3.0 tool set [BA97], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user level instructions, performing a detailed timing simulation of a dynamically scheduled microprocessor. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction. Latency values for the caches and register files were obtained using CACTI [SJ01] for a 100nm process technology at 4 GHz. All cores in this study are targeted at this frequency unless stated otherwise.

We used the SPEC2000 benchmark set for our experiments. Although the results are gathered for all the benchmarks, we only show results for a randomly selected subset of programs for clarity. The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C and C++ compilers under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). We picked the 4 most dominant phases as determined by the hardware phase detection technique described in [SSC03] and simulated these phases as representative samples of the program. On average,

| Benchmark | Phase 1 | Phase 2 | Phase 3 | Phase 4 | Total |
|---|---|---|---|---|---|
| applu | 0.23 | 0.16 | 0.16 | 0.10 | 0.65 |
| apsi | 0.46 | 0.09 | 0.06 | 0.05 | 0.65 |
| art | 0.43 | 0.30 | 0.12 | 0.07 | 0.93 |
| bzip | 0.15 | 0.15 | 0.12 | 0.07 | 0.48 |
| crafty | 0.40 | 0.22 | 0.14 | 0.05 | 0.81 |
| equake | 0.45 | 0.14 | 0.05 | 0.03 | 0.67 |
| eon | 0.43 | 0.15 | 0.09 | 0.08 | 0.74 |
| galgel | 0.35 | 0.29 | 0.11 | 0.06 | 0.81 |
| gap | 0.20 | 0.18 | 0.13 | 0.12 | 0.62 |
| mcf | 0.26 | 0.20 | 0.14 | 0.08 | 0.68 |
| mesa | 0.71 | 0.16 | 0.03 | 0.03 | 0.92 |
| parser | 0.09 | 0.04 | 0.04 | 0.03 | 0.20 |

Table 2.1: Percentage of execution in each phase of the benchmark

they accounted for approximately 70% of the execution time of each benchmark. All benchmarks were simulated using the *ref* inputs.

Table 2.1 indicates percentage of execution in each of those phases for the benchmarks we present. All benchmarks were simulated using the *ref* inputs.

Table 2.2 presents the simulation parameters for the monolithic and $\mu$core architectures we explore in this work. We include an 8K entry gshare branch predictor in out model. We have made significant modifications to SimpleScalar to model the various speculative techniques in this architecture.

Note that the four cycle difference in branch misprediction penalty is the extra latency attributed to the larger branch predictor, register file and instruction cache in the monolithic core. Since the monolithic core and the $\mu$-core are running at the same clock frequency, the results we present have an optimistic view (i.e. an upper-bound on the performance) of a three-block ahead branch predictor [SJS96].

Our power analysis includes an accurate model of dynamic and leakage power for both the monolithic and $\mu$-core architectures. We used process parameters for a *100nm* process at 4GHz with a 1V supply voltage.

We performed dynamic power analysis based on the power models proposed by Wattch [BTM00]. Main processor units are modeled such that they fall into following categories: array structures(e.g. data and instruction caches), fully associative content-addressable memories(e.g. TLBs), combinational logic/wires (e.g. functional units) and clocking. Various clock gating schemes were explored during the experiments. The results are based on the most aggressive clock gating strategy in Wattch, where power scales linearly with port (or unit usage) and idle units dissipate only 10% of the maximum power.

Leakage power contributes a significant portion of the overall power for process technologies such as ours. Hence, we incorporated leakage power analysis, based on the leakage power models from [ZPS03]. Specific leakage parameters (such as Vth variations, Tox, etc) for 100nm process are also from [ZPS03].

The latency values for the main units in the processor are extracted using CACTI [SJ01]. We modeled serial and parallel versions of cache-like structures for power efficiency and access latency respectively. We did a serial cache access on the different levels of the hierarchy, and derived the latencies by adding latencies of the differnt levels. These values are listed in Table 2.2 and Table 2.3.

## 2.4 Results

In this section we evaluate the impact of factored architectures based on the performance and power dissipation results from our experimental analysis.

### 2.4.1 Impact of Factoring

Figure 2.4 compares the performance and power dissipated by three architectures relative to the monolithic core.

- $\mu$-core Architecture (*U*): excludes the first level (L1) data cache, L1 instruction cache, L1 BBTB, value predictor, and data prefetcher. The L2 register file is tuned to hold at most 128 registers. This demonstrates how well the architecture would perform if all structures were simply scaled down without factoring.

- Optimized Monolithic(*OM*): a conventional core with serial access caches, BBTB, and register file. These structures are much more energy efficient, as they pinpoint the specific data array that is required for each access to the structure, consuming energy for that array alone. However, energy savings come at a cost of latency in accessing the structures. We optimistically assume that all structures can be pipelined to accomodate this increased latency without impacting the cycle time of the processor, and further assume that the BBTB can be pipelined using a technique like multiple block ahead prediction [SJS96] without any loss of accuracy.

- $\mu$-core Architecture with Helper Engines (*UwH*): all of the helper engines are activated and the L2 register file is tuned to 512 registers. These configurations are summarized in Table 2.3.

In Figure 2.4 we present the average of individual phase results, weighted by the amount of execution time contributed by that phase. All results are normalized to the performance of the monolithic core. All architectures make use of simple serial access structures for address and value prediction to save energy.

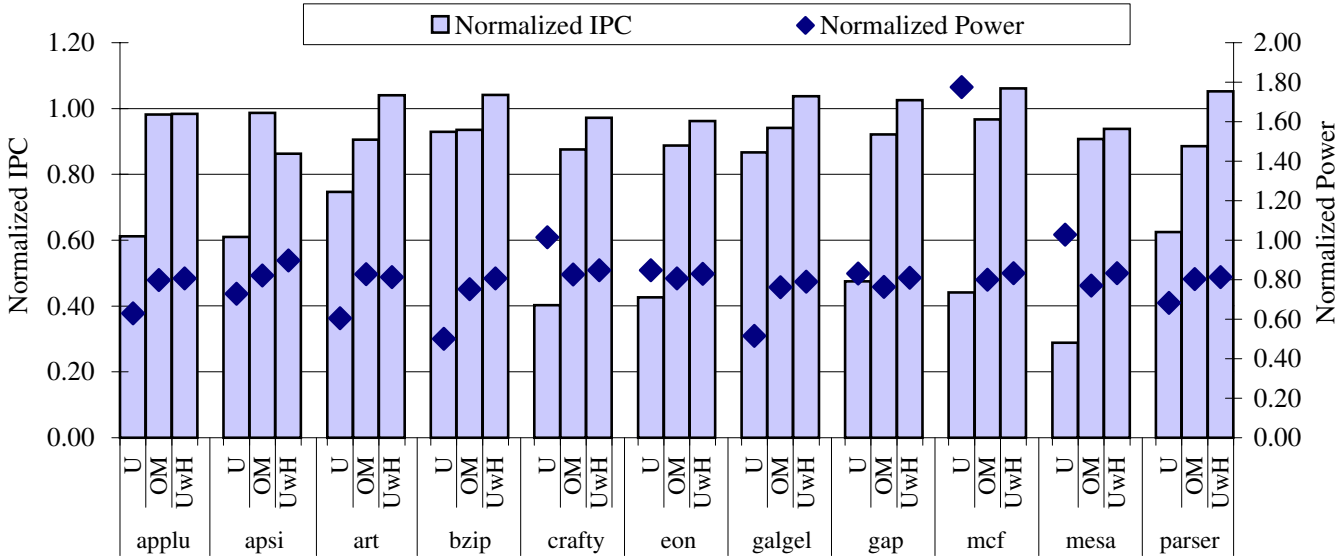


Figure 2.4: Performance and power comparison of the monolithic core and our $\mu$-core architecture

The optimized monolithic (OM) architecture provides a dramatic power reduction over the baseline monolithic architecture (21% reduction on average). Yet, it can also impact performance due to the added latency of the core structures, dropping IPC by 10% on average.

The main performance benefit of the $\mu$-core (U) over the monolithic architecture comes from the fact that the smaller structures in the $\mu$-core have shorter access latencies, as in the case of the L0 caches. However, for the $\mu$-core architecture, performance is severely impacted by insufficient capacity in the caches and BBTB, insufficient instruction window size, and the lack of value prediction and data prefetching. This drop in performance can actually increase the total power consumed by the $\mu$-core architecture over the baseline monolithic architecture, as seen in em crafty, *mesa*, and especially *mcf*. The larger power dissipation is due to various reasons, including: an increase in L2 activity due to L0 misses, wasted power on wrong path execution, and simply the overall increase in run time (i.e. a larger amount of leakage power).

However, when all helper engines are enabled, the $\mu$-core architecture is able to achieve higher performance. Even though the structures on the $\mu$-core are much smaller than their monolithic counterparts, they are able to capture enough ILP such that the pipeline remains full while accesses to the larger helper engines are pending. In fact benchmarks such as *art*, *bzip*, *galgel*, *gap*, *mcf*, and *parser* are able to outperform the monolithic case, despite the latency of their helper engines. Apsi experiences a 14% drop in performance due to the reduction in state tracked by the L0 data cache. As will be seen, the miss rate for the first level data cache in apsi increases in the $\mu$-core.

Note that the primary impact of the $\mu$-core architecture is on the power consumption of the factored hardware. The fact that it minimizes performance impact while improving power consumption is in stark contrast to most other low-power techniques

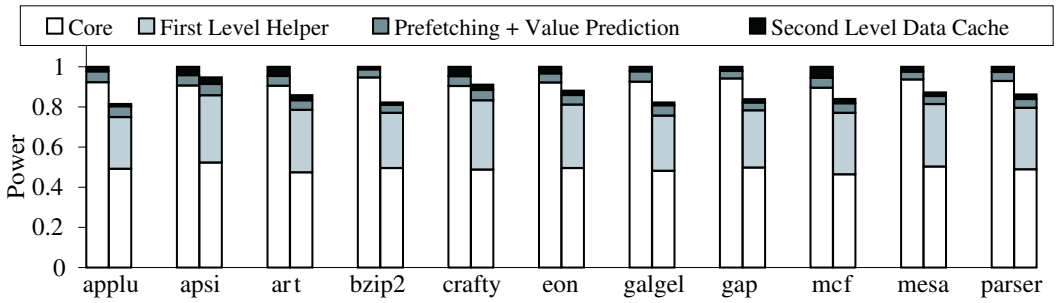


Figure 2.5: Power breakdown for Monolithic and $\mu$-core architectures

that try to take advantage of the slack in programs by slowing down the processor. In those schemes, the goal is to maximize power savings with the least amount of slowdown possible. Our results demonstrate that we are able to save as much power as the optimized monolithic (OM) architecture (within 3% on average) yet with performance comparable or better than the baseline monolithic architecture (1% improvement on average).

Figure 2.5 shows how different helper engines contribute to power consumption in our $\mu$-core architecture. Power values are normalized to the baseline monolithic architecture (shown in the first bar). The second bar represents the power of the $\mu$-core with all helper engines enabled. We separated value prediction and prefetching from other helpers, because they are completely factored. *First level helper* accounts for power consumed in all the other helper engines including the DL1, IL1, BBTB1, and the hierarchically extended register file.

Figure 2.5 also illustrates the reduction in core power in the $\mu$-core compared to the baseline monolithic architecture. The core dissipates 90-94% of the power in baseline monolithic architecture, compared to 54-60% in the $\mu$-core architecture. With more power dissipated in the latency tolerant helper engines, the $\mu$-core is able to take better advantage of optimizations trading performance for power.

Further data is presented in Table 2.4 and 2.5. Despite an increase in the L0 miss rates for the hierarchically extended structures (note that the register file design does not miss), the $\mu$-core with helper engines (UwH) is still able to reduce the power impact with a slight average IPC improvement.

### 2.4.2 How Far Can We Factor?

To better explore how far we can scale the size L0 structures without performance loss, we simulated a fairly large design space of structure sizes for the hierarchically extended helper engines.

Figure 2.6 presents results for different data cache sizes and associativities. We adjusted latencies for each configuration using CACTI [SJ01]. While smaller associativities seem to help reduce power dissipation, associativity is critical for high performance in applications like *apsi* and *applu*. Capacity is also important – applications such as *art* and *galgel* see a degradation in performance when going to smaller sized caches, even with comparable associativities.

Results for different BBTB sizes (in number of entries) and associativities are illustrated in Figure 2.7. Although some benchmarks are unaffected by reduced BBTB state, others such as *crafty*, *eon*, *gap*, *mesa*, and *parser* see dramatic degradation in performance as overall predictor size or associativity is decreased. Despite available backing storage from the helper engines, there is clearly a critical amount of state that must be maintained in the core to achieve high performance. We do not present results for the L0 instruction cache size, as almost all of the benchmarks seem to be able to tolerate small L0's. Only *mesa* saw any significant IPC degradation, even for a 1KB IL0.

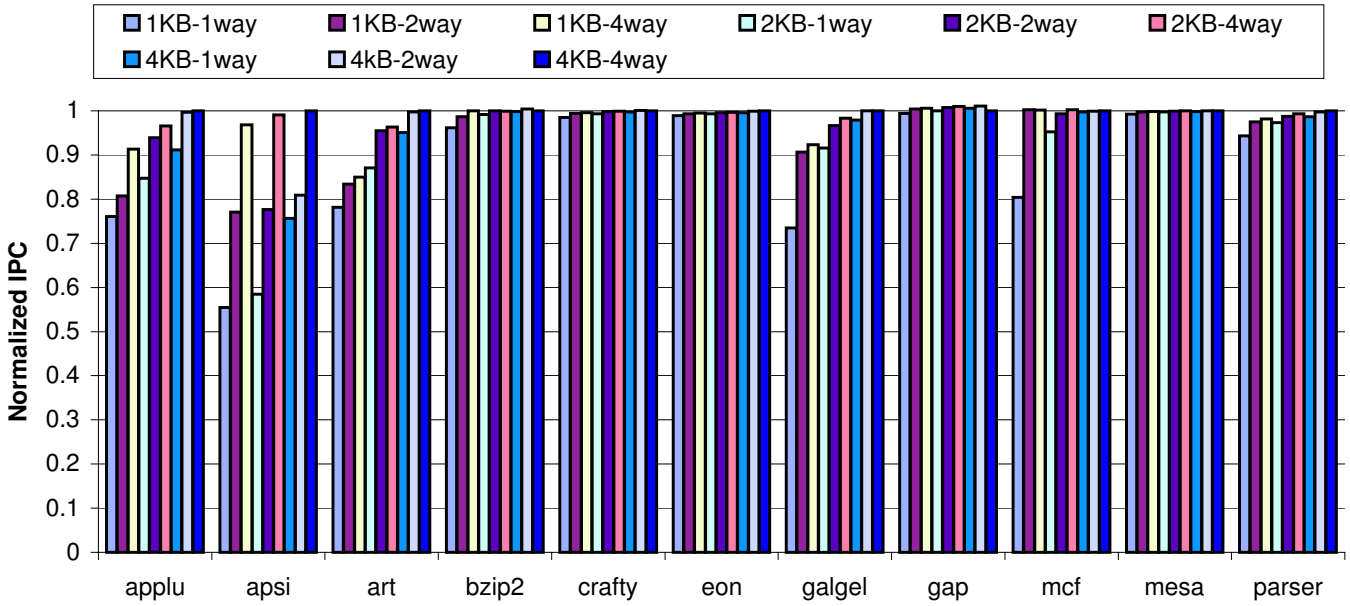


Figure 2.6: IPC impact of L0 data cache scaling on $\mu$-core with backup secondary L1 data cache

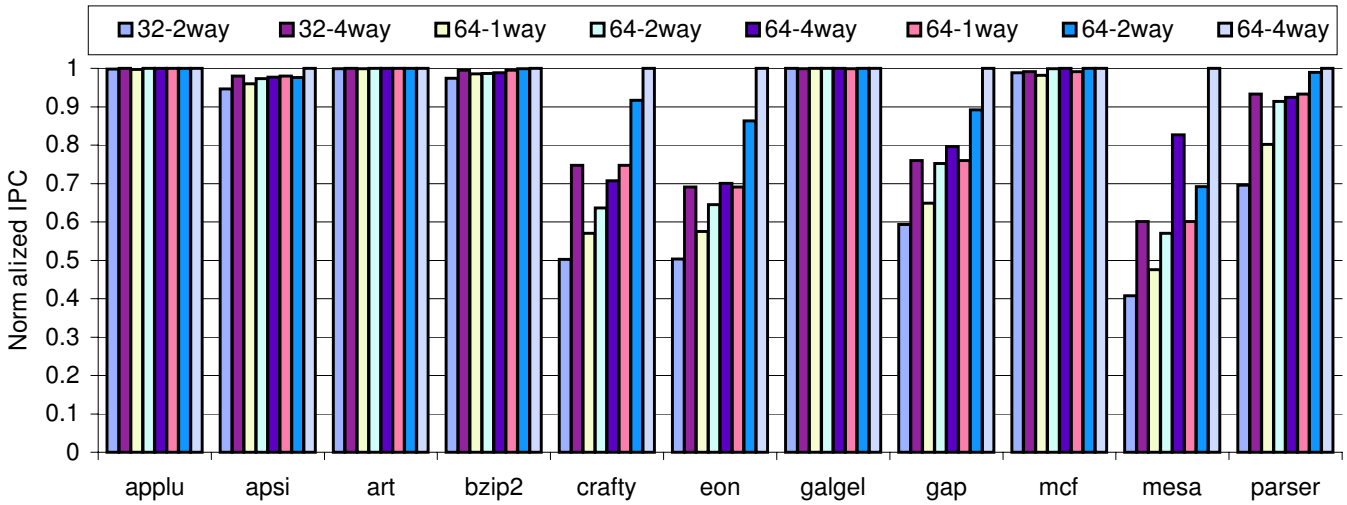


Figure 2.7: IPC impact of L0 BBTB scaling on $\mu$-core with backup secondary L1 BBTB

### 2.4.3 Tuning Helper Engines

The results presented thus far, have assumed that all of the helper engines are on at all times. Prior research have shown that programs can be divided into blocks of similar execution behavior, called phases (this will be discussed in more details in chapter 3). Various techniques have been proposed to extract the phase behavior of a program. Program phase behaviour can provide even more significant energy savings by allowing us to selectively turn on/off a subset of the available helper engines.

Note that it would be possible to dynamically tune the monolithic architectural structures to a particular program phase (as in [Alb99]), but this approach limits how far the latency of these structures can be reduced, as the structures still consume the same chip area whether they are on or off, and can still impact wire routing and wire latency. Tuning the helper engines is one alternative to this, and will migrate the control logic needed to perform the reconfiguration and the actual larger structures themselves away from the core. The $\mu$-core structures could still be dynamically tuned as in prior work to provide even more power savings.

The flexibility of helper engine design allows us to independently control individual units so that only the critical helpers for the execution of a given application phase are actually on, thus reducing overall power dissipation.

In the $\mu$-core design, there are number of helper engines (our initial implementation has six) all working together to speed up the execution of a program. The key observation here is that for any given program phase, some helpers are more helpful than others, yet in the simplest design they are all being used and all consuming power. In many cases it is possible to turn off the non-critical helper engines to reduce the power consumption. To achieve this, we need a policy for turning helper engines on/off or tuning them (e.g. in the case of the L2 register file).

| | | r | v | b | i | p | c | IPC | Power |
|---|---|---|---|---|---|---|---|---|---|
| applu | 1 | X | | | X | X | X | 0.97 | 0.92 |
| | 2 | | X | | X | X | X | 0.97 | 0.76 |
| | 3 | X | | | X | X | X | 0.95 | 0.92 |
| | 4 | | | | X | X | X | 0.95 | 0.71 |
| art | 1 | | | | | X | X | 0.98 | 0.64 |
| | 2 | | | | | X | X | 0.97 | 0.65 |
| | 3 | | | | | X | X | 0.96 | 0.64 |
| | 4 | | | | | X | X | 0.97 | 0.65 |
| crafty | 1 | | | X | X | | X | 0.98 | 0.63 |
| | 2 | | | X | X | | X | 0.99 | 0.63 |
| | 3 | | | X | X | | X | 0.99 | 0.63 |
| | 4 | | | X | X | | X | 0.98 | 0.63 |
| galgel | 1 | X | | | | | | 0.97 | 0.86 |
| | 2 | | | | | X | X | 0.97 | 0.63 |
| | 3 | X | | | | | X | 0.96 | 0.87 |
| | 4 | X | | | | | X | 0.96 | 0.87 |
| mcf | 1 | | X | | | X | X | 0.98 | 0.71 |
| | 2 | | X | | | | X | 0.98 | 0.67 |
| | 3 | | X | | | X | X | 0.97 | 0.71 |
| | 4 | | X | | | | X | 0.98 | 0.67 |
| parser | 1 | | X | X | X | | X | 0.98 | 0.68 |
| | 2 | | X | X | X | X | X | 0.96 | 0.74 |
| | 3 | | | X | X | | X | 0.96 | 0.64 |
| | 4 | | X | X | X | X | X | 0.96 | 0.73 |

| | | r | v | b | i | p | c | IPC | Power |
|---|---|---|---|---|---|---|---|---|---|
| apsi | 1 | | | | X | X | X | 0.96 | 0.70 |
| | 2 | | | X | X | X | X | 0.98 | 0.71 |
| | 3 | | | | X | X | X | 0.96 | 0.70 |
| | 4 | | | X | X | | X | 0.98 | 0.65 |
| bzip2 | 1 | | | | | X | | 0.95 | 0.61 |
| | 2 | | X | X | X | X | X | 0.96 | 0.72 |
| | 3 | | | | | X | | 0.96 | 0.60 |
| | 4 | | | | | | X | 1.00 | 0.56 |
| eon | 1 | | | X | X | | X | 0.99 | 0.64 |
| | 2 | | | X | X | | X | 0.99 | 0.63 |
| | 3 | | | X | X | | X | 0.98 | 0.63 |
| | 4 | | | X | X | | X | 0.97 | 0.64 |
| gap | 1 | | | X | X | | X | 0.96 | 0.63 |
| | 2 | | X | X | X | | X | 0.97 | 0.71 |
| | 3 | | | X | X | | X | 0.97 | 0.63 |
| | 4 | | X | | | X | | 1.00 | 0.70 |
| mesa | 1 | | | X | X | | X | 0.97 | 0.63 |
| | 2 | | | X | X | | X | 0.97 | 0.64 |
| | 3 | | | X | X | | X | 0.97 | 0.64 |
| | 4 | | | X | X | | X | 0.97 | 0.64 |

Figure 2.8: Optimal configuration. A marked square indicates that for the given phase the corresponding helper engine should be turned “on” for optimal or near-optimal performance.

However it is not straightforward to determine which helper engines should be turned on/off to minimize the power for a given performance target. The performance improvement due to each helper engine is *not* independent of the other helper engines. For example, *crafty* has an interesting branch behavior. When the first level branch predictor is off, turning on any other helper engine improves IPC by at most 5%. Turning the BBTB1 on alone doubles IPC (100% speed up). With this improved branch prediction, turning on the first level data cache and instruction cache together gives another 30% improvement over the baseline with all helpers off. This is a fairly simple example, the real interactions between helpers like value prediction and prefetching are far more complex.

Figure 2.8 illustrates the optimal configuration (according to $\frac{Power}{IPC^2}$ metric) for the four most dominant phases of each benchmark. The columns of the table represent the helper engines (r=ROB, v=value prediction, b=branch prediction, i=instruction cache, p=prefetching, c=data cache). A marked square indicates that for the given phase the corresponding helper engine should be turned "on" to gain optimal performance.

Helper engine configurations usually refer to "on" or "off". However, the L2 register file is never turned off, but is tuned so that either 128 or 512 entries in the file are active and available for renaming. The final two columns show the normalized IPC and total power dissipation for each configuration relative to keeping all helper engines on.

The configurations in Figure 2.8 all achieve performance within 5% of the highest performance configuration. They were found by a brute force search of the design space, simulating every possible case and taking the configuration with the lowest $\frac{Power}{IPC^2}$ measurement. Figure 2.8 demonstrates the fact that for some benchmarks very different configurations are best suited for different phases.

It is important to note that the overall power and performance varies significantly

depending on which helper engine is turned off. We have investigated all possible on/off configurations of helper engine, even though only the configurations that have optimal performance ($\frac{Power}{IPC^2}$) are presented in Figure 2.8.

Another critical bit of knowledge that we can take away from Figure 2.8 is that the performance improvement from each helper is *not* independent of the status of the other helpers. For example, it may be the case that using only a large branch predictor or only a large instruction cache provides no additional performance, but by using both in tandem, there are significant benefits to be had. The benchmarks *crafty* and *eon* both exhibit this behavior. Even more complex interactions exist between the value predictor, prefetcher and data cache.

For example, the fourth phase of *eon* (as shown in Figure 2.8) can substitute the extra register file space for the prefetcher to achieve the same level of performance. Another example is the fourth phase of *bzip2*, where we can substitute both the data cache and value predictor for the extra register file space and achieve nearly the same level of performance.

Figure 2.9 and 2.10 shows the IPC and power consumption for all configurations in our design space for two benchmarks applu and bzip2. Power values are normalized over the maximum power among four phases for each benchmark. Note that this is a different normalization than what is done in Figure 2.8. The dark lines in the figure connect configurations with optimal IPC for a given level of power for each phase. The optimal configuration reported in Figure 2.8 lies on this line based on the metric described above. Each cluster of points indicates the use of a certain helper that had a noticable effect on either performance or power.

Applu, which has high address predictability, benefits significantly from data prefetching. It sees speedups as high as 60% for phases 1 and 3, and 20% for phases 2 and 4. This explains the distribution of points and the big jump for phases 2 and 4 in this

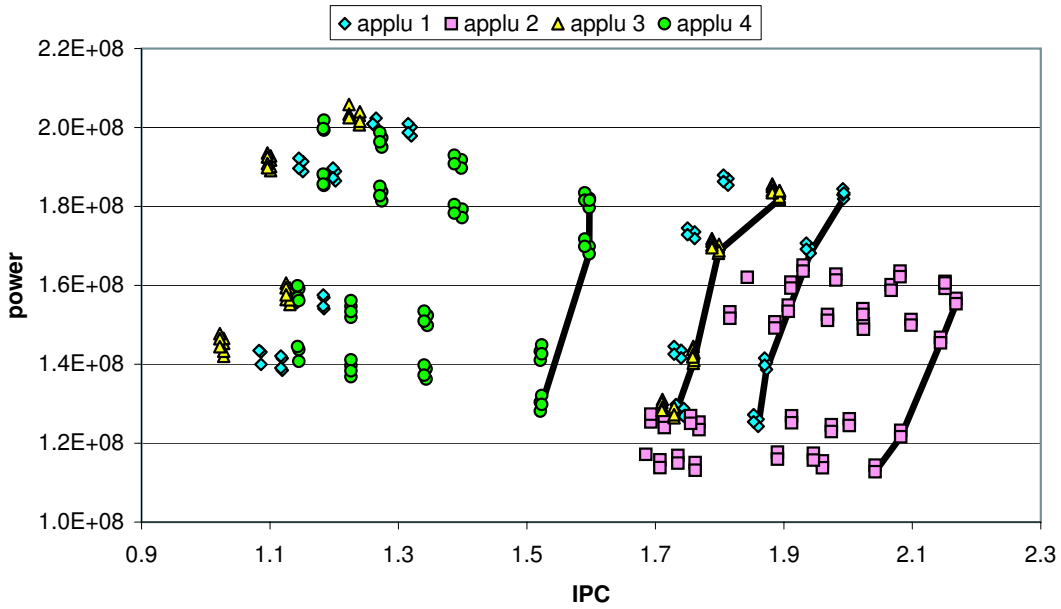


Figure 2.9: Design space explored for *applu*

Figure. Optimal configurations for phases 1 and 3 lie at the top of the lines where data cache, ROB and instruction cache helpers are on. Three clusters down the line refer to turning off these helpers accordingly. For phase 2, a similarly optimal configuration lies on the upper cluster, whereas for phase 4, the optimal configuration belongs to the second cluster. The next clusters on this line indicate increasing the register file size and turning on value prediction, which due to their high power consumption are not optimal based on our metric.

Bzip2 does not show the wide range of IPC and performance as compared with applu, but it has an interesting variation among the four phases. For Phases 1 and 3 there is 10% improvement from prefetching alone, and adding other helpers increases power without significant improvement in IPC. Phase 4 sees minimal benefit from turning on helpers. The vertical line in Figures 2.9and 2.10 indicates that turning on

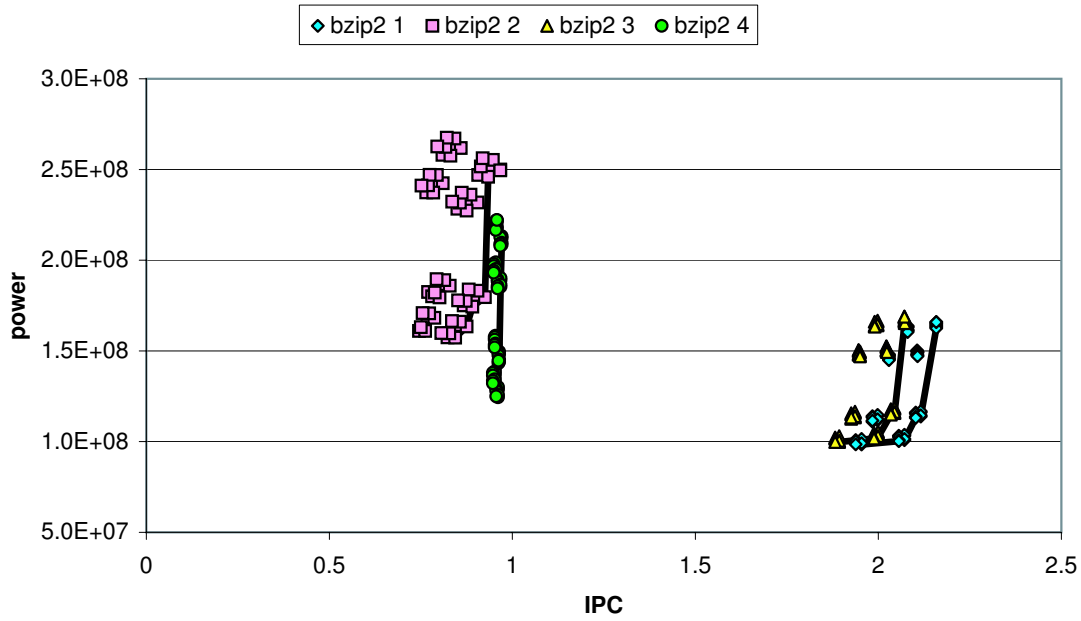


Figure 2.10: Design space explored for *bzip2*

helper engines will only increase power consumption. The optimal configuration uses the data cache which leads to a 2% improvement for this phase. Phase 2 shows four clusters of points with different power consumption. The additional power in the upper clusters is due to increasing the register file size, which does not help performance for this phase. The clusters indicate different combinations of turning on and off branch prediction and value prediction. The optimal configuration reported in table 2.8 keeps the helper engine register file size small and keeps all the other helpers on.

Obviously, a brute force search among exponential number of possible cases is not a good design choice for a runtime system. Therefore we include hardware use counters along with each helper engine to determine the utility of each structure. For each program phase (dynamically identified as in [SSC03]), the utility of each structure can be determined after the initial execution of that phase, where the usage counts of each helper engine determines whether or not that engine should be activated in

that phase. Usage counters will only be incremented when a helper engine is actually successfully used, not just probed (i.e. the number of L1 data cache hits would be counted, not the number of accesses). Each helper engine would have an activation threshold that the usage counter would need to exceed before it is considered necessary for that phase. This dynamically formed phase configuration can be stored in a predictor and used guide future configuration of the helper engines. However, such an approach is not guaranteed to find an optimal configuration. Future work will explore such approaches and more intelligent schemes to selectively enabling helper engines.

| | Monolithic Core | $\mu$core | |
|---|---|---|---|
| | | L0 | Helper Engines |
| Instruction Window and Physical RF | 512 entry ROB<br>80 entry RF0, 512 entry RF1 | 80 entry RF0 | 128 or 512 entry ROB<br>128 or 512 entry RF1 |
| BBTB | 1024-entry 4-way set associative | 128-entry 4-way set associative | 1024-entry 4-way set associative |
| L1 Data Cache | 16KB 4-way set associative, dual port with a 32 byte block size, 4 cycle latency | 2KB 4-way set associative, dual port with a 32 byte block size, 3 cycle latency | 16KB 4-way set associative, single port with a 32 byte block size, 6 cycle latency |
| L1 Instruction Cache | 16KB 2-way set associative, single port with a 32 byte block size, 4 cycle latency | 2KB 2-way set associative, single port with a 32 byte block size, 2 cycle latency | 16KB 2-way set associative, single port with a 32 byte block size, 5 cycle latency |
| Value Predictor (1 prediction per cycle) | 2K-entry stride<br>8K-entry Markov | none | 2K-entry stride<br>8K-entry L2 Markov |
| Address Predictor (1 prediction per cycle) | 2K-entry stride<br>4K-entry Markov | none | 2K-entry stride<br>4K-entry Markov |
| Stream Buffer | 32-entry FA buffer | none | 32-entry FA buffer |
| Branch Misprediction | 31 cycles | 27 cycles | |
| Core Width | 6-way issue, 3-way decode, 3-way commit | | |
| Memory and L2 Cache | 120 cycle memory latency, 512KB 4-way set associative unified (instruction and data) cache with a 64 byte block size and latency for Monolithic and $\mu$core : 20, 22 cycle | | |
| Functional Units | 3 integer ALUs, 1 integer MULT/DIV, 1 FP ALU, 1 FP MULT/DIV, 2 load/store | | |

Table 2.2: Simulation parameters for the monolithic and $\mu$core architectures

| | L1 DCache | L1 ICache | L2 Cache | Register File | BBTB | Branch Misprediction |
|---|---|---|---|---|---|---|
| Baseline | 4 cycle | 4 cycle | 20 cycle | 4 cycle | 3 cycle | 31 cycle |
| Optimized | 6 cycle | 5 cycle | 25 cycle | 7 cycle | 5 cycle | 37 cycle |

Table 2.3: Latency parameters for the monolithic and $\mu$-core architectures

| | IPC | Dcachee Miss Rate | Icache Miss Rate | BBTB Miss Rate | BBTB Mispredict Rate |
|---|---|---|---|---|---|
| applu | 2.00 | 0.15 | 0.01 | 0.00 | 0.02 |
| apsi | 0.96 | 0.17 | 0.02 | 0.00 | 0.07 |
| art | 0.91 | 0.39 | 0.00 | 0.00 | 0.06 |
| bzip | 1.43 | 0.03 | 0.00 | 0.00 | 0.05 |
| crafty | 0.72 | 0.02 | 0.06 | 0.08 | 0.07 |
| eon | 1.03 | 0.01 | 0.04 | 0.04 | 0.04 |
| galgel | 1.83 | 0.16 | 0.00 | 0.00 | 0.05 |
| gap | 1.35 | 0.03 | 0.01 | 0.01 | 0.06 |
| mcf | 0.62 | 0.42 | 0.00 | 0.00 | 0.05 |
| mesa | 1.58 | 0.01 | 0.03 | 0.01 | 0.04 |
| parser | 0.83 | 0.04 | 0.00 | 0.00 | 0.06 |

Table 2.4: Statistics for the baseline monolithic architecture

| | IPC | DL0 Miss Rate | DL1 Miss Rate | IL0 Miss Rate | IL1 Miss Rate | BBTB0 Miss Rate | BBTB1 Miss Rate | BBTB Mispredict Rate |
|---|---|---|---|---|---|---|---|---|
| applu | 1.95 | 0.20 | 0.62 | 0.04 | 0.12 | 0.00 | 0.15 | 0.02 |
| apsi | 0.83 | 0.32 | 0.39 | 0.04 | 0.44 | 0.09 | 0.58 | 0.11 |
| art | 0.94 | 0.47 | 0.69 | 0.00 | 0.46 | 0.00 | 0.00 | 0.06 |
| bzip | 1.49 | 0.05 | 0.43 | 0.00 | 0.63 | 0.00 | 0.02 | 0.06 |
| crafty | 0.69 | 0.18 | 0.11 | 0.08 | 0.72 | 0.28 | 0.29 | 0.10 |
| eon | 0.97 | 0.07 | 0.06 | 0.07 | 0.55 | 0.22 | 0.20 | 0.07 |
| galgel | 1.90 | 0.29 | 0.48 | 0.00 | 0.92 | 0.00 | 0.00 | 0.05 |
| gap | 1.34 | 0.04 | 0.25 | 0.04 | 0.46 | 0.19 | 0.05 | 0.07 |
| mcf | 0.67 | 0.44 | 0.85 | 0.00 | 0.59 | 0.00 | 0.00 | 0.05 |
| mesa | 1.46 | 0.11 | 0.03 | 0.09 | 0.28 | 0.23 | 0.05 | 0.07 |
| parser | 0.88 | 0.09 | 0.40 | 0.02 | 0.04 | 0.02 | 0.05 | 0.07 |

Table 2.5: Statistics for the $\mu$-core architecture with all helper engines enabled

# CHAPTER 3

# Helper management

While the unceasing march of Moore's law has given computer architects a continually increasing number of transistors to design with, emerging technology trends including poor wire latency scaling, increased power density, and reduced transistor reliability threaten to limit the usefulness of those designs. Chip Multi-processing based designs [HNO97] address some of these scaling constraints by statically partitioning processor resources on the die into different cores that can exploit available parallelism in a more power-efficient manner.

Researchers have also proposed decoupling the large memory-laden structures that are crucial to single thread performance from the critical processor loop and redesigning them as specialized *helpers* [KS02, KGM97, RAC99, BDA01]. The essence of these helpers is that they are latency tolerant. They can provide an added performance advantage without impacting core scalability.

In some ways, helpers are analogous to power grid architectures that supply major urban areas with electricity. While coal or nuclear power plants typically supply the majority of power needs, during periods of peak power demand, auxiliary power plants (often powered by natural gas) are used to meet temporary load requirements. In the context of microarchitecture, we have a simple processor pipeline that can be augmented by auxiliary helpers when the need for a particular speculative technique is high. In our analogy to the power grid, the presumed goal of utility companies is to avoid power outages that can adversely impact normal electrical consumption patterns.

Similarly, as microarchitecture desingers, we would like to optimize performance by finding a way to determine when helpers are needed for our processor pipeline so that they can be efficintly managed. This is achieved by taking advantage of the dynamic reconfigurability or polymorphism of helpers and allowing a core to adapt to changing applications, workloads, or phases. This is particularly critical in embedded devices designed for general purpose use. Here, die area can be efficiently utilized by sharing helper engines among multiple cores [KJT04].

In order to resolve these optimization problems, several questions must first be addressed - which resources should be shared; how should they be allocated; how can we efficiently manage their power? To answer these questions at runtime, we need a set of shared helper management policies that can adaptively allocate resources in a way that takes into consideration the needs of each executing workload.

In this chapter, we present novel rule-oriented shared-helper management policies that are empirical based. We show that through the application of metrics based on a few simple-to-gather run-time statistics, processor resources can be allocated to a set of running programs in a way that is nearly optimal.

We begin by examining a single-threaded core with a variety of helpers. Our metrics predict which resources will be most valuable to an executing program so that inefficient helpers may be put into a low power state without impacting overall performance. By using a statistic as simple as the number of cache hits, we can pick the best overall configuration in a single try. Furthermore, we show how these techniques can be extended to help guide the sharing of resources between multiple cores on a chip. We observe how different helpers have different sharing requirements and demands that result in that prior work may not be able to find the best sharing combination. We have implemented a variety of different helpers and extended our simulator to support multiple executing processes so that a detailed treatment of the subject can be

presented.

This chapter makes the following contributions:

- Analysis of program behavior and resource requirements in an architecture with many major components decoupled via helpers.
- Proposal of an intelligent mechanism to dynamically tune helpers to the specific needs of the application for optimal power/performance.
- Extension of this mechanism to intelligent sharing of resources among multiple cores. Prior work has only considered whether or not simple sharing is possible. We demonstrate how different types of helpers may be shared differently and how sharing must be *flexible* to attain maximal performance.
- Investigation of constructive sharing when the same application is allowed to share fetch state across different phases of execution and different input sets.

The rest of this chapter is organized as follows. In Section 3.1 prior work on dynamic adaptation and CMP sharing is discussed. Section 3.3 provides an analysis of the varying requirements for different applications for different helpers. We explore a mechanism to enable or disable helpers and further extend this to managing shared helpers in a multicore environment in Section 3.4.

## 3.1 Related Work

We are not the first to look at dynamically configuring resources during program execution. There are number of prior work that address this matter and can be devided to three main categories: Identifying and tracking program phase behavior, Dynamic adaptation to phases of execution and resource sharing among cores in a CMP setting. We discuss each category in more details.

### 3.1.1 Program Phase Behavior

To help us guide our power management scheme we make use of specialized hardware to detect when we need to perform a reconfiguration and to which reconfiguration we should switch. The work of Dhodapkar and Smith [DS02b, DS02a] aims to identify program phases in hardware using working set analysis. They found a relationship between phases and instruction working sets, and that phase changes occur when the working set changes. They propose that by detecting phases and phase changes, multi-configuration units can be re-configured in response to these phase changes. They have used their working set analysis for instruction cache, data cache and branch predictor re-configuration to save energy [DS02b, DS02a]. Our power managment scheme uses the phase finding technique of Sherwood et. al [SSC03]. In [SSC03], Sherwood et al. propose another online phase tracking architecture based on a hardware approximation of the techniques described in [SPH02]. They also present a prediction architecture that determines when and to which phase the next phase transition will occur. Yet another scheme is described in [HRJ03]. In this paper Huang et al. detail their phase detection technique that uses subroutines to identify the phases. They analyze the call stack to track the time spent in a subroutine and its callees and if this hierarchical time exceeds a threshold the subroutine is identified as a major phase. Our power management scheme builds upon the phase detection techniques above, but additionally with our configuration predictor, we can accurately determine which of the $pow(2,n)$ configuration to switch to rather than by using trial and error. These different phase detection techniques are compared in [DS03] using different metrics to determine the merits of each scheme in capturing phase behavior, and we refer the reader to this for additional information on hardware phase detection.

In [SPC01], Sherwood et al. proposed that by profiling only the code that was executed over time we could automatically identify periodic and phase behavior in pro-

grams. The goal was to automatically find the repeating patterns observed in [SC99], and the lengths (periods) of these patterns. This work was extended in [SPH02], using techniques from machine learning to break the complete execution of the program into phases (clusters) by only tracking the code executed. They found that intervals of execution grouped into the same phase had similar behavior across all the architecture metrics examined. From this analysis, they created a tool called SimPoint [SPH02], which automatically identifies a small set of intervals of execution (simulation points) in a program to perform architecture simulations. These simulation points provide an accurate and efficient representation of the complete execution of the program.

Merten et al. [MTB01] developed a run-time system for dynamically optimizing frequently executed code. Then in [BNM02], Barnes et al. extend this idea to perform phase-directed complier optimizations. The main idea is the creation of optimized code "packages" that are targeted towards a given phase, with the goal of execution staying within the package for that phase. Barnes et al. concentrate primarily on the compiler techniques needed to make phase-directed compiler optimizations a reality, and do not examine the mechanics of hardware phase detection and classification. We believe that using the techniques in [BNM02] in conjunction with our phase classification and prediction architecture will provide a powerful run-time execution environment.

### 3.1.2 Dynamic Adaptation

There is a large amount of work in computer architecture on the dynamic adaption of various structures. In a modern processor, a significant amount of energy is consumed by the data cache, yet this energy may not be sufficiently utilized if an application is not accessing large amounts of data with high locality. To address this potential inefficiency, previous work has examined the potential of dynamically reconfiguring the data caches with the intention of saving power. In [BAB00], Balasubramonian et

al. present two different schemes with which re-configuration may be guided. In one scheme, hardware performance counters are read by re-configuration software every hundred thousand cycles. This configuration management is based on miss rate and the number of branches in this fixed interval.

The software then makes a decision based on the values of the counters. In another scheme, re-configuration decisions are performed on procedure boundaries instead of at fixed intervals. To reduce the overhead of re-configuration, software to trigger re-configuration is only placed before procedures that account for more than a certain percentage of execution.

A different approach to re-configurable cache that has been proposed dynamically divides the data cache into multiple partitions, each of which can be used for a different function such as instruction reuse buffers, value predictors, etc [RAJ00]. These techniques can be triggered at different points in program execution including procedure boundaries and fixed intervals.

The overhead of re-configuration can be quite large and making these policy decisions only when the large scale program behavior changes, as indicated by phase shifts in our hardware tracker, can minimize overhead while guaranteeing adequate sensitivity to attain maximum benefit.

Another way to reduce the energy consumption in a processor through adaptation, is to reduce the number of instructions entering the pipeline every cycle [IM01, AGG03]. We call this adjusting the width of the processor. Reducing the width of the processor reduces the demand on the fetch, decode, functional units, and issue logic.

Certain phases can have a high degree of instruction level parallelism, whereas other phases have a very low degree. We can potentially save energy without hurting performance by throttling back the width of the processor for phases that have low IPC, while still using aggressive widths for phases with high IPC.

### 3.1.3 Resource Sharing

Sharing some processor resources among cores in a CMP setting was first proposed by Dolbeau and Seznec [DS02c]. Dolbeau and Seznec [DS02c] propose the CASH architecture as an intermediate design point between CMP and SMT architectures. A typical CASH architecture shares caches, branch predictors, and division functional units between dynamically-scheduled cores.

Kumar et al. [KJR03, KTR04] introduce a new architecture, that of a heterogeneous set of cores on a single multi-core die, sharing the same ISA. They examine a single application switching among cores to optimize some function of energy and performance.

Kumar et al. [KJT04] also propose resource sharing between adjacent cores of a chip multiprocessor to reduce die area with minimal impact on performance. They investigate the possible sharing of floating-point units, crossbar ports, instruction caches, and data caches and provide detailed analysis of area savings that each kind of sharing entails. Both [DS02c] and [KJT04] examine a round-robin based access model where a resource is allocated to a particular core every cycle. Kumar et al. also investigate a more sophisticated scheme for caches. After suffering a cache miss, a core relinquishes the control of the cache to the other core until the miss is serviced.

The TRIPS [SNL03] architecture uses large, coarse-grained processing cores to achieve high performance on single-threaded applications with high ILP, and allows the core to be subdivided for explicitly concurrent applications at different granularities. Contrary to conventional large-core designs with centralized components that are difficult to scale, the TRIPS architecture is heavily partitioned to avoid large centralized structures and long wire runs. These partitioned computation and memory elements are connected by point-to-point communication channels that are exposed to software schedulers for optimization.

Our work differs from these techniques in two dimensions:

- In addition to reactive (or demand-based) resource sharing such as when sharing caches, we also consider sharing always active resources such as prefetchers and value predictors that run ahead of the execution stream.
- We investigate an intelligent approach to assigning resources to cores based on utilization.

At a high level, our design shares much motivation with Simultaneous Multithreading. The Simultaneous Multithreading [TEL95] execution model assumes the ability of a superscalar architecture to issue instructions from multiple threads to the functional units each cycle, with the presumption that there are under-utilized resources available. In our architecture, we separate these resources from the core so that we can exploit them for power management purposes, rather than allowing multiple threads to access them. There is an interesting relationship between SMT and the $\mu$-core design and in fact we believe that the optimal solution will use both techniques. However, we leave this combination for future work.

## 3.2 CMP Methodology

We use the same methodology described in chapter 2. In addition to modeling all of the structures and latencies in the architecture, we have extended SimpleScalar to include a cycle accurate, execution driven model of chip multiprocessing (CMP) [HNO97]. All the parameters used in our multicore experiments are the same as in Table 2.2 for each core, except that we increase the size of the L2 cache to an 4MB, 4-way set-associative cache shared among all cores.

Per-thread performance metrics are measured for execution up to a maximum per-

thread instruction count. All completed threads continue execution past this point while other threads execute. This prevents freeing of resources when certain threads complete earlier than others.

We make use of *weighted speedup* [ST00] as a performance measure. This metric ensures that high IPC threads are not favored and that we are measuring real increases in the rate of progress of all applications in the mix. Weighted speedup equalizes the contribution of each thread to the sum of total work completed in the interval by dividing the IPC of that job in the mix by the IPC of a single threaded run.

## 3.3 Tuning a Single Core

As our goal is to develop techniques to make good choices between a variety of different run-time processor configurations, we need to begin with a processor model that has been highly decoupled. We use the decoupled architecture proposed in chapter 2 (figure 2.3).

In this section we limit ourselves to single core designs, and then in Section 3.4 we show how to extend the ideas developed here to apply to cases where multiple cores may compete for resources.

### 3.3.1 Helper Utilization

While there are many circuit level advantages to decoupling large structures from the processor core, it also makes it very easy to *tune the processor to the specific needs of an application*. Each of these helpers has a well defined interface, and the processor can tolerate a wide range of access latencies to any one of these helpers (further explored in Section 3.4). Thus, adding supplemental control to each of these devices that will allow them to maintain separate power states does not introduce unreasonable

cost or performance degradation.

As might be expected, for any given program some helpers will be more helpful than others. If the program is spending all of its time doing data accesses, it is more likely to get benefit from the data cache and prefetcher than the instruction cache. In a naive design, one might leave all of the helpers in an "on" state at all times. Clearly, this will be wasteful if a program gets no benefit from a subset of the helpers. In fact, we have found that up to half of the helpers can be turned off at any given point in time, with almost no performance impact (2%). The problem is knowing *which* helpers to turn off.

Note that there is an overhead associated with power gating, and the coarse-grain organization of these auxiliary structures into helpers provides a useful abstraction for power gating these structures. Moreover, phase-based optimization helps to hide the latency associated with power gating, since the granularity of our helper allocations is much larger than the latency of power gating.

To further illustrate this, we plotted the set of critical helpers that are needed to maintain top performance for each program in Figure 3.1. Figure 3.1 shows the minimal set of helpers needed to achieve maximal performance for the top four phases of 12 of the programs we have examined (shown in the rows). The columns of the table represent the helpers, (d=data cache, p=prefetching, b=branch prediction, i=instruction cache, and v=value prediction). An X in a given square indicates for that phase, the corresponding helper should be turned "on". Helpers without X's can be disabled without affecting performance. We refer to setting the helpers "on" or "off" as the configuration of the helpers in one of two power states.

All of the configurations in Figure 3.1 perform within 5% of the configuration shown to have the absolute highest performance (typically the configuration with all helpers active). The last two columns present the speed up of this configuration relative

| | | d | p | b | l | v | S U /all | S U /none |
|---|---|---|---|---|---|---|---|---|
| applu | 1 | | X | | | | -4% | 9% |
| | 2 | | | | | | -2% | 0% |
| | 3 | | X | | | | -4% | 20% |
| | 4 | | X | | X | | 0% | 19% |
| apsi | 1 | | | X | X | | -3% | 40% |
| | 2 | | | | | | 0% | 2% |
| | 3 | | | | | | -4% | 0% |
| | 4 | | | | | | -4% | 0% |
| art | 1 | | X | | | X | 0% | 16% |
| | 2 | | X | | | | -4% | 8% |
| | 3 | | | | | X | -4% | 7% |
| | 4 | | X | | | | -4% | 8% |
| bzip2 | 1 | | X | | | X | 0% | 23% |
| | 2 | | | | | X | -2% | 8% |
| | 3 | | X | | | X | 0% | 23% |
| | 4 | | X | | | X | -3% | 10% |
| crafty | 1 | X | | X | X | | -4% | 174% |
| | 2 | X | | X | X | | -3% | 138% |
| | 3 | X | | X | X | | -2% | 192% |
| | 4 | X | | X | X | | -4% | 156% |
| eon | 1 | X | | X | X | | -2% | 125% |
| | 2 | X | | X | X | | -1% | 110% |
| | 3 | X | | X | X | | -2% | 107% |
| | 4 | X | | X | X | | -1% | 105% |

| | | d | p | b | l | v | S U /all | S U /none |
|---|---|---|---|---|---|---|---|---|
| equake | 1 | | X | | | | -1% | 79% |
| | 2 | | | X | | X | -1% | 122% |
| | 3 | | | | | X | -3% | 3% |
| | 4 | | | X | | X | -2% | 112% |
| galgel | 1 | | | | | | -4% | 0% |
| | 2 | | | | | | -1% | 0% |
| | 3 | | | | | | -1% | 0% |
| | 4 | | | | | | -3% | 0% |
| gap | 1 | | | X | X | X | -3% | 174% |
| | 2 | | | X | X | | -4% | 40% |
| | 3 | | | X | X | | -4% | 36% |
| | 4 | | X | | | X | 0% | 85% |
| mcf | 1 | | X | | | X | -1% | 985% |
| | 2 | | | | | X | -2% | 15% |
| | 3 | | | | | X | -3% | 407% |
| | 4 | | | | | X | -2% | 10% |
| mesa | 1 | | | X | X | X | -1% | 109% |
| | 2 | | | X | X | X | -1% | 94% |
| | 3 | | | X | X | X | -1% | 101% |
| | 4 | | | X | X | X | -1% | 102% |
| parser | 1 | X | | X | | X | -2% | 29% |
| | 2 | X | | X | | X | -1% | 24% |
| | 3 | X | | X | | | -4% | 17% |
| | 4 | X | | X | | X | -1% | 18% |

Figure 3.1: Helper configurations that fall within 5% of the maximal performance and have a minimal number of helpers active. An X indicates that the helper is “on”.

to best and worst configurations (all helpers on, and all helpers off). These configurations were found strictly by a brute force search of the design space, simulating every possible configuration and taking the configuration with the least number of helpers "on" that was still within 5% of the case with all helpers turned on.

Obviously, trying each of the $2^n$ possible configurations is not a viable design choice for a runtime system, but there are several important points that can be drawn from this graph. It is clear from Figure 3.1 that there is no *one* good configuration that fits all applications. For many programs different configurations are even needed for different phases. This means a new, intelligent, and adaptive management scheme is going to be needed.

### 3.3.2 Helper Configuration

Helper configuration could be tackled in a variety of different ways. While static approaches could use profile or compiler guided heuristics to find one good configuration for the application as a whole, this may prove ineffective due to the time varying behavior of applications. A more effective static approach may be to inject special reconfiguration instructions that help tune the processor to specific phases. While these techniques are possible, in this work we choose to focus on hardware based dynamic techniques as they require no a priori knowledge of the program and operate in a completely on-line manner.

To guide our decision about which configuration to choose, we need some information on how a program interacts with the processor. This information must be simple and cheap to obtain, and should be highly indicative of the benefits the program is reaping from access to each helper. In order to gather this information, we use simple performance counters that track the "help" that each helper provides. For example, performance counters can track the number of hits in cache helpers, successful predic-

tions by predictor helpers, and so forth. Specifically, our helpers are modified to track the following events:

**Data/Instruction Cache:** the number of times a cache line that missed in the L0 data/instruction cache hits in the data/instruction cache helper.

**BBTB:** the number of times a PC that missed in the L0 BBTB hits in the BBTB helper.

**Data Prefetcher:** the total number of hits in the stream buffers and in the L2 cache that were brought in by the prefetcher. Each line in the L2 cache is augmented with a bit to indicate whether it was brought in by a demand miss or a prefetch. On the first use of a line marked as a prefetch, the bit is flipped.

**Value Prediction:** the number of instructions issued using a predicted value.

These counter values are compared to pre-determined threshold values to decide whether the helper engine should be turned on or off. We performed a sensitivity analysis to various threshold values for the SPEC 2000 benchmark suite with reference inputs. We show results for a conservative set of thresholds to curtail performance degradation. More aggressive thresholds can be used when power reduction is the primary objective. Furthermore, dynamic threshold values can be calculated by adjusting the thresholds to maintain certain "on" and "off" state performance counter standard deviation values. However, this is beyond the scope of this work, and we leave this for future work.

We make use of phase-based memoization [SSC03] to track the helper configuration per application phase. A small hardware structure tracks a bit vector per phase for all the helpers in the architecture. If the bit at a particular location is set, the helper represented by that particular location should be on. Otherwise the helper can be turned off. The first time a phase is seen, we turn all helpers on and track the performance

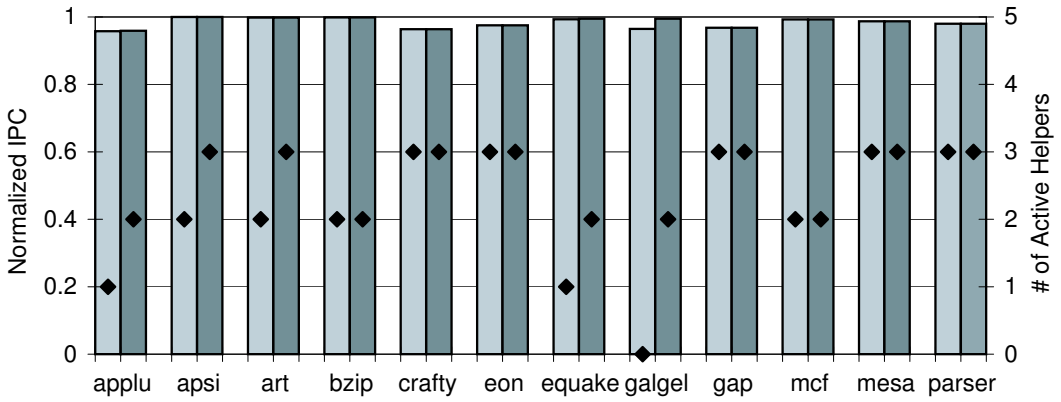


Figure 3.2: Comparison of the configuration from Figure 3.1 (light grey) and our counter guided configuration (dark grey).

using the above counters. Each counter is compared against the threshold for keeping the helper on, and the bit vector for that particular phase is updated in our phase-based memoization table. The helpers can then be guided by a simple last phase predictor. The sampling period need not be as long as a phase, and can be limited to an interval of one million cycles.

We also memoize the observed IPC for each phase during the sampling period, and if the IPC resulting from a particular configuration is not within some threshold IPC seen during sampling (5% for this study), we clear the bit vector for that phase and force another sampling of the above counters. This helps recover from phase mispredictions or events that can impact performance (such as power throttling).

### 3.3.3 Performance of Counter-Guided Configuration

Figure 3.2 compares the performance of the configuration from Figure 3.1 (light grey) to our counter-guided configuration. Performance here is normalized to an architecture running with all helpers on. On average, we are able to come within 1.5% of

that performance with an average of only 2.6 helpers turned on. The configuration from Figure 3.1 comes within 2% of the performance of all helpers on, only using 2.2 helpers on average. In some cases, we use one more helper than needed due to our choice of conservative thresholds. More often than not this is the data cache helper, which can provide load hits that are not performance critical. One approach to fine tuning this further would be to incorporate the notion of load criticality [SJL01] or to try and correlate this with the number of L2 misses – applications with a large number of L2 misses may not see benefit from an increase in L1 hits if the L2 misses that dominate the critical path of the application are not reduced.

These results demonstrate the ability of performance counters to fine tune helper engine utilization. This can allow an architecture to reduce power wasted in helpers that do not provide useful work or as we will see in the next section, to coordinate sharing among cores using a common pool of helpers.

## 3.4 Management Across Cores

As demonstrated in Section 3.3, resource demand varies significantly across different applications and even across different phases of the same application. In the case of a single core, this fact can be exploited to reduce power by finding a configuration that still provides good performance but with a bare minimum of helpers left in a high power state. Managing helpers when multiple cores are involved presents a tougher challenge. While the easiest approach to supporting multiple cores on a chip would be to give each one its own set of helpers, previous work has shown that this instills unnecessary area complexity without a significant performance benefit [DS02c, KTR04]. The alternative that we also explore in this work is the use of a common pool of helpers, shared among all the cores. Sharing has a number of benefits such as reducing the area spent to implement redundant functionality and the potential of optimization through

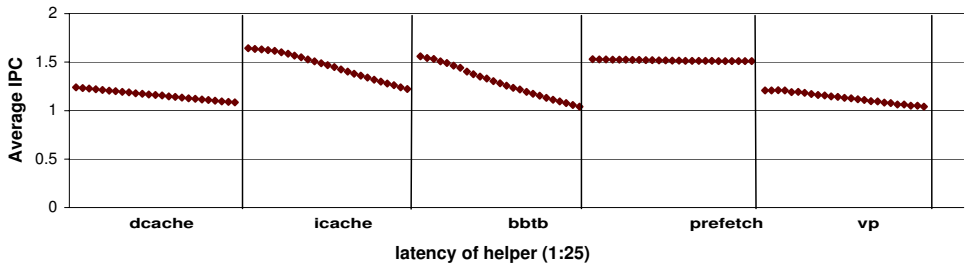


Figure 3.3: The X axis shows the performance impact of increasing the latency of each helper from 1 cycle to 25 cycles.

dynamic resource allocation. In this section, we present techniques to effectively manage helpers in a multicore environment.

### 3.4.1 Design Decisions

There are a large number of design decisions to be made in examining helpers in a multicore setting: the number of cores, the number of each type of helper, the topology of the interconnect between cores and helpers, the physical layout of cores and helpers, and the application mix to execute on the cores. This is an enormous design space, and it is simply not manageable to try all possible combinations. To get a set of experiments which is tractable, we limit our search in this work by considering results for three possible machine organizations: two cores sharing a single helper of each type, four cores sharing a single helper of each type and four cores sharing two helpers of each type. Additionally, we consider applications that are not cooperative, but our work could certainly be applied to cooperative multithreading. We assume that all cores share a common second level cache, and that any core may connect to any helper.

#### 3.4.1.1 Helper Latency Tolerance

One question that immediately comes to mind when we propose that any core may connect to any helper is that it is going to take a good deal longer to communicate with a helper on the other side of the chip than with one in close physical proximity to the core. While this is true when partitioning arbitrary processor resources, helpers have an inherent advantage due to their higher latency tolerance.

To demonstrate that helpers are naturally latency tolerant, we present Figure 3.3 which plots the performance impact of latency on our various helpers by varying the access latency from 1-25 cycles. For these results, we consider a single core with private helpers, and average the IPC observed over benchmarks that use the helper from table 2.8. As seen from the figure, for most helpers, there is little impact on performance of these helpers from smaller latencies – the most is seen by the branch target buffer (BBTB) helper, which suffers an IPC degradation around 1% for each additional cycle of latency. Prefetching sees the least impact, less than a 0.01% drop in IPC for each additional cycle of latency. The prefetcher hides the latency of memory, and even 25 cycles is tolerable when compared with this latency. The remaining helpers see less than 0.5% degradation per cycle for each additional cycle. As the latency increases above 10 cycles, its impact on the performance of the helper increases non-linearly.

Our architecture is also not impacted by nonuniform access latency from different cores to a common helper. The arbiter that selects what requests from a core should be serviced by a helper would be located close to the helper itself. Therefore, if core A sees a two cycle latency to a helper and core B sees a single cycle latency to a helper, and if core A pipelines its requests over two cycles, then the helper will simply see a stream of requests from A and B without any notion of heterogeneous latency. A will see its predictions a cycle later than B will see its predictions, but as we have demonstrated, this has a negligible impact on performance.

Our multicore architecture and its floorplan are shown in Figure 3.4. We use CACTI to estimate the size and dimensions of the L2 cache and all of our helpers. The area of our core was calculated using the area of EV6 and EV5 scaled to a 70nm feature size, similar to [KJR03].

Our flexible sharing requires a link from each core to each helper. This would double the number of interconnections compared to a conjoined architecture where each helper is statically shared between two cores. We used a similar method of crossbar area estimation as [KTR04]. For our choice of helpers and their respective bandwidth requirements, the crossbar area occupies 10% of the total area of the processor, which is 5% more than conjoined cores.

The other hidden cost in sharing helpers is the potential increase in requests for each resource, making helper bandwidth a serious concern. Ideally, each core would have its own dedicated port to each helper, but the cost would be prohibitive. Instead, a helper can make use of port arbitration to satisfy multiple core requests. One possibility is allowing cores to take turns accessing a helper. Another would involve more sophisticated control hardware that would arbitrate among several requests, much like what is done with a unified second level cache.

In an architecture where cores share a common pool of helpers, the helpers can either be exclusively assigned to a core (i.e. partition the helpers), shared among several cores (i.e. sharing common helpers), or some combination of both (i.e. some of the cores sharing a common helper). Partitioning is useful when the bandwidth or internal storage demands placed on a helper by a single core would preclude benign sharing with other cores. In that case, the most favorable option would be dedicating a helper to a single core. Sharing can be useful when individual cores do not consume all of the bandwidth or storage space of a given helper.

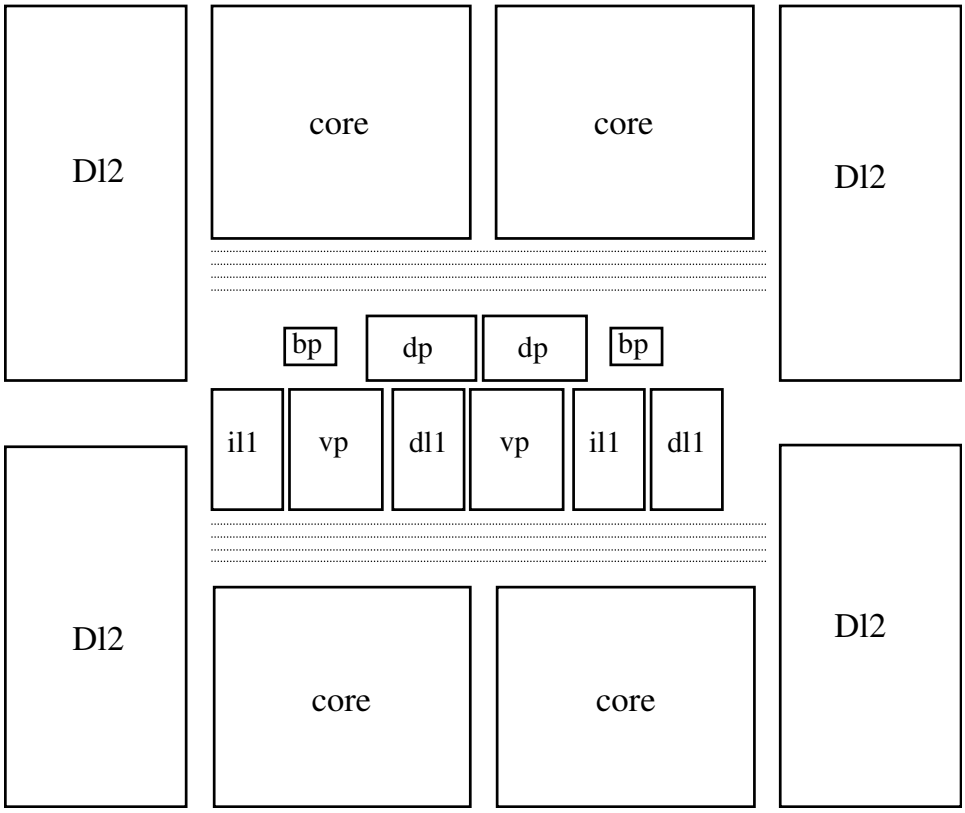


Figure 3.4: Our CMP floorplan, with cores and L2 cache banks distributed around 2 sets of helpers.

#### 3.4.1.2 Always On vs. On Demand Sharing

Different helpers exhibit different tolerances to sharing. At a high level, helpers can be divided into those that are accessed on-demand and those that are always active. On-demand helpers include those that are hierarchical extensions of core structures, like the instruction cache, data cache, and branch predictor. These helpers are only accessed when their corresponding core structure misses. Locality in the corresponding core structures filters the majority of the requests obviating frequent accesses to these helpers. This means that bandwidth to the helper may more easily be shared among multiple cores. However, the amount of state contained in the helper may still be insufficient to allow effective sharing between cores – but bandwidth is not usually a limiting factor.

Other helpers do not have corresponding core structures, and are therefore not on-demand. The value prediction and prefetching helpers are examples of this class of helpers. Any load instruction can be value predicted and any cache miss can initiate a prefetch stream, but a helper may not have enough bandwidth to handle competing requests from multiple cores if there are lots of loads or cache misses. Helper state is also a problem here, as sharing cores would need to contend with one another for value and address predictor space.

#### 3.4.1.3 Simple vs. Counter Guided Sharing

One simple approach to sharing is to have conjoined cores take turns accessing a common set of helpers. Taking turns is a viable option for on-demand helpers where contention for helper bandwidth is less common. But helpers that are not on-demand can suffer from a turn-based approach. Consider value prediction in a two core setting, where each cycle a new set of load PCs are fetched and could potentially be value pre-

dicted. Taking turns would mean that value prediction would only occur every other cycle, potentially halving the number of value predicted instructions. Note that the issue here is not access latency (because helpers are latency tolerant), but the fact that we lose the opportunity to predict load instructions. This problem only becomes worse when trying to share among more than two cores. We further optimize the turn-based strategy by allowing other cores to access a helper on a given core's turn if that core does not require the helper's bandwidth. This can happen when a core has suffered a pipeline flush or in the case of on-demand helpers that simply do not see any accesses to the helper.

Another way is to grant the first two requests to the value predictor, regardless of which core makes them. The advantage of this over taking turns is that if one core does not have two loads to predict in one cycle, it allows the other core to use the bandwidth. We use this latter approach in our simulations.

Even though some cores may make use of a helper, the helper may not provide any benefit, even in the case of on-demand helpers. Consider a thread whose working set does not fit in the data cache helper – it will thrash in the data cache as it tries to contain its working set, but will still be plagued by misses that must be serviced by the L2 and will evict potentially useful entries from other threads. Similarly, a thread may not see any benefit from value prediction and may simply be stealing available bandwidth from a thread that does see benefit.

We make use of the utilization counters from Section 3.3 to guide helper sharing. These counters provide a filtering mechanism to avoid sharing a helper among cores that see no benefit from that helper. Cores can then request access from a global helper arbiter to the helpers from which they expect to see benefit.

While filtering useless sharing is vital, we also need to provide an intelligent approach to choose what cores will share a common helper. In a four core scenario where

all cores (labeled A-D) want a BBTB helper, and there are only two helpers available, performance may substantially improve if A and B are allowed to share instead of A and C. Or, it may be best for A to have its own private helper and for B-D to share the remaining helper.

To arbitrate sharing among the filtered set of helpers, we make use of correlating counters that can guide sharing. A good example of this is the prefetching helper. If multiple cores are contending for a pool of prefetch helpers, threads that have a greater magnitude of prefetches should be given private access to a prefetcher if one is available. If there are not enough prefetchers to grant a private to threads with a large number of prefetches, threads with a comparable number of prefetches should be paired together. This prevents threads with a greater number of prefetches from starving other threads from getting access to stream buffers.

At each core, the correlating counters are tested against a number of thresholds to classify the demand for a particular helper into a utilization class (i.e. light, medium, heavy). The utilization class is then memoized in the phase-detection hardware to track the expected utilization class of each helper at each phase.

The global helper arbiter is responsible for taking the requests from all cores for helpers (this has already been filtered at each core by the performance counters and phase-detection mechanisms) and the actual utilization class for each helper requested from each core. The global arbiter then tries to match requests of similar utilization classes together in the case of helpers that are not on-demand and tries to mix requests in the case of helpers that are on-demand. The global arbiter makes decisions about assignments of helpers to cores every 10 million cycles, the same granularity as our phase detection intervals. This granularity of decision making can easily absorb both the latency to communicate the counter values to the arbiter and the latency for the arbiter to make a decision based on these counters.

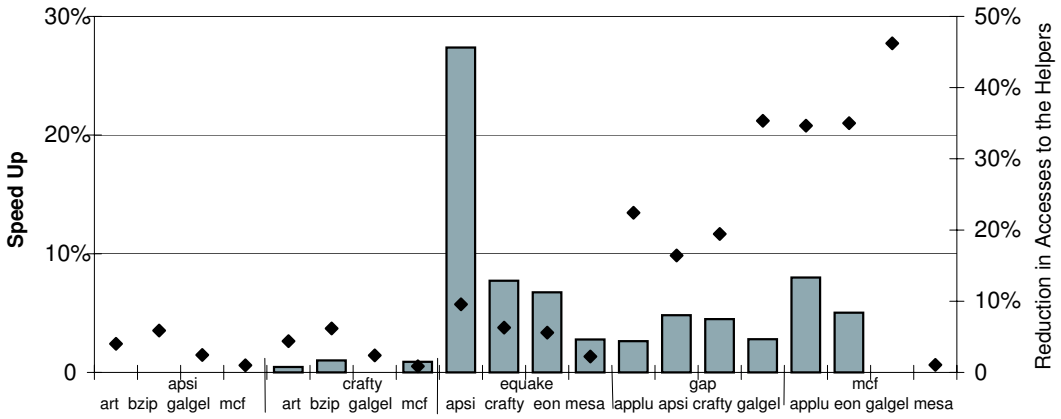


Figure 3.5: Impact of filtering useless accesses to a single set of helpers shared by two cores. Results shown are relative to conjoined sharing without filtering

We will refer to the architecture that can allow helper sharing between any set of cores on a CMP our flexible sharing architecture.

### 3.4.2 Sharing Results

Utilization counters can potentially improve multi-threaded workload in two ways: filtering useless sharing and providing information to choose the best pairing of applications to share a helper. To better evaluate the contribution of each of these factors we study both 2-core and 4-core CMPs sharing helpers. The former case can only see benefit from filtering useless accesses as there is only one way to share helpers. The latter case can take advantage of flexible sharing of helpers as well as filtering useless sharing.

#### 3.4.2.1 2-Core Results

Figure 3.5 compares the performance of our counter guided approach to a baseline that simply shares each helper among two cores. Utilization counters prevent applications from accessing the helper when they do not see benefit from that helper. Bars represent normalized IPC, and dots represent the reduction in the total number of accesses to the helpers. The application mixes are constructed to include threads with disjoint demand for helpers. For example, *apsi* and *crafty* heavily use instruction and branch prediction helpers (on-demand helpers) while *equake*, *gap*, and *mcf* see significant improvement using value prediction and prefetch helpers.

On-demand helpers do not see much improvement from counter guided approach because they already inherently filter requests as can be seen from small reduction in the number of accesses to the helper. However filtering useless accesses is essential to always-on helpers. *Equake* sees a 27% speedup just from eliminating useless accesses to the prefetch helper from *apsi*. The large reduction in accesses to the helpers for these mixes is another indication that the counter guided approach can improve performance and reduce power at the same time.

#### 3.4.2.2 4-Core Results

Next, we examine the benefits of flexible sharing. First, to better evaluate the difference between the two types of helpers (section 3.4.1.2), we more closely examine the BBTB helper (an on-demand helper) and the prefetcher helper (not on-demand). For each, we examine situations where four cores have private versions of all other helpers but share either the BBTB or the prefetcher. The four cores are then forced to share either one or two instances of either the BBTB or the prefetcher. In this section, with the exception of our baseline architecture, simple sharing *always* uses counter-guided filtering to

reduce accesses to helpers. But cores may or may not be able to flexibly share helpers. In simple sharing schemes, cores may only share helpers with certain other cores.

We consider the following architectures:

**baseline** - Four cores share a single helper - this configuration does not use counter-guided filtering.

**allprivate** - All cores have their own helpers - no sharing is done.

**1helper** - Four cores share a single helper using counter-guided filtering.

**average, worst, best** - This configuration shares two sets of helpers among four cores. When using simple sharing, performance depends on what cores are actually sharing (i.e. conjoined). With four cores, there are three possible pairings. For simplicity, we show values for the best of the three configurations, the worst of the three, and the average over all three.

**flexible** - The flexible helper sharing policy from section 3.4.1.3. One core can share a prefetcher helper with another core, and a value predictor helper with a different core.

For each helper that we explore, we construct an application mix by selecting five benchmarks that benefited from the helper and five benchmarks that did not benefit from the helper. We then form five application mixes of four threads each that represent all possible combinations: all threads need the helper, three out of four threads need the helper, two out of four threads need the helper, only one thread needs the helper and no thread needs the helper – in that order. For cases where only one or no application demands a helper, there is obviously little impact from these sharing approaches – but these results are shown for completeness.

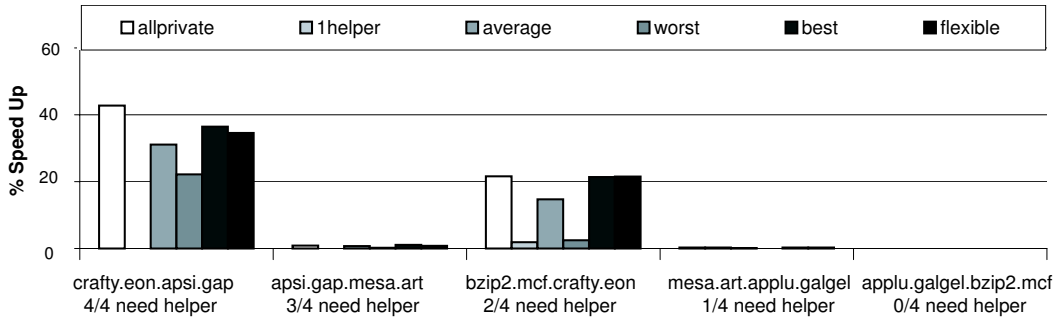


Figure 3.6: Simple vs Counter-Guided Sharing of the BBTB Helper

Our helper configurations are as presented in Section 2.3. The value predictor helper has only two ports, and can therefore only satisfy two requests for prediction per cycle. The prefetcher can only prefetch one cache line per cycle.

Figure 3.6 illustrates the performance of an on-demand helper: the BBTB helper. Results are shown for a weighted IPC speedup relative to the performance of four conjoined cores, sharing a single BBTB helper. The first bar shows results when all cores have private BBTB helpers. The second bar shows results for four cores sharing a single BBTB helper with our intelligent counter-based approach. The final four bars show results for four cores and two sets of BBTB helpers, all with counter-based filtering. The first three are the average, worst, and best cases for different combinations of conjoining cores into pairs of two sharing a BBTB helper. The final bar is our flexible approach, where all four cores can share two helpers in any way. For the single helper runs, there is not much improvement from counter-guided sharing because on-demand helpers naturally will not be accessed if there is no benefit. However, when sharing two helpers among four cores, there is a destructive relationship between *eon* and *crafty* when sharing a common BBTB due to aliasing. Our counter-guided approach is able to determine the best combination for benign sharing – even when all four applications want to share the BBTB. Because *eon* and *crafty* see many

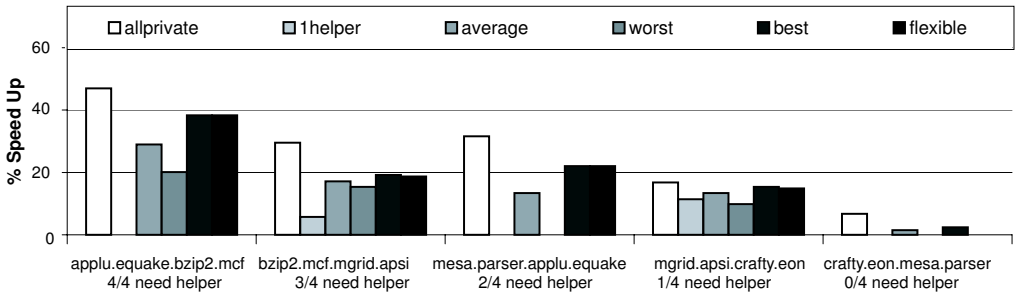


Figure 3.7: Simple vs Counter-Guided Sharing of the Prefetching Helper

more helper BBTB hits than other applications, they should not be combined together to avoid contention for space in the BBTB. The gap in performance between having a single BBTB helper and having two BBTB helpers is clearly greater when more applications demand the BBTB, demonstrating the impact that contention for space can have, even for on-demand helpers. The last two mixes of benchmarks, where only one or no application needs the BBTB, do not see any impact from sharing approaches.

Figure 3.7 illustrates the performance of a helper that is not on-demand: the prefetch helper. Counter-guided sharing is useful for the single helper run when there are two or three benchmarks competing for bandwidth that do not see any benefit from the helper. The benchmark mix *mesa-parser-applu-equake* is such a case, where *mesa* and *parser* do not benefit from prefetching but *applu* and *equake* do. By filtering *mesa* and *parser* out via our correlation counters, we are able to use a single helper to outperform the worst and average cases of conjoined cores with two helpers.

With two helpers, there is more disparity between different conjoined core runs. *Applu* and *mcf* issue far more prefetches than *bzip2* and *equake*, and unlike the case of on-demand helpers, it is actually beneficial here to have *applu* and *mcf* share the same helper. If we combine either one of these applications with one that benefits from prefetching but does not have the same magnitude of prefetches (like *bzip2* or

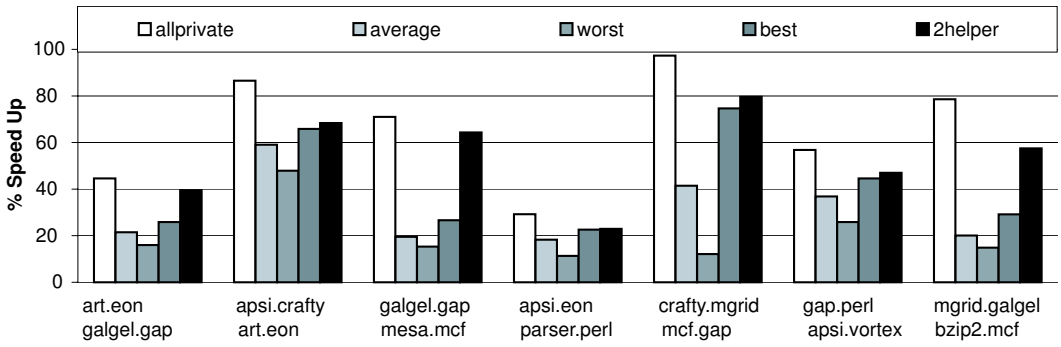


Figure 3.8: Simple vs Counter-Guided Sharing of All Helpers

*equake*), the application with less prefetches will not get a fair share of stream buffer resources. In the case of *mesa-parser-applu-equake*, *parser* contends for prefetch bandwidth, despite not seeing a benefit from prefetching, and impedes the prefetching of *applu* or *equake*. The counter-guided approach is able to identify the inability of *parser* to effectively use prefetching, and prevents this degradation. For all runs, our counter-guided approach is able to perform as well or better than the best conjoined core combination.

Figure 3.8 presents results for sharing all helpers at once. The first application mix on the figure enjoys a large improvement from our counter-guided approach for two main reasons. First, *gap* is able to get a private value predictor. Second, *art* and *eon* do not do well when conjoined with *galgel*. By giving flexibility to helpers in how they share, our approach is able to outperform any conjoined combination. This is also evident from the third benchmark mix, where value prediction contention hampers the performance of *mcf* and *gap*. On average our counter-guided approach for one set of helpers sees 13% improvement over a baseline naively sharing helpers among all cores. Sharing two set of helpers, our approach provides 54% improvement while conjoining cores can see benefit ranging from 20% to 40% depending on how cores are conjoined.

#### 3.4.2.3 Constructive Sharing

As we demonstrated in the previous section, sharing one helper among four cores can degrade performance significantly when applications running on all cores need the helper. However, there are cases where common code or data (i.e. OLTP, parallel processing) may be executing in the CMP environment, and sharing can actually be constructive if threads are allowed to share state in a common helper. Our flexible helper management allows constructive sharing of a helper among such workloads. We consider the case where the same application is executing on all cores, but each application instance is executing different input sets or different phases of the same input set.

Figure 3.9 illustrates the benefit of constructive sharing for the BBTB and instruction cache helpers. For *crafty*, *eon* and *mesa* we simulated four different phases of each application running concurrently on four cores. For *vortex* we used four different inputs. All of these application use both BBTB and instruction cache helpers intensively. The first bar shows results when there are private helpers dedicated to each core, and the second bar shows results when only one helper is shared constructively among the four cores. The speedup presented is relative to one helper shared among four cores, but without any constructive sharing among threads. Our counter mechanism is still used in this case. Our results indicate that sharing the instruction cache between multiple cores incurs a similar miss rate as a dedicated cache per core with the same capacity, as reported in [KAD04]. In *vortex*, a constructively shared instruction cache even outperforms the private cache performance by avoiding misses to cache blocks used by multiple cores. The contention when sharing the BBTB among applications that have high demand for this helper can be extremely severe when not shared constructively. *Crafty* and *vortex* see a 4X speedup when using dedicated BBTB helpers instead of sharing a single BBTB helper. This is due to the reduced accuracy of branch

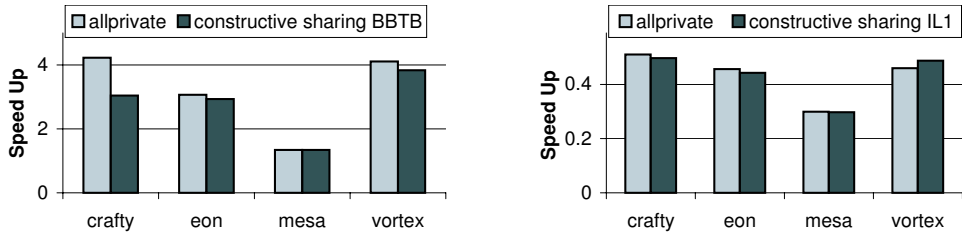


Figure 3.9: Speedup of private helpers and constructively shared helpers (BBTB (left) and Instruction Cache (right))

prediction when threads thrash for space in the shared predictor. There is a significant increase in the number of instructions executed as a result of this mispeculation which can further pollute caches and waste energy. Sharing the BBTB constructively among cores eliminates this thrashing effect for all applications simulated except *crafty* which still sees some impact from this. This application has more complex branch behavior that can inhibit constructive sharing across different phases.

#### 3.4.2.4 Power Savings from Flexible Sharing

To better understand the tradeoffs between sharing helpers and keeping private helpers for each core, we integrated Wattch [BTM00] into our simulation infrastructure and obtained energy results for our flexible sharing policy and the configuration where every core has a private helper. We used process parameters for a 70nm process at 5.6GHz with 1V supply voltage. Our results are extracted with the most aggressive conditional clocking strategy, where dynamic power scales linearly with access to the ports.

Figure 3.10 illustrates that on average, the flexible sharing policy is able to reduce the energy dissipated, despite the fact that the performance of the flexible policy can be slightly worse than the all private configuration.

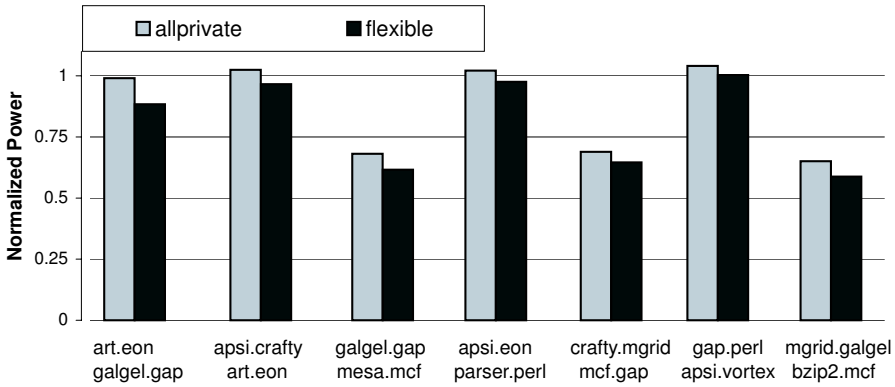


Figure 3.10: Power dissipated in flexible and allprivate configurations normalized to baseline configuration with shared helpers.

# CHAPTER 4

# Low-Overhead Core Swapping for Thermal Management

Thermal characteristics of contemporary processors are creating significant challenges to microprocessor design. Various trends threaten to make things even worse: the number of on-chip transistors is quickly approaching one billion, clock frequencies are dramatically increasing, feature sizes are dropping to deep submicron levels, and supply voltage reduction is expected to slow down as it approaches noise margin barriers. As a result, power densities and on-chip temperatures are expected to increase even faster for the next generation of processors.

Thermal issues have gained significant importance in the past few years. Processor heating raises problems that threaten vital aspects of the microprocessor design, such as proper functionality, reliability, cost, and performance.

The reliability of an electronic circuit is exponentially proportional to the junction temperature. A 10°C increase in temperature usually translates to an approximate 2x difference in the lifespan of the device [LR01]. At higher operating temperatures, the microprocessor operates at relatively lower speeds [VWW00].

Furthermore, temperatures are not constant across the chip. $30-40$°C thermal gradients are quite common, which causes potential timing and data errors [ABP01]. There is a non-linear relationship between cooling capabilities and the cost of a cooling solution. The cost of cooling increases at a higher (almost exponential) rate for higher

temperatures [GBC01].

In recent years, dynamic thermal management (DTM) [DM01, KMH03, WJS00, LDG02] has become an integral part of microprocessor design to adapt to increasing on-chip temperatures.

The disparity between the maximum possible power dissipation and typical power dissipation has become more pronounced. This, along with the exponential increase in cooling device costs, has created a new trend where cooling systems are designed for the typical worst case power dissipation instead of the maximum possible power dissipation. Therefore, dynamic thermal management has become essential to ensure that processor temperature does not reach or exceed the maximum tolerable temperature.

Many power optimization techniques do not seem to address problems caused by processor heating, as they are targeting relatively cooler parts of the chip, such as caches. With the expected increases in power consumption and temperature, there is no doubt that more DTM techniques specific to microprocessor designs are needed. DTMs usually target the removal of excessive heat from the processor after a certain temperature threshold is reached. Such a removal often causes performance degradation, as a result of reduced clock frequency or voltage, or from temporarily shutting down the entire chip. Therefore, efficient thermal management techniques with less impact on processor performance are extremely desirable.

Activity migration (i.e. core swapping) has been proposed as an efficient technique for power density reduction [HBA03], with subsequent studies [KTR04, Kan02, PGV04] further exploring the viability of this approach. While core swapping has the potential to drastically alleviate thermal problems, it can be plagued with two main drawbacks: 1) the latency overhead of performing a swap and 2) the area overhead of retaining additional resources to perform the swap. The latency overhead stems from both the time to propagate resources required for computation to migrate from one

core to another and the warmup time for various predictors that may not be migrated from one core to another.

In this chapter, we further explore the use of swapping an application between multiple cores when a given core exceeds a particular thermal threshold, specifically addressing both the latency and area overhead of swapping. Our cores feature a small, fast pipeline augmented with helper engines 1. All large structures are factored out of this *microcore* [KSS06] and are relocated as helper engines, taking advantage of locality in the first level structures. The helper engines buffer state during core swaps and help reduce the overhead of swapping. We compare this approach to current DTM techniques.

Our contributions include:

- We propose a thermally-triggered core-swapping technique as a DTM for dual-microcore architectures. Our architecture solves the switching overhead problem inherent to core swapping by buffering the state in helper engines. We compare the performance of core swapping with other DTMs such as global clock gating and dynamic frequency scaling.
- We study the area overhead of dual-microcore architecture and investigate different architectures with comparable area including SMT and CMP cores for a two application workload.

The rest of this chapter is organized as follows. In Section 4.1 we discuss the prior work, followed by an introduction of the architectures we investigate in Section 4.2. Section 4.3 presents the methodology and experimental results are in Section 4.4.

## 4.1 Related Work

### 4.1.1 Thermal and Reliability Management

The circuit design community has proposed a great deal of work on dynamic power optimization techniques, which are also used as dynamic thermal management techniques in microprocessors in various forms. Such techniques include dynamic voltage scaling (DVS) and dynamic frequency scaling (DFS). In this section we will focus on the studies that are close to our own and specifically target microprocessor power/thermal optimization.

The Pentium 4 incorporates a low cost, yet reliable, thermal management system based on processor power modulation that has been commonly used in mobile systems. It utilizes the existing *stopclock*, an architectural low-power logic mechanism that halts the clock signal to the bulk of the processor [GBC01]. whenever any of the thermal sensors indicate that the die is hotter than critical temperature.

Thermal management is automatically invoked whenever any of the thermal sensors indicates that the die is hotter than a predetermined critical temperature. The mechanism stays active until the die temperature drops below the critical value. The clock signal is gated at certain intervals or permanently, depending on the thermal and power management state.

Brooks and Martonosi introduced an adaptive thermal management system through speculation control in [BM00]. They also compared commonly used DTM techniques such as clock frequency scaling, voltage and frequency scaling, decode throttling, speculation control and instruction cache toggling [DM01]. An energy-management framework that combines energy efficiency and temperature management, DEETM, was presented by Huang et al. [WJS00].

They propose several power optimization techniques such as global clock gating,

DVS, sub-banking, filtered instruction cache. Although these studies provide valuable DTM techniques with significant thermal alleviation, detailed resistance-capacitance thermal models were not available at the time. As a result some of the overheating blocks were not addressed.

DIVA [Aus99] is a dynamic implementation verification architecture that decouples the commit stage of the pipeline from the rest so as to place a very simple functional checker to verify the results generated by the highly speculative complex core.

The results are only committed when they are found to be accurate by the functional checker. The checker is simple design lends itself to more rigorous functional unit verification testing and makes it more immune to noise and radiation related faults.

The Razor architecture [EKD03] on the other hand, uses redundancy at the circuit level, double sampling the delay-critical latches between pipeline stages. The redundant latch, called the shadow latch, is controlled with a delayed clock so that it successfully latches the values missed by the main flip-flop.

Since they reduce the supply voltage to below minimum recommended values to save power, the redundant latch is used to detect signal errors. The value in the shadow latch is used to restore the main flip-flop.

Skadron et al. provide details on their architecture-level thermal model and then describe a temperature-aware microarchitecture implementing several temperature control mechanisms using that model [SSH00]. One of the techniques they propose is duplicating structures that are prone to heat problems (such as the integer register file) and swap between register files to alleviate the overheating problem.

### 4.1.2 Activity Migration for Thermal Alliviation

Him, Daash and Cai [LDG02] introduced a dual pipeline processor, with a secondary low-power pipeline. The power efficient single-issue, in-order pipeline only gets activated when the primary pipeline exceed a threshold temperature. When the superscalar core overheats, it is flushed and the secondary pipeline is activated until the primary pipe cools down to a safe temperature.

The register file, fetch engine and execution units are shared among the two pipelines. However, it is important to note that this technique is mainly targeting mobile devices and applications that can tolerate low performance. There is a significant performance penalty when the architecture transitions to the secondary pipeline.

Heo, Barr and Asanovic [HBA03] proposed an activity migration technique for power density reduction. Activity migration reduces the temperature by moving the computation between multiple replicated blocks. They analyze multiple configurations with some of the microprocessor units replicated or shared. The study concludes that the best configuration has a shared instruction cache, data cache, rename table, and issue queue. Although, duplicated microprocessor units reduce the on-chip temperatures, they argue that this is dominated by the performance overhead due to activity migration.

Ghiasi et al. [SD03] suggested migration between asymmetrical cores instead of symmetrical cores. This thermal reduction yields lowered leakage power values and can also be improved with a dynamic voltage scaling technique to further reduce the power and temperature. They investigate optimal ping-pong intervals, exploring the trade-off between thermal alleviation and performance overhead.

### 4.1.3 Temperature-Aware Deisgn Issues for SMT and CMP Architectures

SMT and CMP increase throughput and thus on-chip heat, but also provide natural granularities for managing power-density. Recently there have been a few works to leverage SMT and CMP to address power density.

Powell et al. [PGV04] propose heat-and-run SMT thread assignment to increase processor-resource utilization before cooling becomes necessary by co-scheduling threads that use complementary resources. They propose heat-and-run CMP thread migration to migrate threads away from overhead cores and assing them to free SMT contexts on alternate cores, leveraging availability of SMT contexts on alternate CMP cores to maintain throughput while allowing overheated cores to cool. This co-scheduling allows threads to evenly heat a core and make more efficient use of activity migration.

Donald et al. [DM04] compare SMT and CMP and find that SMT produces more thermal stress than CMP. They propose that allowing hot functional blocks to be allocated more die area can reduce processor's hottest unit temperature significantly. In a more recent study [LBH05], Li et al. study the heating behavior of CMP and SMT cores and the performance of different DTM techniques on these architectures.

## 4.2 Core Swapping on a Microcore

We examine core swapping on a dual core version of our microcore architecture, where a set of larger structures are moved out of the cores as helper engines 2(figure 4.1). The results in 2 showed that the microcore architecture is able to reduce total processor power dissipation by 20% on average, while it attains comparable performance to a deeply pipelined monolithic design at the same clock frequency. The inherent power efficiency of the microcore makes it an attractive design for temperature aware architectures. We use the microcore framework of 2 for the remainder of this chapter.

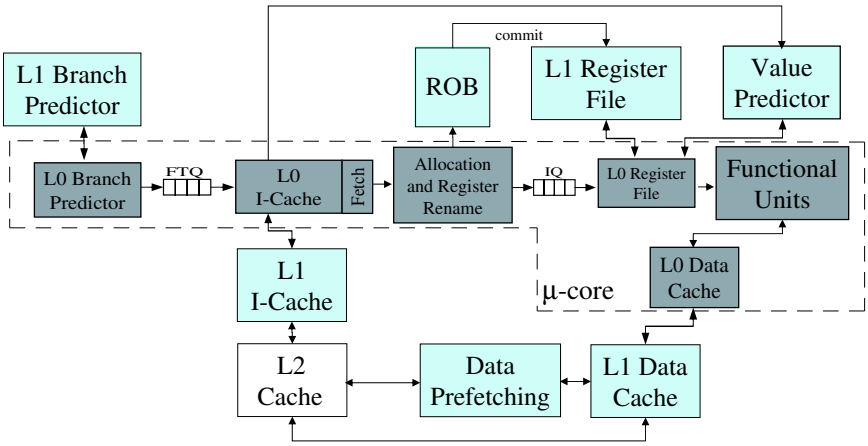


Figure 4.1: The factored microcore architecture

The microcore design provides a suitable framework for core swapping for two main reasons:

- Performance efficiency: State buffering in helper engines shared between cores reduces the core swapping overhead significantly.
- Area efficiency: Resource duplication is limited to the smaller microcores, while larger structures are shared between cores.

Core swapping can impact processor performance significantly. On a core swap, we flush the pipeline similar to a branch misprediction. We need to propagate the first level register file state, the store buffer, and the dirty cache blocks in the L0 cache. Register file and store buffer state is copied to the other core, and dirty cache blocks are written back to the level one cache (the helper engine), which is shared between the cores. We could also have used a writethrough policy with our L0 data cache. We overlap copying register file and store buffer state and writing back dirty blocks with the restart of the pipeline on the new core. Unlike [HBA03], our core swaps are triggered by thermal sensors, removing the overhead of unnecessary core swaps.

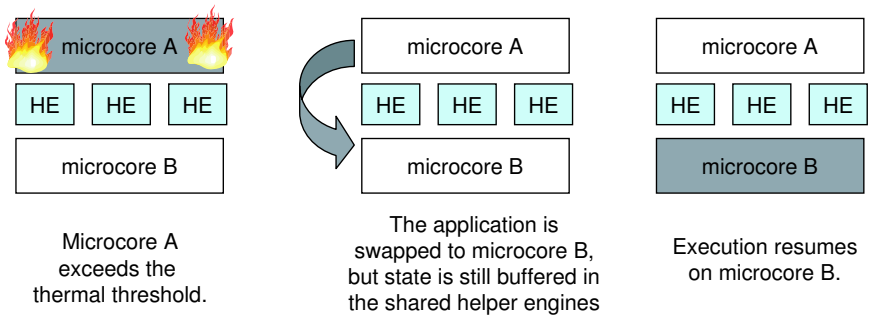


Figure 4.2: Core Swapping

When one core exceeds a thermal threshold, the application workload is swapped to the other core.

The cold start effect of caches and predictors causes an even more severe impact on the second core. These structures need to warm up and depending on their size, there is an overhead involved. The microcore architecture, with less state in the core and more buffering between the cores, provides a very tolerant framework for core swapping, while decreasing the cost of core replication to a small factored core. We evaluate these features in Section 4.4 by studying the performance and thermal behavior of selected architectures with comparable area, including core swapping.

## 4.3 Methodology

To evaluate our results we used 8 floating point and 8 integer benchmarks from SPEC2000 set for our experiments. We simulate 100 Million instructions after fast-forwarding an application-specific number of instructions as proposed by Sherwood et. al in [SPH02].

In addition to modeling all of the structures and latencies in the microcore architecture, we have extended SimpleScalar to include a cycle accurate, execution driven model of simultaneous multithreading (SMT) [TEL95] and chip multiprocess-

ing (CMP) [HNO97].

### 4.3.1 Power and Thermal Simulator

A complete analysis of the static and dynamic power consumption and resulting temperature characteristics of different architectures is crucial to our study. Our power/thermal simulator performs cycle-accurate analysis of the investigated architectures based on the following recently developed power and thermal models. We used process parameters for a *70nm* process at 5.6GHz with 1V supply voltage, in order to have a better understanding of next generation submicron, low supply voltage, aggressively clocked microprocessors.

We have incorporated Wattch [BTM00] models for dynamic power analysis of the microprocessor blocks. The experimental results we present are extracted with the most aggressive conditional clocking strategy, where the dynamic power scales linearly with access to the ports.

For submicron technologies, such as 70nm, leakage power constitutes a significant portion of the overall power. ITRS [itr03] predicts that leakage power is likely to increase exponentially and make up 50% of the total power dissipation for the next deep submicron processes. We adapted leakage models from Hotleakage [ZPS03] in our power/thermal simulator. Hotleakage models are extended and improved versions of the well-known Butts and Sohi leakage equations [BS00]. The public version of Hotleakage only provides a software implementation of the leakage models for the data cache. We have extended and modified the tool significantly to accommodate other caches and cache-like structures in the microprocessor. We also used leakage parameters from Hotleakage's predetermined values specific to the 70nm process technology.

A detailed and accurate thermal analysis of the different architectures we explore

in this study is crucial. We use Hotspot's [KMH03] thermal resistance/capacitance models and RC solvers for our analysis. We used area values for our various architectural blocks based on our analysis with CACTI [SJ01]. Our power/thermal simulator also incorporates the thermal runaway phenomena enabled by the Hotleakage and Hotspot models. Thermal runaway is caused by the exponential dependency of leakage power on temperature: increased temperature increases leakage power, increased leakage power causes even further increase in temperature. The positive feedback loop between leakage power and temperature is quite significant and can cause device failure.

Dynamic and leakage power consumption for each microprocessor unit are collected over a predetermined thermal sampling interval, as the temperatures change over periods greater than every cycle. We experimented with various sampling interval lengths, in order to explore the trade off between error rate and computational overhead. Hotspot [KMH03] proposes a 10K instruction sampling interval for 180nm and 3.3GHz. We used 100 cycle sampling interval for 70nm and 5.6 GHz as well.

Heo, Barr and Asanovic [HBA03], argue that most heat is dissipated vertically on the microprocessor chip, as the wafer thickness is much smaller than the chip area. Therefore, they assume infinite lateral resistances, although it leads to the worst case temperature gradients. We follow their example, and tune HotSpot to only consider the vertical component of temperature. Lateral modeling, while possible with HotSpot, is unrealistic without a more accurate floorplan of the various architectures we consider.

### 4.3.2 Dynamic Thermal Management Techniques

We used some thermal threshold values for the dynamic thermal management techniques. Critical thermal threshold, is the maximum tolerated temperature for proper functionality (timing data errors are likely after this value). DTM needs to be activated

at this temperature. Safety thermal threshold, is the safety temperature for DTM to deactivate during the cooling down. We assume that the critical thermal threshold is 82°C and the safety thermal threshold is 79°C for the 70nm technology process we are investigating according to the ITRS [itr03] projections and results from [KMH03].

Global clock gating is commonly used in many of todays microprocessors, such as the Pentium 4 as discussed in Section 4.1. We implemented global clock gating similar to Pentium 4 as discussed in Section 4.1. The global clock signal is shut down, whenever on-chip temperatures exceed the critical thermal threshold of 82°C. The processor resumes normal operation after the chip temperatures cool down below the safety threshold of 79°C.

We have incorporated an idealized version of dynamic frequency scaling for the experimental analysis. Our DFS has two different frequency settings: 5.6GHz for the normal operation and 4GHz for thermal relief, which gets activated as soon as on-chip temperatures reach the 82°C critical thermal threshold. Because we do not model frequency scaling to lower frequencies, it is possible for the processor to heat up even during the operation in lower frequency. We model a global clock gating that is activated whenever on-chip temperatures exceed the critical thermal threshold of 85°C to prevent such cases. Usually there is a large latency (on the order of $\mu$secs) incurred every time the frequency is adjusted, which results in significant performance penalties in dynamic frequency scaling schemes. Skadron et al. [KMH03] report 10 $\mu$sec for the non-idealized version of DFS. In our dynamic frequency scaling implementation there is no overhead, delay or penalty involved with changing the frequency of the processor.

ITRS [itr03] projects very minor changes in Vdd for 130nm and smaller processes. Noise margins also pose a barrier to scaling the operating voltages [KMH03]. Our starting supply voltage is only 1V, and because [KMH03] indicates that DFS was the best performing DTM among their experimented set of DTMs, we have decided to

model DFS.

Our thermally-triggered core swapping mechanism gets activated when a core reaches 82°C. The runs with this architecture assume an extra core (identical to the main core) that can be used to offload an application when one core overheats. The computation is migrated to the cooler core until the active core heats above the critical thermal threshold and another swap is required. Thermally-triggered core swapping minimizes the swapping overhead relative to approaches that swap at fixed intervals regardless of core temperature.

## 4.4 Experimental Results

In this section we compare the performance impact of core swapping to other DTMs on the microcore architecture. In particular, we examine the ability of the microcore to buffer state when core swapping, resulting in minimal degradation in performance. We further study the area overhead of duplicating the microcore, and investigate different architectures with comparable area. We use the power/thermal simulator framework discussed in Section 4.3 to explore different alternatives of simultaneous multithreading (SMT) and chip multiprocessing (CMP).

### 4.4.1 Core Swapping Performance Overhead

Core swapping can impact processor performance significantly. Figure 4.3 illustrates how this impact can be reduced by buffering state in helpers that are shared between two cores. For this set of results we fix the swap period at 10K cycles regardless of thermal characteristics of the benchmarks. Every 10K cycles we flush the pipeline similar to a branch misprediction and the application workload is swapped to the other core.

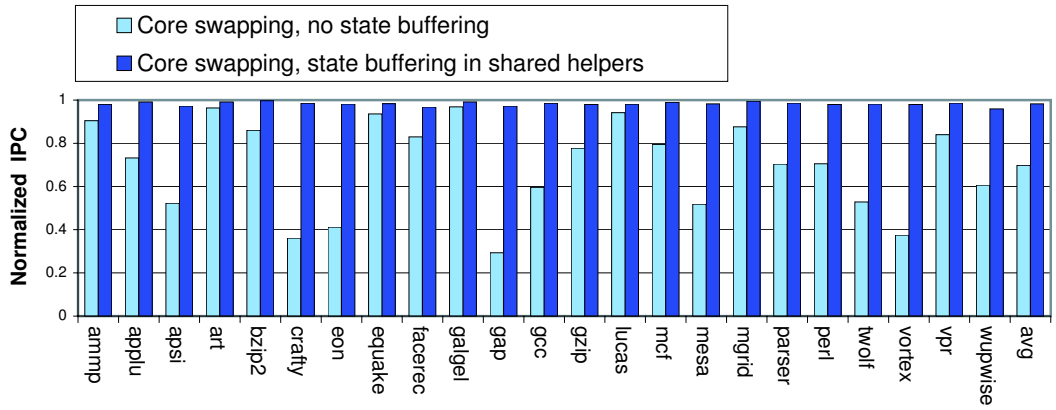


Figure 4.3: Core swapping overhead with and without buffering

The first bar presents the performance impact when helpers are flushed on every core swap. Values are normalized based on a similar architecture with no core swapping. The average 31% degradation reflects the cold start effect of caches and predictors after a core swap. The second bar demonstrates the efficiency of buffering state in our shared helpers. The impact of core swapping is reduced to an average of merely 2%. Benchmarks that depend more heavily on branch prediction such as *crafty*, *eon* and *gap* suffer more from core swapping and cold start effect of branch predictor. Buffering state in second level predictor can reduce this impact significantly.

### 4.4.2 Single Application Workload

Core swapping has an area overhead from the duplication of core resources. The microcore design reduces this cost by sharing larger structures among cores and limiting the duplication to a small microcore. We extracted area numbers for most of the architectural blocks using CACTI [SJ01], and used [PJS97] for structures that could not be easily modeled with CACTI. Our data indicates that duplicating the microcore only increases our overall chip area by 25%. In this section we study alternative architectural

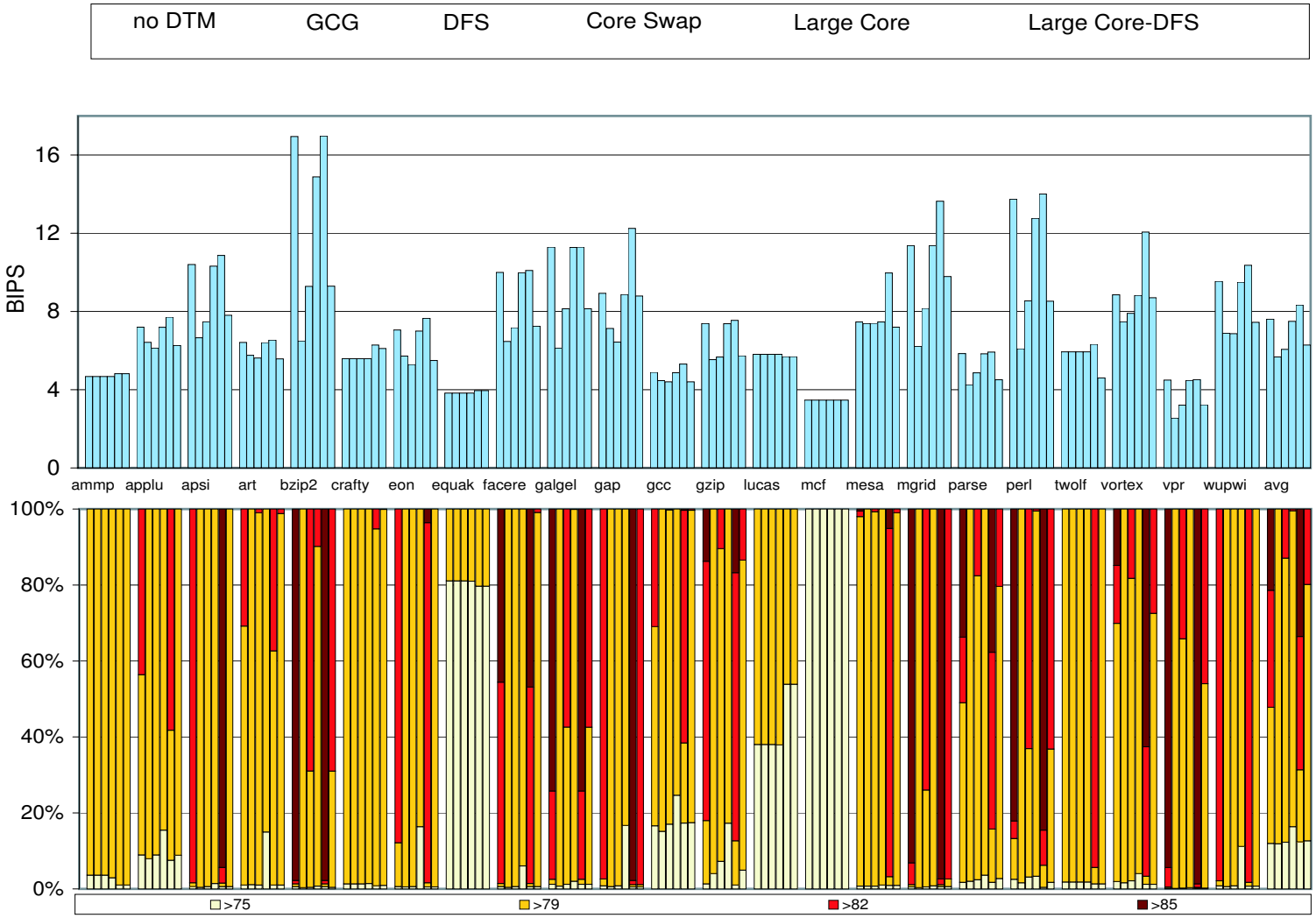


Figure 4.4: Thermal and Performance behavior of different architectures with and without DTM

designs with a comparable increase in area as well as alternative thermal management techniques.

Figure 4.4 compares the performance and thermal behavior of our microcore architecture in the presence of different DTM techniques, including core swapping. The upper half of the figure shows the performance in BIPS for different benchmarks, and the lower half illustrates the heating behavior of the investigated architectures. This latter component shows the percentage of cycles for which at least one block exceeds the indicated temperatures: 75°C, 79°C, 82°C and 85°C. Darker colors in the lower graphs indicate higher temperatures.

Our detailed thermal analysis considers all of the possible overheating blocks. Although some of the hotspots were common among different benchmarks, such as the register file, load-store queue, etc, others varied across the different benchmarks and configurations. Even though the location of hotspots can provide a level of insight, the thermal behavior of the architecture can also be captured by the number of cycles that any of the blocks exceed a given thermal threshold.

It is important to note that the ITRS projects a reduction in maximum permitted junction temperatures for the future generations of process technologies. The maximum tolerated junction temperatures are around 85°C for 130nm and even lower for smaller process technologies.

As mentioned earlier in Section 1.1, performance degradation is commonly experienced with dynamic thermal management techniques. The degradation usually comes from various sources such as frequency decrease, voltage reduction, clock gating. Performance degradation might be quite significant depending on the DTM technique.

The first and fifth bars demonstrate results without thermal management of any kind. The first bar is our default microcore and the fifth bar is the microcore with larger resources. This latter bar doubles critical processor resources including the issue

window (64-entry issue window), the first level data and instruction caches, and the first level branch predictor. The second and third bars show our microcore with global clock gating and idealized dynamic frequency scaling. The fourth bar presents core swapping results. The final bar represents the larger microcore with DFS.

For example benchmarks *bzip2* and *mgrid* see temperature greater than 85°C almost all the time when no thermal management is applied, and all DTMs impact their performance significantly.

Note that for many benchmarks, temperature frequently exceeds the thermal threshold, 82°C. These results should be considered as an upper bound for performance that can not be achieved. It would require sustained operation at a temperature beyond the critical thermal threshold, and a processor operating under such conditions would likely have timing, data and reliability complications.

GCG has a very significant performance penalty from frequently disabling the global clock signal. On average global clock gating sees a 25% degradation on performance.

Thanks to state buffering in the helper engines, core swapping is able to come close to the performance of the architecture without any DTM in most cases, seeing an average of only 2% degradation in performance. Core swapping is extremely effective at thermal management, reducing the temperature below 79°C at least 82% of the time for all benchmarks and well above 95% of the time for many benchmarks. Even *bzip*, which spends most its execution time over 85°C is able to reduce its temperature below 79°C around 83% of the time using core swapping, with only an 11% degradation in BIPS.

Our results with core swapping, indicate that helper engines rarely heat up to critical temperatures, due to the relatively smaller number of accesses to those structures. For benchmarks we simulated, the block or blocks overheated were always inside the

microcore. This is crucial for the effectiveness of our core swapping method in reducing the temperature.

The actual scaling of frequency in DFS can have a significant performance impact if used often. However, we have used an idealized DFS implementation (see Section 4.3) that does not see this impact when transitioning between frequencies. Despite this advantage, core swapping is still able to outperform DFS.

The configurations with larger resources demonstrate one alternative use of the area overhead of core swapping. In some cases, the use of this overhead to instead increase the structures of the microcore has a clear performance advantage, 6% speed up over our baseline architecture on average – but this does nothing to alleviate the thermal problem. Even with idealized DFS on these larger resources, the performance impact is too great in most cases.

Figure 4.5 illustrates thermal profile of different DTMs on a core running *vortex* over measured time. Temperature is sampled in 10K cycles from the first observation of critical temperature of 82°C, so the starting point of sampling is the same instruction for all graphs. The red line indicates the cycles with active DTM. This means stalled pipeline for GCG and operation in lower frequency mode for DFS. For core swapping, active DTM indicates running on the second core. Note that all the graphs start on first time DTM was activated, while GCG and DFS start with the critical temperature of 82°C, core swapping has an advantage of starting from low temperature of 73°C on the other core.

As can be seen from figure 4.5-a, after shutting down the global clock signal in GCG it takes approximately 300K cycles for the processor to cool down to the safe temperature of 79°C and resume process again. During the normal operation similar time of 300K cycles heats up the core to the critical temperature and activates GCG again. This switching behavior can continue until the process moves to a different

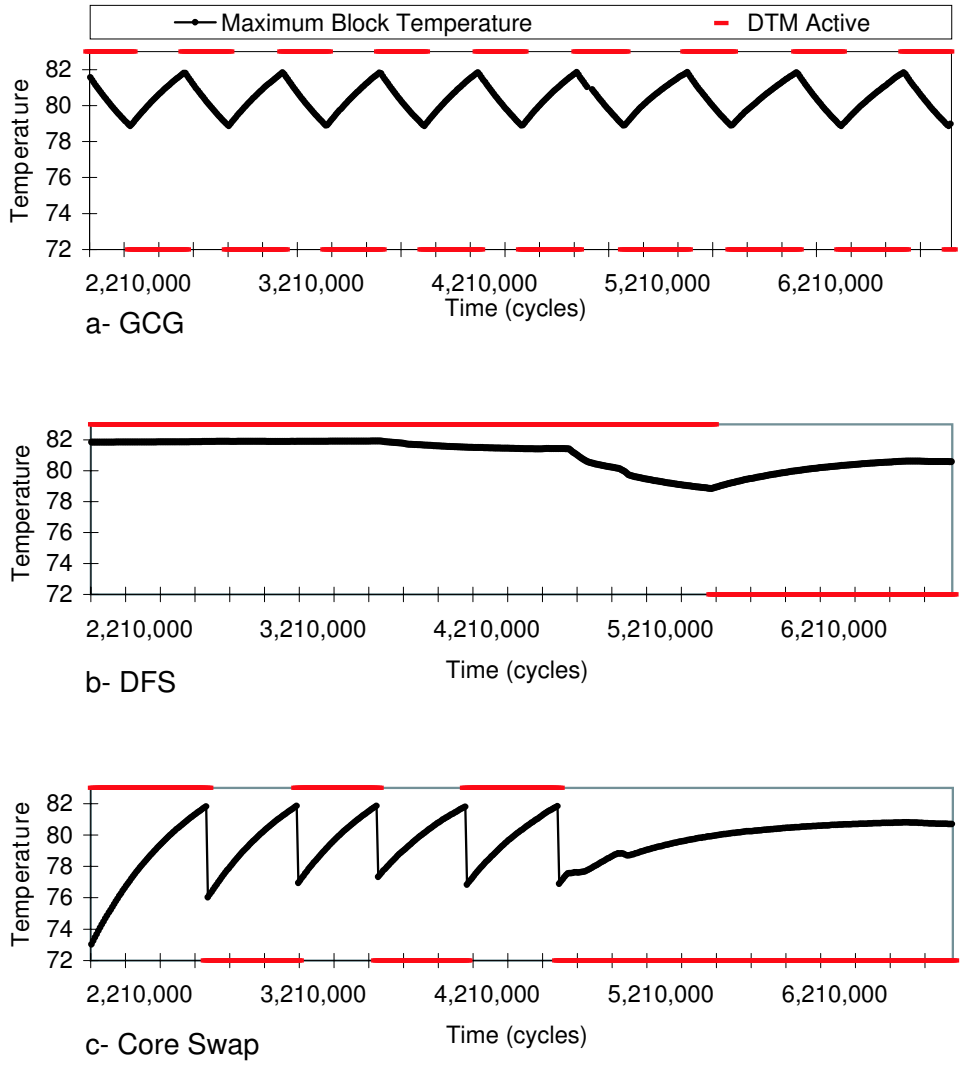


Figure 4.5: Thermal behavior of *vortex* in presence of different DTM techniques. a: global clock gating, b: dynamic frequency scaling, c:core swapping

phase where less heat is generated.

Figure 4.5-b shows that in DFS once processor has reached the critical temperature, it takes much longer to cool down in the lower frequency mode of operation. Eventually application switches to a cooler phase and allows the processor to reach the safe temperature and resume in high frequency again. Long periods of execution in lower frequency in DFS and frequent pipeline stalls in GCG can both hurt performance significantly as can be seen from figure 4.4

Core swapping in figure 4.5-c shows similar thermal profile to GCG. Process migration to the cooler core can be viewed as GCG where the core cools down to safe temperature instantly. The only overhead occurred with core swapping is performance penalty of cold start of the core that is significantly reduced using shared helpers of the microcores. Another difference between core swapping and GCG is the lower temperature of the second core (average of 77°C) compared to the temperature GCG is deactivated. This difference in initial temperature explains the fewer number of core swaps (approximately every 600K cycles) compared to the number of times GCG was activated. This figure clearly indicates the performance benefit of core swapping over DFS and GCG for benchmark *vortex*.

To have a better understanding on behavior of each of the DTMs examined, we present details for each application in table 4.1. This table indicates how frequently each DTM was activated and the percentage of the execution time in temperature-control mode, i.e. periods of stalled pipeline due to global clock gating and operation in lower frequency due to frequency scaling.

When applying GCG as thermal management, our applications spend an average of 20% in stalled mode. This percentage increases to an average of 50% in low frequency mode for DFS and in cases of more aggressive applications is close to 100% of program's execution. Frequency switch count of 1 for those applications indicates that

once frequency is scaled down, the temperature never reaches the safe threshold and execution resumes in lower frequency for the rest of the execution time. This can be fixed by adopting more levels of frequency scaling, which in turn will hurt performance further more. Core swapping shows an interesting behavior. For many of those applications infrequent core swaps can keep the core in low temperatures as can be seen in *apsi*, *facerec* and *gap*. Applications *bzip2* and *perl* see significantly more number of swaps, which corresponds to their high activity and high temperatures. Both these applications perform much better with core swapping compared to the other DTMs. Note that these results are for running 100M instructions, so for *bzip2* this shows a swap every 2000 instructions on average (for an IPC of close to 2, this indicates core swap every 1K cycles). Other applications have noticeably less infrequent core swaps.

### 4.4.3 Two Application Workload

In this section we consider the trade-off between using this extra core to run a second application instead of core swapping. It is important to note that the design space for chip capable of supporting two threads is quite large, but we believe that the results we present in this section highlight some interesting observations that will be explored in future work. In CMP approach each application runs on a separate microcore with its own distinct set of helper engines. This design does not make use of core swapping, but can use dynamic frequency scaling (DFS) to *independently* scale the frequency of either core.

We compare this CMP approach to a microcore that has been enhanced with SMT to run two applications on a single core. This architecture also has a different set of helper engines per thread. One alternative is to simply have one SMT core running both threads without any DTM. Another is to have two SMT cores, but use core swapping to migrate both threads together from one core to the next. A final option is to have

| | GCG | | DFS | | Core Swap | DFS Large | |
|---|---|---|---|---|---|---|---|
| benchmark | stall cnt | stall | fq switch cnt | low fq | swap cnt | fq swtich cnt | low fq |
| ammp | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0.00 |
| applu | 36 | 0.11 | 3 | 0.43 | 25 | 4 | 0.58 |
| apsi | 117 | 0.36 | 1 | 0.98 | 80 | 1 | 0.98 |
| art | 38 | 0.10 | 22 | 0.35 | 32 | 26 | 0.43 |
| bzip2 | 206 | 0.62 | 38 | 0.88 | 41053 | 38 | 0.88 |
| crafty | 0 | 0.00 | 0 | 0.00 | 0 | 4 | 0.07 |
| eon | 73 | 0.19 | 6 | 0.84 | 51 | 1 | 0.98 |
| equake | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0.00 |
| facerec | 118 | 0.35 | 1 | 0.98 | 84 | 1 | 0.98 |
| galgel | 161 | 0.46 | 1 | 0.97 | 273 | 1 | 0.97 |
| gap | 61 | 0.20 | 1 | 0.97 | 42 | 1 | 0.98 |
| gcc | 42 | 0.09 | 11 | 0.27 | 31 | 14 | 0.51 |
| gzip | 98 | 0.25 | 13 | 0.76 | 73 | 13 | 0.79 |
| lucas | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0.00 |
| mcf | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0.00 |
| mesa | 3 | 0.01 | 1 | 0.02 | 2 | 1 | 0.96 |
| mgrid | 157 | 0.45 | 1 | 0.99 | 149 | 1 | 0.99 |
| parser | 139 | 0.27 | 6 | 0.51 | 149 | 11 | 0.78 |
| perl | 198 | 0.56 | 32 | 0.80 | 12035 | 31 | 0.85 |
| twolf | 0 | 0.00 | 0 | 0.00 | 0 | 1 | 0.94 |
| vortex | 45 | 0.16 | 7 | 0.30 | 30 | 1 | 0.96 |
| vpr | 371 | 0.43 | 1 | 0.99 | 428 | 1 | 0.99 |
| wupwise | 87 | 0.28 | 1 | 0.98 | 55 | 1 | 0.98 |
| average | 80 | 0.20 | 5 | 0.51 | 1934 | 6 | 0.67 |

Table 4.1: Number and duration of pipeline stalls, frequency scaling and core swaps in GCG, DFS and Core Swap for each application

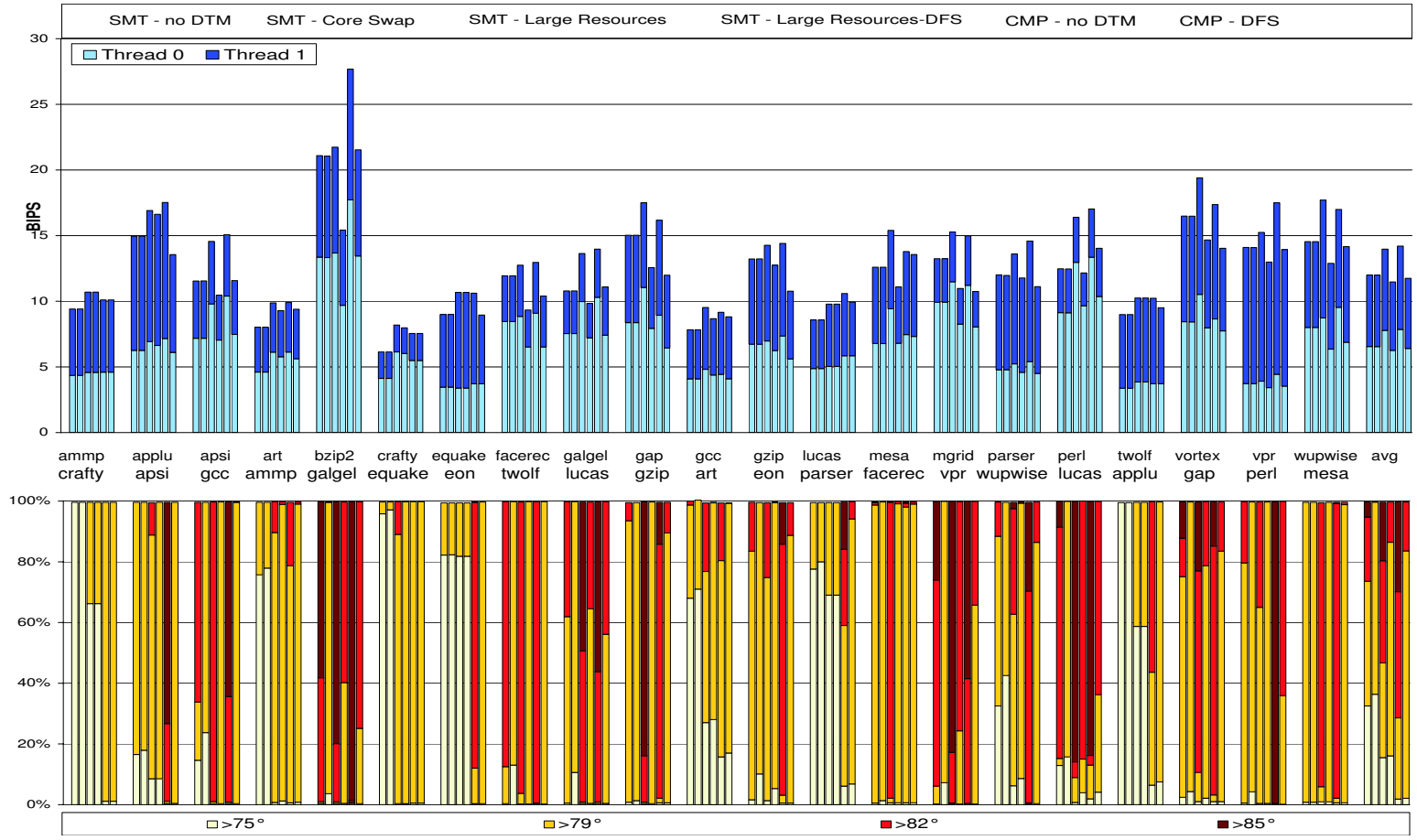


Figure 4.6: Thermal and Performance behavior of different architectures for two-thread workloads with and without DTM

only one SMT core, but make it proportionately larger SMT (to compensate for the added area of the second core), and explore this with and without DFS.

Note that the CMP has the distinct advantage of being able to independently apply its DTM to only one thread. The SMT architecture with core swapping must swap both applications when a thermal threshold is exceeded. Similarly, the SMT architecture with DFS has to scale frequency for both threads.

Figure 4.6 displays the comparison between the SMT and CMP architectures. Applications are paired randomly. Our results include 5 integer-integer and 5 floating point-floating point sets. For the BIPS results, a black line divides the contribution to BIPS from each application. The first bar shown on the graph is a single SMT core with no thermal management. The second bar shows two SMT cores that use core swapping for thermal management – both applications exist on only one core at a time and migrate together. The third bar is a larger single core SMT with double size issue window, caches, and branch predictor to compensate for the area overhead of core swapping. Again, this provides an alternative way to spend that area overhead. The fourth bar is this same architecture with DFS – both applications and set of helpers run in same frequency at all times. The last two bars show two single threaded CMP cores without any thermal management and with our idealized DFS respectively.

When SMT resources are increased to account for second core, most applications see a noticeable performance improvement. The benchmark mixes of *perl.lucas*, *vortex.gap* and *wupwise.mesa* are examples that see up to 40% speed up from larger issue window and cache structures. On average doubling processor resources leads to a 16% improvement on application mixes examined.

Similarly running each application on a separate core in CMP improves thread level parallelism (TLP) significantly. CMP sees an average speed up of 18% compared to baseline SMT architecture. Running *bzip2* and *galgel* on separate microcores improves

their performance by 30% as can be seen from figure 4.6.

Unfortunately, both these alternatives see a significant number of cycles of thermal violation. SMT has lower temperatures than CMP for some applications – this is directly related to the improvement in BIPS seen by CMP. As the difference in BIPS grows, so does the difference in the temperature profile. Figure 4.6 also indicates that there is no certain winner for all applications between CMP and SMT. This is true with or without applying frequency scaling for thermal management and agrees with prior research in this area.

When thermal management is applied, core swapping shows the lowest impact of an average of 1% on performance for successful thermal alleviation. Applying DFS on large SMT core slows down performance by 18% on average. Some mixes of applications such as *ammp.crafty*, *applu.apsi* and *equake.eon* have few cycles of thermal violation while they get a noticeable jump in performance from larger structures. For these benchmarks applying DFS on large SMT provides best performance without thermal violation among designs studied. Other mixes such as *bzip2.galgel* and *mesa.facerec* run in higher temperatures and see more impact from applying DFS. Similarly while running applications on separate cores on CMP shows promising results, scaling frequency on cores that run on high temperatures eliminates the benefit of extra core as can be seen in *applu.apsi* and *parser.wupwise*.

In general our results indicate that while there is a performance and area overhead associated with core swapping, for every application mix that heats up the processor and runs in critical temperatures core swapping provides better or comparable results to the best of the other alternative designs. This is particularly important since recent trends in microprocessor design are all leading towards higher temperatures in next generation processors.

Table4.2 presents details about the frequency of core swaps as well as frequency

| | SMT CS | Large SMT DFS | | CMP DFS | | | |
|---|---|---|---|---|---|---|---|
| | | | | core one | | core two | |
| benchmark | swap cnt | fq cnt | low fq | fq cnt | low fq | fq cnt | low fq |
| ammp.crafty | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| applu.apsi | 0 | 1 | 0.10 | 3 | 0.43 | 1 | 0.67 |
| apsi.gcc | 53 | 1 | 0.47 | 1 | 0.44 | 13 | 0.35 |
| art.ammp | 0 | 21 | 0.09 | 24 | 0.14 | 0 | 0 |
| bzip2.galge | 570 | 3 | 0.57 | 47 | 0.59 | 29 | 0.42 |
| crafty.equake | 0 | 16 | 0.02 | 0 | 0 | 0 | 0 |
| equake.eon | 0 | 0 | 0 | 0 | 0 | 6 | 0.42 |
| facerec.twolf | 1100 | 5 | 0.39 | 1 | 0.42 | 0 | 0 |
| galgel.lucas | 45 | 4 | 0.35 | 1 | 0.34 | 0 | 0 |
| gap.gzip | 1 | 1 | 0.57 | 1 | 0.78 | 13 | 0.76 |
| gcc.art | 3 | 17 | 0.24 | 7 | 0.20 | 1 | 0.01 |
| gzip.eon | 15 | 14 | 0.29 | 13 | 0.74 | 4 | 0.91 |
| lucas.parser | 0 | 0 | 0 | 0 | 0 | 15 | 0.39 |
| mesa.facerec | 2 | 1 | 0.61 | 2 | 0.04 | 2 | 0.03 |
| mgrid.vpr | 174 | 3 | 0.32 | 2 | 0.33 | 1 | 0.99 |
| parser.wupwise | 12 | 15 | 0.35 | 7 | 0.49 | 1 | 0.57 |
| perl.lucas | 197 | 13 | 0.23 | 121 | 0.40 | 0 | 0 |
| twolf.applu | 0 | 0 | 0 | 0 | 0 | 2 | 0.17 |
| vortex.gap | 27 | 14 | 0.67 | 7 | 0.29 | 1 | 0.95 |
| vpr.perl | 20 | 12 | 0.36 | 106 | 0.73 | 108 | 0.15 |
| wupwise.mesa | 0 | 1 | 0.93 | 1 | 0.75 | 2 | 0.05 |
| average | 58 | 6.7 | 0.31 | 16.4 | 0.34 | 9.4 | 0.32 |

Table 4.2: Number and duration of frequency scaling on SMT and CMP running DFS and number of core swaps for two thread workloads.

scales for SMT and each core of CMP running DFS for two thread workloads.

# CHAPTER 5

# Study of Scalable Helpers

Recent studies have examined the impact of technology scaling on processor performance [AHK00], describing the tradeoffs between the IPC achievable by a processor and the processor cycle time. According to this study, measurable gains in both IPC and cycle time will become more and more difficult as process technology sizes shrink. As feature sizes continue to shrink, it has become evident that wire latencies are not scaling with transistor latencies. This trend has been termed the interconnect scaling bottleneck [Boh95, Boh98]. Interconnect is expected to scale poorly due to the impact of resistative parasitics and parasitic capacitance. The result of this impact is that structures composed of a significant amount of closely packed interconnect (like memory structures) may scale poorly to future technology sizes, with larger structures scaling even worse than smaller structures due to the greater amount of interconnect present in such structures. This is further complicated by the observation that processor speeds are scaling at a much faster rate than DRAM latencies, a trend referred to as the memory wall [WS95, Wil95].

As processor clock speeds increase, and cycle times decrease, structures in the processor pipeline need to either decrease their access time or increase the degree to which they are pipelined. Unfortunately, in order to decrease the access time of a memory structure, the size or complexity (i.e. number of ports, associativity, etc) of the structure would also likely need to be reduced. Reducing the size or complexity of a memory structure can mean reduction in performance, and pipelining a structure

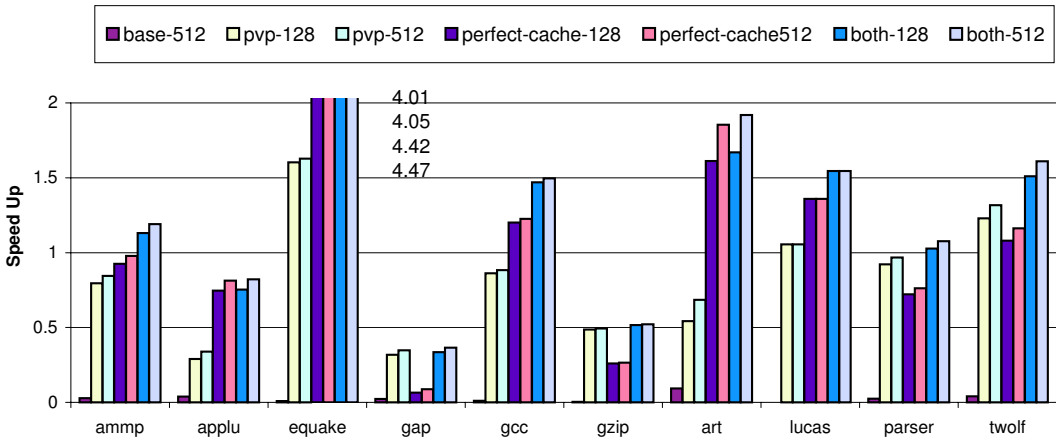


Figure 5.1: Maximal benefit from out-of-order execution, value prediction and data prefetching

across multiple stages can result in considerable performance degradation, especially in the case of the data cache [SC02]. Poor memory scaling combined with latency constrained on-chip caches, create a critical processor bottleneck.

A number of techniques have been proposed to alleviate the memory bottleneck, mostly attacking load instruction latency. In this work, we compare three techniques for dealing with load latency: out-of-order execution, value prediction [LS96a], and data prefetching [Jou90]. Each of these techniques is characterized by different trade-offs, and in this study we are concerned with evaluating these in the context of their ability to scale to future technology demands.

Figure 5.1 presents some preliminary data on the maximal benefit achievable with these three techniques on a subset of the SPEC 2000 benchmark suite. Simply increasing the size of the instruction window to improve the benefit from out-of-order speculation provides an IPC improvement of 3% on average (base-512). Keeping the instruction window the same size, but successfully predicting every load instruction over the execution of an application provides 81% improvement on average (pvp-128).

A larger window provides 86% when combined with perfect value prediction (pvp-512). An infinitely sized data cache (a close approximation of perfect data prefetching) provides 120% improvement (perfect cache-128), and 126% improvement when used with a larger instruction window (perfect cache-512). Finally, value prediction and data prefetching together provide a 144% improvement in IPC (both-128), and a 150% improvement when used with a larger instruction window (both-512). It is important to note several things from this result. First, that scaling the instruction window alone has limited potential. Note that we do not scale instruction issue window. Second, that value prediction and data prefetching do not attack the same sources of latency and in all cases there is some improvement from combining both techniques together. Third, that value prediction can outperform even an infinitely sized cache because it can create instruction level parallelism by breaking true data dependencies. Finally, that an infinitely sized cache can outperform even perfect value prediction because value prediction is an inherently speculative technique that requires verification and this verification can be impacted by memory latency and the size of the instruction window. We will explore these tradeoffs further in the remainder of this chapter.

The remainder of this chapter is organized as follows: Section 5.1 introduces the methodology we use to gather results in this chapter. Sections 5.2, 5.3 and 5.4 explore scaling the instruction window (deriving more benefit from out-of-order execution) value prediction and data prefetching respectively. Section 5.5 examines the combination of these approaches.

## 5.1 Methodology

To perform our evaluation, we collected results for a randomly selected subset of the SPEC2000 benchmark suite. The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C and FORTRAN compilers. We compiled the bench-

| Program | Data set | %lds | %L1 miss-rate | %L2 miss-rate | Branch accuracy | IPC |
|---|---|---|---|---|---|---|
| ammp | ref | 27.4 | 8.4 | 32.7 | 0.94 | 0.94 |
| applu | ref | 30.1 | 18.8 | 17.9 | 0.97 | 1.15 |
| equake | ref | 26.9 | 40.3 | 67.1 | 0.94 | 0.59 |
| gap | ref | 40.7 | 18.4 | 31.4 | 0.97 | 0.42 |
| gcc | 00-exp-ref | 26.0 | 1.1 | 17.3 | 0.99 | 1.61 |
| gzip | source-ref | 24.2 | 12.4 | 29.8 | 0.95 | 0.72 |
| art | 470 | 20.0 | 9.8 | 0.8 | 0.92 | 1.54 |
| lucas | ref | 12.5 | 20.3 | 33.3 | 0.99 | 0.83 |
| parser | ref | 26.3 | 5.0 | 22.8 | 0.93 | 0.92 |
| twolf | ref | 26.8 | 7.6 | 31.7 | 0.90 | 0.73 |

Table 5.1: Baseline results showing the input data set, data cache miss rates, percent of loads executed, branch prediction accuracy and IPC for each program

marks under OSF/1 V4.0 operating system using full compiler optimization (-O4 -ifo). We make use of the SimPoint tool [SSC03] to skip the initialization portion of each benchmark and guide the execution of each benchmark for 100 million instructions.

Table5.1 shows the data sets we used in gathering results, the percent of executed instructions that were loads, the L1 and L2 data cache miss rates, the branch prediction accuracy and the baseline architecture IPC for each program.

### 5.1.1 Baseline Architecture

Our baseline simulation configuration models a future generation microarchitecture. We've selected the parameters to capture underlying trends in microarchitecture design. Our out-of-order processor can issue 4 operations per cycle and we explore different instruction window sizes (reorder buffer and register file) in this study, ranging from 128 to 512 entries. The processor has an 8KB 4-way set associative data cache

with a 32-byte block size and a 2 cycle latency (comparable to that of the Pentium 4 [HSU01]). The unified 256KB L2 cache is 4-way set-associative with 64 byte blocks and a 12 cycle latency. We rewrote the memory hierarchy in SimpleScalar to better model bus occupancy, bandwidth, and pipelining of the second level cache and main memory. The round trip time to memory is 120 cycles and our bandwidth to memory is 32 bytes/cycle. We extended SimpleScalar to model a 32-entry issue queue, providing scalable wakeup and select logic even with a much larger instruction window. As in the Pentium 4, we make use of a 7-stage schedule to execute window to model the impact of deeply pipelining the instruction scheduler and dispatcher, and the register file. We also use selective replay [HSU01] for load mischeduling. In the baseline architecture, there is a 20-cycle minimum branch miss-penalty (this includes the impact of the schedule to execute window). The processor has 4 integer ALU units, 2-load/store units, 1-FP adders, 1-integer MULT/DIV, and 1-FP MULT/DIV. The latencies are: ALU 1 cycle, MULT 3 cycles, Integer DIV 20 cycles, FP Adder 2 cycles, FP Mult 4 cycles, and FP DIV 12 cycles. All functional units, except the divide units, are pipelined allowing a new instruction to initiate execution each cycle. We use a gshare predictor [McF93] to drive our fetch unit.

## 5.2 Value Prediction

Value prediction has been shown to be effective at reducing instruction latency in the processor pipeline. A value predictor attempts to predict the result of a particular instruction. Load value prediction breaks true data dependencies by allowing instructions dependent on the load execute speculatively using the predicted value. Prior work has demonstrated several predictor architectures including last value prediction [LWS96a, LS96b, LWS96b], two-delta stride prediction [GM96, GG98], context prediction [SS97], and hybrid approaches [WF97b]. More recently, hybrid predictors

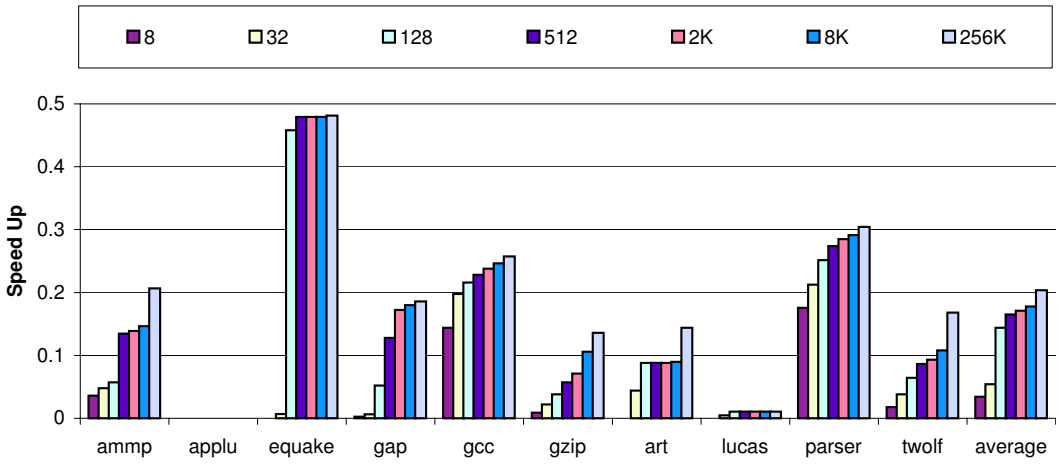


Figure 5.2: Impact of predictor size in performance of value prediction. Speed up is over the baseline architecture.

with the ability to dynamically classify instructions have been evaluated [BJB98b, BJB98a, SYP00]. In [CRT99, Bur99] sophisticated history based confidence estimation was proposed to improve the confidence used by value prediction. [RLP00, BL02] address energy and latency related issues with value prediction. In this work, we look at a hybrid predictor that uses a stride predictor and a first order Markov predictor [WF97b]. We model perfect confidence, since prior work [CRT99, Bur99] shows sophisticated history based confidence estimation can achieve performance close to perfect confidence. This is very different from perfect value prediction (figure 5.1) as we are limited by the ability of the predictor to capture the particular load value pattern and the total capacity of the predictor.

To compensate for poor wire scaling and for the need to improve the processor clock, smaller structures (i.e., predictors, caches) may become more attractive for use in the processor pipeline. However, these smaller predictors will hold less state and there will likely be a reduction in performance. The predictor tables used can be quite complex and fairly large. A good example of this is context prediction [SS98, SS97].

In a context predictor, the last *n* values of an instruction are stored in hashed form in a first level table that is hashed by instruction PC. These values are used to index into a second level table that contains the actual value to be predicted. In [SS98], a first level table with $2^{16}$ entries and a second level table with $2^{20}$ entries were used. Assuming the predictor stores 32-bit values in the second level table, the second level will be at least 32 MB in size, and will likely require multiple cycles to access alone (not including the number of cycles necessary to access the first level table to generate the index to the second level table).

Figure 5.2 shows speedup over the baseline architecture for different predictor sizes. For all of our experiments Markov predictor entries are twice the stride predictor enteries. It can be seen that value prediction is very sensitive to the size of the predictor and sees performance improvement going from an 8k-entry stride predictor with a 16k-entry Markov predictor, to 256K-entry stride and Markov predictor. *equake* improves significantly by 46% using a 128 entry predictor. Storing more values using a larger predictor size does not provide much additional benefit over 128 entry predictor. Some applications experience a big jump in performance *ammp*, *art*, *twolf* using an ideally large predictor. This indicates that even a predictor with 8K enteries can not capture all the predictable loads. On avarge doubling the size of the predictor beyond 128 to 8K provides 2% improvement in performance.

Figures 5.3, 5.4 and 5.5 presents coverage of predictors examined. Figure 5.3 corresponds to the coverage of our hybrid predictor. For all applications this result corresponds directly to performance gain shown in figure 5.2. Figures 5.4 and 5.5 present coverage of each predictor seperately. As can be seen from these results increasing predictor size in stride predictor beyond 2K enteries does not provide additional benefit for many of the applications, where larger Markov predictor can noticably improve performance of value prediction. This is not surprising results, considering the rel-

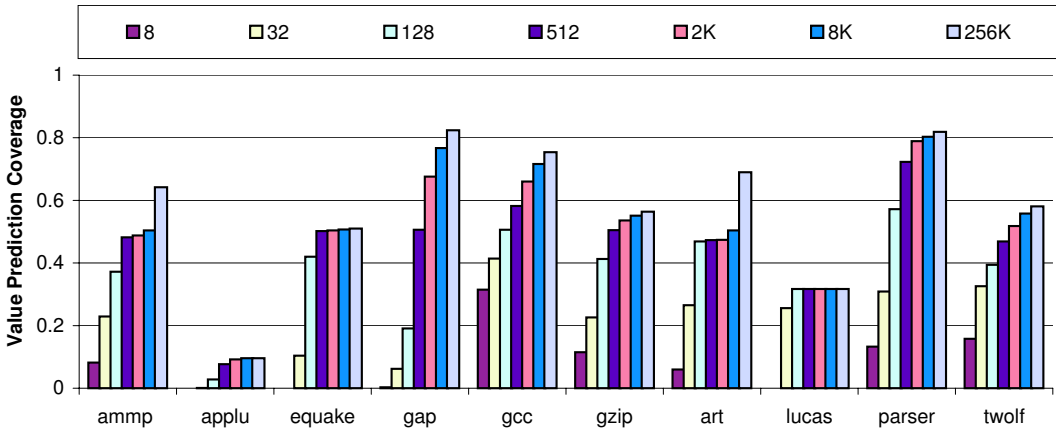


Figure 5.3: Load value prediction coverage in different predictor sizes

atively small number of static load instructions executed frequently and that stride predictor is accessed using instruction PC where Markov predictor is indexed using a hash of PC and previous value seen by that instruction.

## 5.3 Data Prefetching

Many approaches have been proposed for prefetching data to reduce or eliminate load latency, ranging from compiler-directed prefetching to fetch or predictor-directed hardware prefetchers. Compiler-based prefetchers insert explicit prefetch instructions to bring data into the cache before it is needed [LM98, TMA92]. Hardware-based prefetchers attempt to dynamically predict the memory address stream. One class of hardware prefetching, Fetch Guided Prefetching [CB92, KPN97, CB95, ALP93], uses the instruction fetch stream to steer prefetching. Chen and Baer [CB92] used a Look-Ahead PC to run ahead of the normal instruction fetch engine and guide hardware prefetching. Reinman et. al [RAC99] extended this approach to instruction prefetching, using a decoupled front-end and multilevel branch predictor.

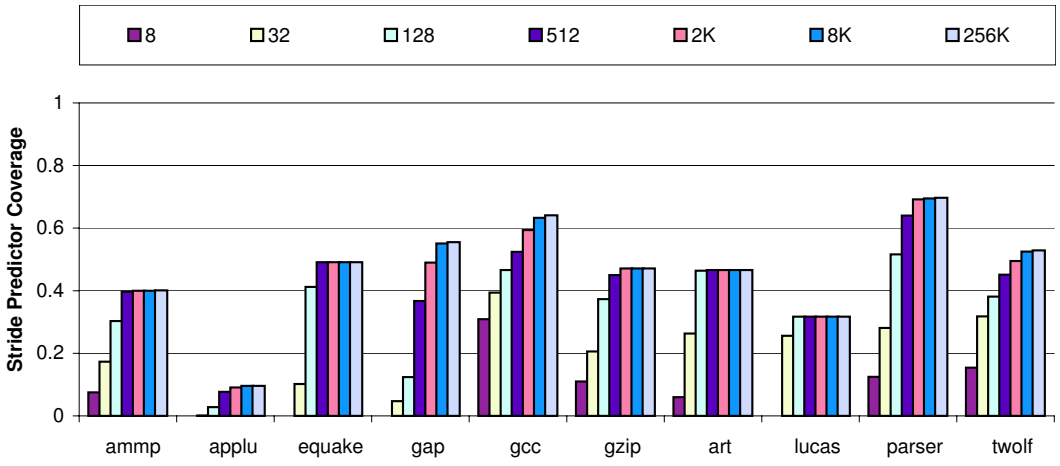


Figure 5.4: Load value prediction coverage of stride predictor for different sizes

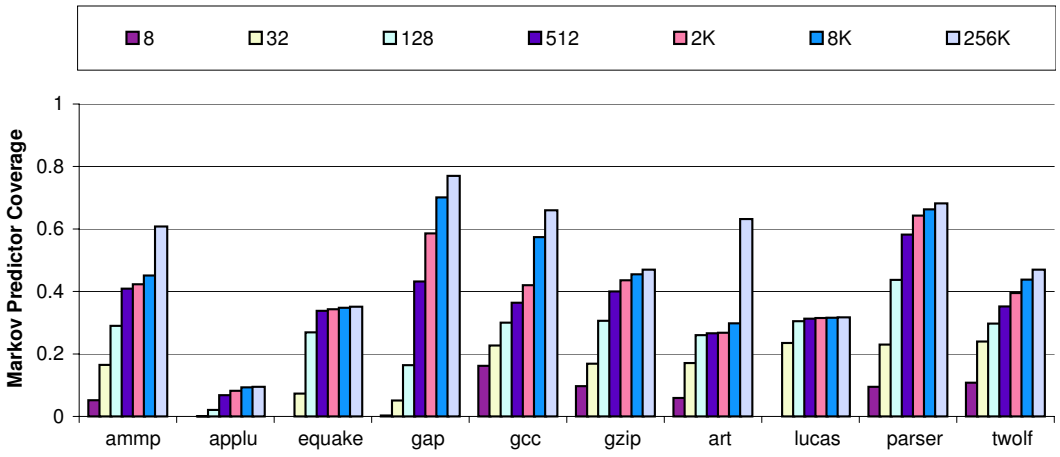


Figure 5.5: Load value prediction coverage of Markov predictor for different sizes

Another class of prefetching is Demand Based Prefetching which generates prefetches after a cache miss [Jou90] or the use of a cache block such as Next Line Prefetching [SH92]. Markov prefetching [JG97] is another example of this method. Markov predictors are used to provide the next set of possible cache addresses followed by the current miss.

Another class of hardware prefetchers are Decoupled/Stream prefetchers. In this class, the prefetcher is loosely decoupled from the instruction fetch stream and can potentially prefetch down multiple predicted streams [ALP93, Jou90]. Roth et. al [RS99] examined both a software and hardware approach for prefetching recursive data structures using such a decoupled model.

Jouppi [Jou90] introduced stream buffers to improve direct mapped cache performance. The stream buffers follow multiple streams prefetching them in parallel. Palacharla and Kessler [PK94] suggested two techniques to enhance the effectiveness of stream buffers: allocation filters and non-unit stride detection. Farkas et. al [KPN97] enhanced stream buffers by preventing duplicated prefetches across different streams and by using a PC-based stride predictor to guide the prefetched address stream.

Recently, Sherwood et. al [SSC00] proposed Predictor-Directed Stream Buffers (PSB), where they use a more sophisticated address predictor to generate the next address to prefetch. They used a hybrid stride-filtered Markov address predictor to direct stream buffer prefetching and confidence techniques to guide the allocation and prioritization of stream buffers and their prefetch requests. In this work, we model a predictor directed stream buffer similar to their work [SSC00].

In the commit stage, we update the address predictor for every load that missed in the L1 data cache. The Markov predictor is only updated if the stride predictor could not predict that load, as in [SSC00]. On every update, confidence counters are

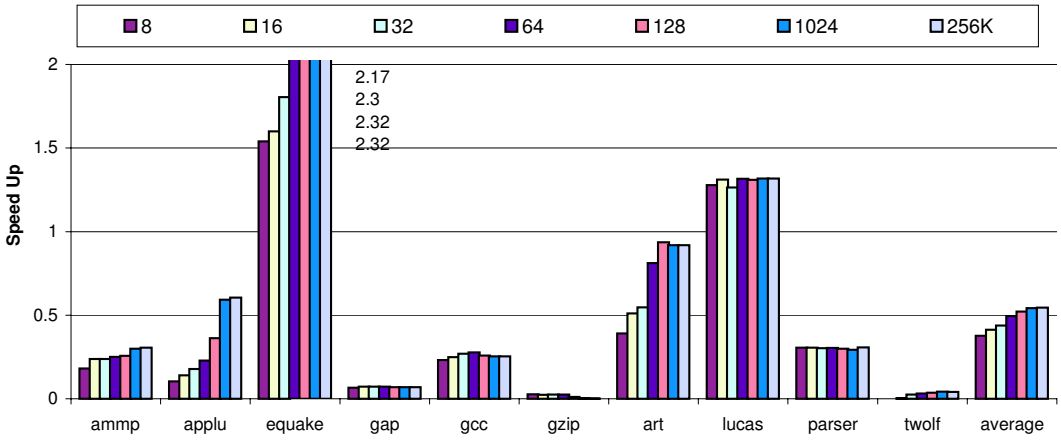


Figure 5.6: Impact of predictor size in performance of load prefetching. Speed up is over the baseline architecture.

incremented in the case of a correct prediction, and decremented otherwise. A load is only allocated a stream buffer if the stream buffer's confidence counter is less than the load's confidence, preventing re-allocation of a stream buffer with good performance. We examined using confidence counters to guide our stream buffer prediction and allocation, but did not see any improvement. We only allocate one stream buffer to given PC, if a stream buffer is already allocated to a load PC that misses, we simply update the existing stream buffer history with the new information for that PC. This ensures that stream buffers do not follow the same stream and significantly reduces the number of useless predictions.

Figure 5.6 shows the speedup over the baseline architecture for different predictor sizes. Similar to our value predictors we kept a ratio of 1:2 for stride and Markov predictors. There are eight stream buffers, each has four entries. This results indicate that load prefetching, even with a very small predictor can achieve a high speedup for most benchmarks, on average 38% for a very small predictor and 55% with large predictor. For our benchmarks we saw little to no improvement from going beyond

| Program | ammp | applu | art | equake | gap | gcc | gzip | lucas | parser | twolf |
|---|---|---|---|---|---|---|---|---|---|---|
| 50% | 19 | 65 | 12 | 25 | 50 | 98 | 9 | 5 | 59 | 22 |
| 80% | 91 | 100 | 23 | 48 | 121 | 499 | 36 | 10 | 186 | 103 |
| 90% | 110 | 139 | 44 | 61 | 148 | 807 | 49 | 12 | 294 | 162 |
| 100% | 218 | 744 | 311 | 518 | 579 | 3200 | 282 | 25 | 1587 | 798 |

Table 5.2: Number of static load PCs that accout for dynamic load misses

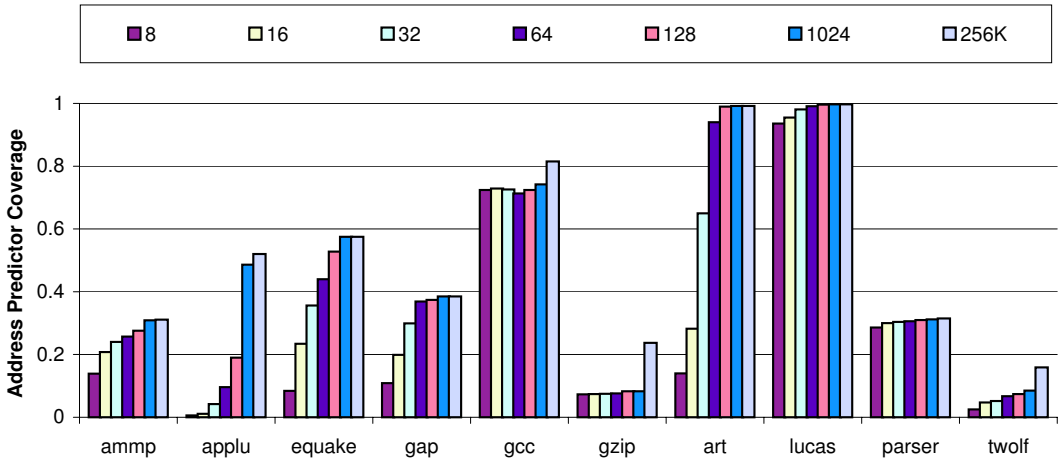


Figure 5.7: Address prediction coverage for loads that miss in DL1 in different predictor sizes

128 entry stride predictor. This can be explained by the small number of static load PCs that account for most of the misses during program execution. Table 5.2 shows how many static load PCs account for 50%, 80% and 90% of misses during execution.

Prediction coverage in figure 5.7 also agrees with our previous results on small number of load PCs that miss in first level cache and their predictable pattern. A small predictor with merely 8 enteries can capture more than 95% of all the misses in *lucas*. There is no stream thrashing and stream buffers can accurately follow the miss pattern, hense prefetching on *lucas* achieves very close to infinite cache. Another interesting

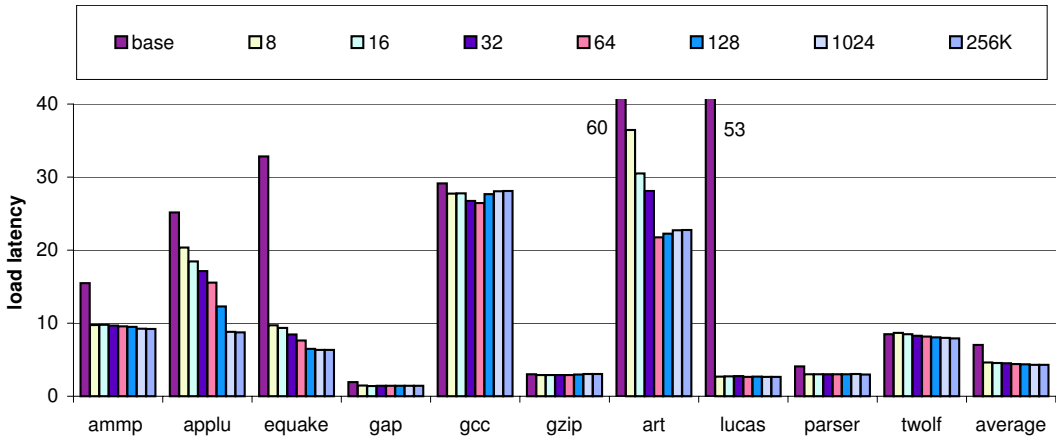


Figure 5.8: load latency without prefetching and with prefetching with different predictor sizes

benchmark is *art* that is as predictable as *lucas* with larger predictors but has poor predictability with smaller ones. Our studies indicated that there are more than 64 static load PCs that miss during execution of *art*, therefore increasing predictor size will improve coverage, but it will not prevent stream buffer thrashing. This is the reason *art* can not see the benefit of infinite cache from prefetching even with address prediction coverage of close to 100%.

Figures 5.8 and 5.9 provide more details on impact of prefetching with different predictor sizes. *Twolf* and *gizp* are not address predictable and see small improvement from prefetching, although an infinite cache can boost their performance significantly. *Gap* has a very low load latency and sees no improvement from an infinite cache nor prefetching (figure 5.1).

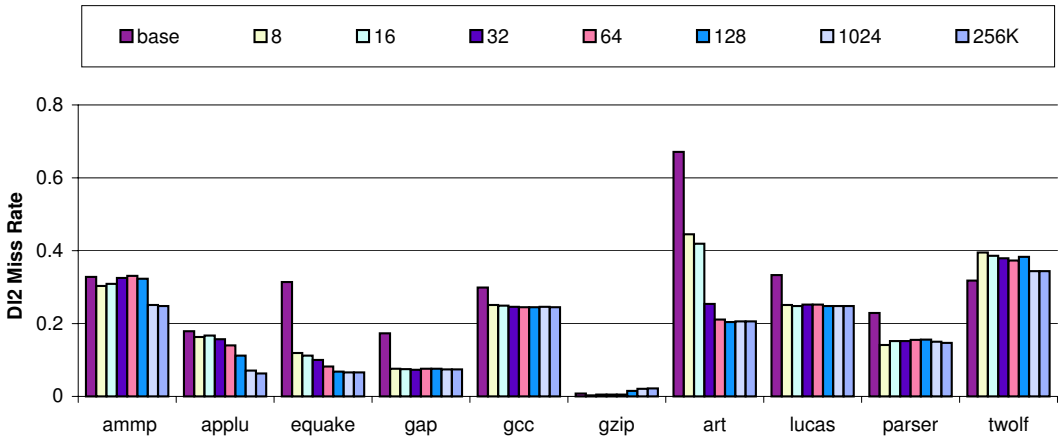


Figure 5.9: DL2 miss rate without prefetching and with prefetching with different predictor sizes

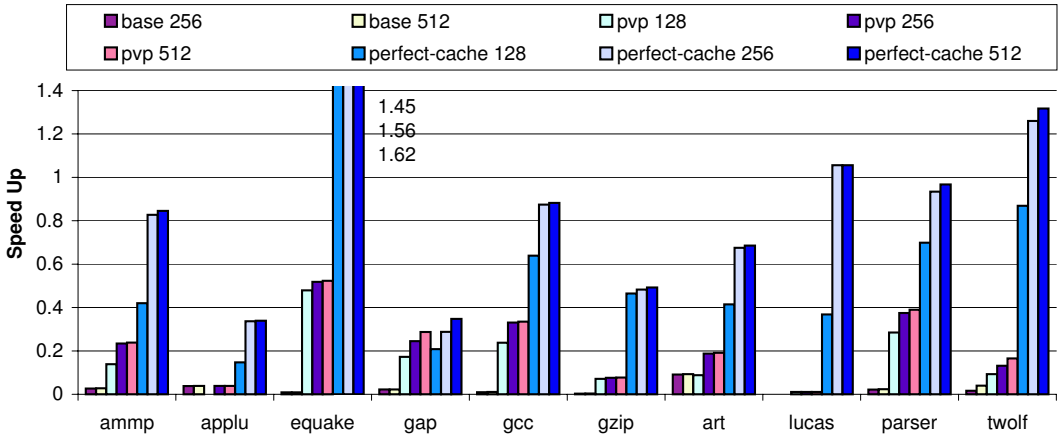


Figure 5.10: Impact of increasing instruction window size in presence of ideal value prediction and infinit data cache

## 5.4 Out Of Order Speculation

Modern high-performance processors use an out-of-order dynamic superscalar core to extract instruction-level parallelism from applications. These processors examine a large window of in-flight instructions to tolerate long latency instructions by continuing to execute independent instructions. The benefit achievable from this technique depends on the size of this window and the existing ILP in the application. Without independent instructions in the window, out-of-order speculation cannot continue to execute instructions in the presence of a long latency operation. Supporting a large window of in-flight instructions requires large structures within the processor: a large register pool to hold physical registers and a large reorder buffer. Large scalable register files have been proposed through reducing the number of required registers in the register file on the critical path [BDA01] or reducing register file port complexity [PJS97]. In this section we study the performance improvement of larger instruction windows and particularly the effect of a larger window combined with load value prediction and prefetching. Figure 5.10 shows the speedup in IPC achievable with a larger instruction window over our baseline architecture with an instruction window of 128. As can be seen from figure 5.10 increasing instruction window alone does not provide additional benefit for most benchmarks. In this study we limit the issue window to 32 entries for all ROB sizes examined.

Table 5.3 shows average reorder buffer and issue queue occupancy for different instruction window sizes. We show results for our baseline architecture, an ideal value prediction, and perfect data cache. It can be seen that with increasing instruction window size issue queue can be bottleneck for instruction level parallelism particularly with value prediction. In perfect value prediction while instruction depending on loads can execute speculatively, long latency load remaining in the issue window will eventualy clog the window and limit the benefit of larger instruction window as well. This

| Program | RUU Ocuupancy / Issue Qeueue Ocuupancy | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | base | | | perfect value prediction | | | perfect cache | | |
| RUU size | 128 | 256 | 512 | 128 | 256 | 512 | 128 | 256 | 512 |
| ammp | 74/30 | 80/31 | 81/31 | 99/23 | 161/30 | 167/30 | 43/29 | 46/29 | 51/29 |
| applu | 80/30 | 88/32 | 88/32 | 95/27 | 122/32 | 122/32 | 49/29 | 53/29 | 58/29 |
| equake | 53/31 | 56/32 | 56/32 | 102/29 | 124/31 | 131/32 | 53/31 | 53/31 | 53/31 |
| gap | 47/25 | 57/26 | 57/26 | 60/21 | 91/23 | 118/24 | 34/25 | 34/25 | 34/25 |
| gcc | 59/28 | 62/28 | 64/28 | 89/22 | 115/25 | 120/25 | 30/21 | 33/21 | 39/21 |
| gzip | 43/28 | 44/28 | 44/28 | 52/26 | 57/26 | 60/27 | 38/27 | 38/27 | 38/27 |
| art | 74/28 | 85/31 | 85/31 | 102/24 | 127/29 | 128/29 | 45/25 | 50/25 | 54/23 |
| lucas | 39/32 | 39/32 | 39/32 | 108/23 | 183/32 | 183/32 | 41/32 | 41/32 | 41/32 |
| parser | 53/27 | 58/28 | 58/28 | 70/21 | 97/24 | 107/25 | 36/25 | 36/25 | 36/25 |
| twolf | 55/28 | 62/28 | 65/29 | 84/20 | 137/24 | 144/25 | 35/25 | 35/25 | 35/25 |

Table 5.3: Reorder Buffer and Issue Queue occupancy for different sizes of instruction window, an ideal value prediction and perfect data cache

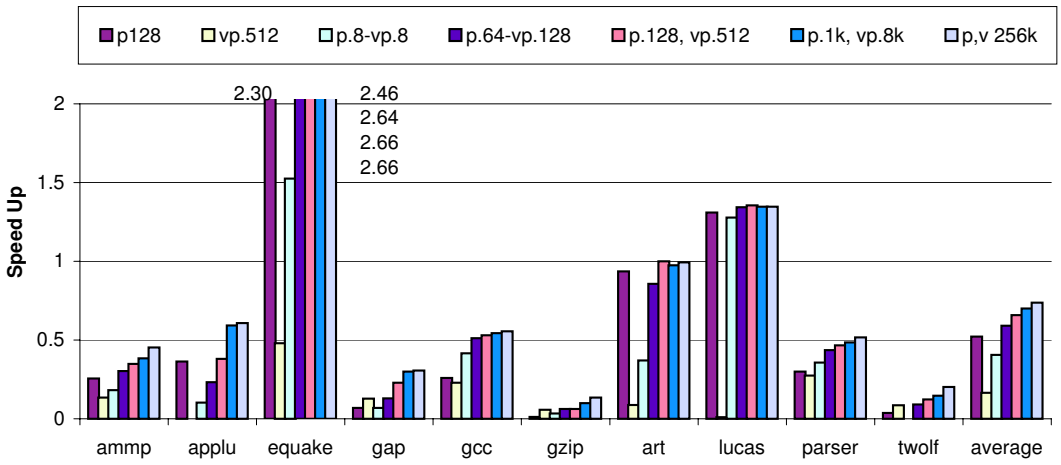


Figure 5.11: Perfromance of different configurations of combination of value prediction and load prefetching.

explains why reorder buffer occupancy does not increase much beyond 256-entry RUU and there is little improvement in performance for that increase.

Perfect cache on the other hand removes load instructions as well as their dependent instructions from the issue queue and provides more performance benefit from larger window. This can be seen from the jump in performance of perfect cache when increasing window size from 128 to 256.

## 5.5 Interaction between Value Predction and Load Prefetching

Figure 5.11 shows speed up over baseline architecture for comibnation of value prediction and load prefetching. We examined five different combination of address and value predictors. The first two bars show results of using only prefetcher or value prediction. All other results show combination of these two techniques using different predictor sizes. The results presented are for instruction window of 128. As mentioned before these results indicate that value prediction and prefetching attack differ-

ent sources of latency. Some application, such as *gcc* see better performance from a small value predictor combined with a small load prefetcher (p.8, vp.8) compared to a relatively larger value predictor or prefetcher alone (p.128 or vp.512). Other applications such as *equake* and *lucas* only see benefit from prefetcher. These applications either hard to value predict or exhibit little instruction level parallelism.

These results suggest that given a certain chip area or transistor budget, applications require different resources to deliver their best performance. While some applications use both value predictor and prefetcher, others might see best performance from value prediction or prefetching using a larger predictor. In order to achieve best performance for all applications future designs should study predictors that can dynamically be reconfigured as value or address predictors.

# CHAPTER 6

# Conclusion and Future Directions

In this thesis, we have proposed a factored architecture that is enhanced by a number of different helper engines. The critical pipeline structures of instruction fetch, branch prediction, data memory access, and the register file have all been hierarchically extended, and the value predictor, data prefetch engine, and commit hardware have been completely decoupled from the factored core. This new $\mu$-core architecture is able to reduce total processor power dissipation by 20% on average, while it attains comparable or better performance than a deeply pipelined monolithic design at the same clock frequency. This power gain does not take into account further possible reduction in routing complexity and wire delay that can be achieved by reducing the size of structures in the critical pipeline core. Moreover, flexibility of $\mu$-core architecture opens the door to further energy-saving techniques that can be applied to the more latency tolerant helper engine structures.

By dynamically configuring the helper engines to different application phases, an additional 13% power savings can be attained with only an average 3% degradation in performance. The complexity of this reconfiguration can be completely decoupled from the $\mu$-core itself.

A counter-based approach to configure helpers is discussed. We explore the helper management policies for both single and multicore configurations. With this approach, we can come within 2% of the best performing helper configuration (typically the one with all five helpers active), with an average of less than three helpers turned on. This

counter-guided scheme can be applied to the multicore environment to more effectively share a pool of common helpers among a number of cores.

Our approach to sharing is intelligent and flexible enough when used with four cores sharing two sets of helpers to see an average 54% weighted speedup over a baseline naively sharing only one set of helpers. Statically shared helpers can see benefit performance by an average of 20% to 40% depending on how cores are shared. Constructive sharing can provide even more benefit, effectively providing performance comparable to private helpers when running the same application on all cores, even for different inputs and phases.

In addition, we have explored the use of core swapping on our $\mu$-core architecture for thermal management. Core swapping is complicated by two factors: the cost of migrating an application from one core to another and the area overhead of the additional core for thermal management. Microcores enable efficient core swapping by buffering processor state in shared helper engines that reduce startup costs when switching to a new core, and they are small enough to reduce the area overhead of core replication.

Our results demonstrate that our microcore reduces the impact of core swapping significantly, on average by 29% while showing promising thermal reduction ability. It also has favorable performance (as measured in BIPS) when compared to other DTM techniques such as GCG and an idealized version of DFS.

Additionally, we evaluated alternative approaches to spending the area overhead of the additional microcore, including larger microcores, CMP cores, and SMT cores, all with DFS. Our results indicate that while additional core on CMP and larger resources on SMT can improve performance, if the processor heats up, even an idealized version of DFS can hurt their performance significantly. Core swapping has the smallest performance overhead of only 1% on average.

Finally we present a case study of two of our helpers: value prediction and prefetch-

ing. We study impact of predictor size on benefit attained from these techniques. Further we explore how these techniques interact with each other and other parts of the processor to attack load latency in a modern processor.

# REFERENCES


[ABP01] A. Ajami, K. Banerjee, M. Pedram, and L. van Ginneken. "Analysis of non-uniform temperature-dependent interconnect performance in high performance ICs." In *41st Design Automation Conference*, pp. 567–572, June 2001.

[AGG03] J.L. Aragon, J. Gonzalez, and A. Gonzalez. "Power-aware Control Speculation through Selective Throttling." In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, February 2003.

[AHK00] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures." In *27th Annual International Symposium on Computer Architecture*, 2000.

[Alb99] D. Albonesi. "Selective Cache Ways: On-Demand Cache Resource Allocation." In *32nd International Symposium on Microarchitecture*, November 1999.

[ALP93] A.Berrached, L.Coraor, and P.Hulina. "A Decoupled Access/Execute Architecture for efficient access of structured data." In *In the Hawaii International Conference on System Services*, January 1993.

[Aus99] T. M. Austin. "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design." In *32nd International Symposium on Microarchitecture*, pp. 196–207, December 1999.

[BA97] D. C. Burger and T. M. Austin. "The SimpleScalar Tool Set, Version 2.0." Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, June 1997.

[BAB00] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures." In *33rd International Symposium on Microarchitecture*, pp. 245–257, 2000.

[BCB00] D. Brooks, P. Cook, P. Bose, S. Schuster, H. Jacobson, P. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, and M. Gupta. "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors." In *IEEE Micro*, November 2000.

[BDA01] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. "Reducing the complexity of the register file in dynamic superscalar processors." In *Proceed-*

*ings of the 34th Annual International Symposium on Microarchitecture*, December 2001.

[BJB98a] B.Rychlik, J.Faistl, B.Krug, A.Y.Kurland, J.J.Sung, M.N.Velev, and J.P.Shen. "Efficient and Accurate Value Prediction using Dynamic Classification." In *Technical Report, Carnegie Mellon University*, 1998.

[BJB98b] B.Rychlik, J.Faistl, B.Krug, and J.P.Shen. "Efficancy and Performance Impact of Value Prediction." In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1998.

[BL02] R. Bhargava and L.K.John. "Latency and Energy Aware Value Prediction for High-Frequency Processors." In *Proceedings of the 16th International Conference on Supercomputing (ICS)*, 2002.

[BM00] D. Brooks and M. Martonosi. "Adaptive thermal management for high-performance microprocessors." In *Workshop on Complexity Effective Design*, June 2000.

[BNM02] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. W. Hwu. "Vacuum Packing: Extracting Hardware-Detected Program Phases for Post-Link Optimization." In *35th International Symposium on Microarchitecture*, December 2002.

[Boh95] M. Bohr. "Interconnect Scaling - The Real Limiter to High-Performance ULSI." In *Tech. Dig. of the International Electron Devices Meeting*, pp. 241–244, December 1995.

[Boh98] M. Bohr. "Silicon trends and limits for advanced microprocessors." *Communications of the ACM*, **41**(3):80–87, March 1998.

[BS00] J.A. Butts and G.S. Sohi. "A static power model for architects." In *27th Annual International Symposium on Computer Architecture*, pp. 191–201, June 2000.

[BTM00] D. Brooks, V. Tiwari, and M. Martonosi. "Wattch: A framework for architectural-level power analysis and Optimization." In *27th Annual International Symposium on Computer Architecture*, pp. 83–94, June 2000.

[BTM02] E. Borch, E. Tune, S. Manne, and J. Emer. "Loose Loops Sink Chips." In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, 2002.

[Bur99] B.G.Z.M. Burtscher. "Prediction Outcome History-Based Confidence Estimation for Load Value Prediction." In *Journal of Instruction-Level Parallelism, vol.1*, May 1999.

[CB92] T.F. Chen and J.L. Baer. "Reducing Memory Latency via Non-Blocking and Prefetching Caches." In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 51–61, October 1992.

[CB95] T-F. Chen and J-L. Baer. "Effective hardware-based data prefetching for high performance processors." *IEEE Transactions on Computers*, **5**(44):609–623, May 1995.

[CRT99] B. Calder, G. Reinman, and D.M. Tullsen. "Selective Value Prediction." In *26th Annual International Symposium on Computer Architecture*, pp. 64–74, June 1999.

[CS00] Yuan Chou and John Paul Shen. "Instruction path coprocessors." In *The 27th Annual International Symposium on Computer architecture*, pp. 270–281, New York, NY, USA, 2000. ACM Press.

[DM01] D.Brooks and M.Martonosi. "Dynamic thermal management for high-performance microprocessors." In *International Symposium on High-Performance Computer Architecture (HPCA-7)*, pp. 171–182, January 2001.

[DM04] J. Donald and Margaret Martonosi. "Temperature-Aware Desing Issues for SMT and CMP Architectures." In *2004 Workshop on Complexity-Effective Desin*, 2004.

[DS02a] A. Dhodapkar and J. E. Smith. "Dynamic Microarchitecture Adaptation via Co-Designed Virtual Machines." In *International Solid State Circuits Conference*, February 2002.

[DS02b] A. Dhodapkar and J. E. Smith. "Managing Multi-Configuration Hardware via Dynamic Working Set Analysis." In *29th Annual International Symposium on Computer Architecture*, May 2002.

[DS02c] R. Dolbeau and A. Seznec. "CASH: Revisiting hardware sharing in single-chip parallel processor." Technical Report IRISA Report 1491, IRISA, November 2002.

[DS03] A. S. Dhodapkar and J. E. Smith. "Comparing Program Phase Detection Techniques." In *36th International Symposium on Microarchitecture*, December 2003.

[EKD03] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, and T. Mudge. "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation." In *36th International Symposium on Microarchitecture*, December 2003.

[FCJ97] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. "The Multicluster Architecture: Reducing Cycle Time Through Partitioning." In *International Symposium on Microarchitecture*, 1997.

[FS96] M. Franklin and G. S. Sohi. "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References." *IEEE Transactions on Computers*, **46**(5), May 1996.

[GBC01] S. Gunther, F. Binns, D. Carmean, and J. Hall. "Managing the impact of increasing microprocessor power consumption." In *Intel Technology Journal Q1*, 2001.

[GG98] J. Gonzalez and A. Gonzalez. "The Potential of Data Value Speculation to Boost ILP." In *12th International Conference on Supercomputing*, 1998.

[GM96] F. Gabbay and A. Mendelson. "Speculative Execution Based on Value Prediction." EE Department TR 1080, Technion - Israel Institue of Technology, November 1996.

[HBA03] S. Heo, K. Barr, and K. Asanovic. "Reducing power density through activity migration." In *International Symposium on Low Power Electronics and Design*, August 2003.

[HNO97] L. Hammond, B. A. Nayfeh, and K. Olukotun. "A single-chip multiprocessor." *IEEE Computer*, **30**, 1997.

[HRJ03] M. Huang, J. Renau, and J.Torrellas. "Positional Adaptation of Processors: Application to Energy Reduction." In *30th Annual International Symposium on Computer Architecture*, June 2003.

[HS00] Timothy H. Heil and James E. Smith. "Concurrent Garbage Collection UsingHardware-Assisted Profiling." In *ISMM*, pp. 80–93, 2000.

[HSU01] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. "The Microarchitecture of the Pentium 4 Processor." *Intel Technology Journal Q1*, 2001.

[IM01] A. Iyer and D. Marculescu. "Power aware microarchitecture resource scaling." In *Proceedings of the DATE 2001 on Design, automation and test in Europe*, pp. 190–196, 2001.

[itr03] In *International Technology Roadmap for Semiconductors*, 2003.

[JG97] D. Joseph and D. Grunwald. "Prefetching Using Markov Predictors." In *24th Annual International Symposium on Computer Architecture*, June 1997.

[Jou90] N. Jouppi. "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers." In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[KAD04] Partha Kundu, Murali Annavaram, Trung Diep, and John Shen. "A case for shared instruction cache on chip multiprocessors running OLTP." *SIGARCH Comput. Archit. News*, **32**(3):11–18, 2004.

[Kan02] Michael Kanellos. "At Intel, The Chip With Two Brains." In *CNET News.com*, August 2002.

[KGM97] J. Kin, M. Gupta, and W. Mangione-Smith. "The Filter Cache:An Energy Efficient Memory Structure." In *IEEE International Symposium on Microarchitecture*, December 1997.

[KJR03] R. Kumar, N. Jouppi, P. Ranganathan, and D. Tullsen. "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction." In *36th International Symposium on Microarchitecture*, December 2003.

[KJT04] R. Kumar, N. Jouppi, and D. Tullsen. "Conjoined-core Chip Multiprocessing." December 2004.

[KMH03] K.Skadron, M.Stan, W. Huang, S.Velusamy, K. Sankaranarayanan, and D. Tarjan. "Temperature-aware microarchitecture." In *30th Annual International Symposium on Computer Architecture*, pp. 2–13, June 2003.

[KPN97] K.Farkas, P.Chow, N.Jouppi, and Z.Vranesic. "Memory-system Design Considerations for Dynamically-Scheduled Processors." In *24th Annual International Symposium on Computer Architecture*, June 1997.

[Kro81] D. Kroft. "Lockup-Free Instruction Fetch/Prefetch Cache Organization." In *8th Annual International Symposium of Computer Architecture*, pp. 81–87, May 1981.

[KS02] H-S. Kim and J. E. Smith. "An instruction set and microarchitecture for instruction level distributed processing." In *Proceedings of the 29th annual international symposium on Computer architecture*, pp. 71–81, June 2002.

[KSS06] E. Kursun, A. Shayesteh, S. Sair, T. Sherwood, and G. Reinman. "An Evaluation of Deeply Decoupled Cores." In *Journal of Instruction Level Parallelims*, volume 8, 2006.

[KTR04] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farakas. "Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance." June 2004.

[LBH05] Y. Li, D. Borrks, Z. Hu, and K. Skadron. "Performance, Energy, and Thermal Considerations for SMT and CMP Architecture." In *International Symposium on High-Performance Computer Architecture (HPCA-2005)*, 2005.

[LDG02] C-H. Lim, W. Daasch, and G.Cai. "A thermal-aware superscalar microprocessor." In *International Symposium on Quality Electronic Design*, pp. 517–522, March 2002.

[LM98] C.-K. Luk and T. C. Mowry. "Cooperative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors." In *31st International Symposium on Microarchitecture*, December 1998.

[LR01] L.T.Yeh and R.Chu. "Thermal management of microelectronic equipment." In *American Society of Mechanical Engineers - ISBN:0791801683*, 2001.

[LS96a] M. Lipasti and J. Shen. "Exceeding the Dataflow Limit via Value Prediction." In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1996.

[LS96b] M.H. Lipasti and J.P. Shen. "Exceeding the Dataflow Limit via Value Prediction." In *29th International Symposium on Microarchitecture*, December 1996.

[LSM05] Y. Liu, A. Shayesteh, G. Memik, and G. Reinman. "Tornado Warning: the Perils of Selective Replay in Multithreaded Processors." In *Proceedings of the 18th International Conference on Supercomputing (ICS)*, June 2005.

[LWS96a] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. "Value Locality and Load Value Prediction." In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138–147, October 1996.

[LWS96b] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. "Value Locality and Load Value Prediction." In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[McF93] S. McFarling. "Combining Branch Predictors." Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[MTB01] M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhaal, and Wen mei W. Hwu. "An Architectural Framework for Run-Time Optimization." *IEEE Transactions on Computers*, **50**(6):567–589, June 2001.

[Mud00] T. Mudge. "Power: A First Class Design Constraint for Future Architecture and Automation." In *Proceedings of the 7th International Conference on High Performance Computing*, 2000.

[PGV04] M.D. Powell, M. Gomaa, and T.N. Vijaykumar. "Heat-and-Run:Leveraging SMT and CMP to Manage Power Density Through the Operating System." In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[PJS97] S. Palacharla, N. P. Jouppi, and J. E. Smith. "Complexity-Effective Superscalar Processors." In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 206–218, June 1997.

[PK94] S. Palacharla and R. Kessler. "Evaluating stream buffers as secondary cache replacement." In *21st Annual International Symposium on Computer Architecture*, April 1994.

[Pow] Power4. In *IBM. Power4:http://www.research.ibm.com/power4*.

[RAC99] G. Reinman, T. Austin, and B. Calder. "A Scalable Front-End Architecture for Fast Instruction Delivery." In *26th Annual International Symposium on Computer Architecture*, May 1999.

[RAJ00] P. Ranganathan, S. V. Adve, and N.P. Jouppi. "Reconfigurable caches and their application to media processing." In *27th Annual International Symposium on Computer Architecture*, pp. 214–224, June 2000.

[RLP00] R.Moreno, L.Pinuel, S.del Pino, and F.Tirado. "A Power Perspective of Value Speculation for Superscaler microprocessors." In *Proceedings of the 2000 International Conference on Computer Design (ICCD)*, September 2000.

[RS99] A. Roth and G. Sohi. "Dependence based prefetching for linked data structures." In *26th Annual International Symposium on Computer Architecture*, June 1999.

[SC99] T. Sherwood and B. Calder. "Time Varying Behavior of Programs." Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[SC02] E. Sprangle and D. Carmean. "Increasing Processor Performance by Implementing Deeper Pipelines." In *29th Annual International Symposium on Computer Architecture*, 2002.

[SD03] S.Ghiasi and D.Grunwald. "Thermal Management With Asymmetrical Dual Core Designs." Technical Report CU-CS-965-03, University of Colorado, 2003.

[SH92] J. E. Smith and W.-C. Hsu. "Prefetching in Supercomputer Instruction Caches." In *Proceedings of Supercomputing*, November 1992.

[SJ01] P. Shivakumar and Norman P. Jouppi. "CACTI 3.0: An integrated cache timing, power, and area model." In *Technical Report*, 2001.

[SJL01] S. Srinivasan, R. Ju, A. R. Lebeck, and C. Wilkerson. "Locality vs. Criticality." In *28th Annual International Symposium on Computer Architecture*, June 2001.

[SJS96] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. "Multiple-Block Ahead Branch Predictors." In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 116–127, October 1996.

[Smi01] J. E. Smith. "Instruction-Level Distributed Processing." *IEEE Computer*, **34**(4):59–65, April 2001.

[SMT]

[SNL03] K. Sankaralingam, R. Nagarajan, H. Liu, J. Huh, C.K. Kim, D. Burger, S.W. Keckler, and C.R. Moore. "Exploiting ILP, TLP, and DLP Using Polymorphism in the TRIPS Architecture." In *30th Annual International Symposium on Computer Architecture*, June 2003.

[SP85] James E. Smith and Andrew R. Pleszkun. "Implementation of Precise Interrupts in Pipelined Processors." In *12th Annual International Symposium on Computer Architecture*, 1985.

[SPC01] T. Sherwood, E. Perelman, and B. Calder. "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications." In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[SPH02] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. "Automatically Characterizing Large Scale Program Behavior." In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[SRP97] J. Stark, P. Racunas, and Y. N. Patt. "Reducing the Performance Impact of Instruction Cache Misses by Writing Instructions into the Reservation Stations Out-of-Order." In *30th International Symposium on Microarchitecture*, pp. 34–43, December 1997.

[SS97] Yiannakis Sazeides and James E. Smith. "The Predictability of Data Values." In *30th International Symposium on Microarchitecture*, 1997.

[SS98] Y. Sazeides and J. E. Smith. "Modeling Program Predictability." In *25th Annual International Symposium on Computer Architecture*, June 1998.

[SSC00] T. Sherwood, S. Sair, and B. Calder. "Predictor-Directed Stream Buffers." In *33rd International Symposium on Microarchitecture*, December 2000.

[SSC03] T. Sherwood, S. Sair, and B. Calder. "Phase Tracking and Prediction." In *30th Annual International Symposium on Computer Architecture*, June 2003.

[SSH00] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. "Temperature-aware microarchitecture." In *Proceedings of the 30th annual international symposium on Computer architecture*, June 200.

[ST00] A. Snavely and D. M. Tullsen. "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor." In *In Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[SYP00] S.Lee, Y.Wang, and P.Yew. "Decoupled Value Prediction on Trace Processors." January 2000.

[TEL95] Dean Tullsen, Susan Eggers, and Henry Levy. "Simultaneous Multithreading: Maximizing On-Chip Parallelism." In *Proceedings of the 22rd Annual International Symposium on Computer Architecture (ISCA)*, June 1995.

[TMA92] T.C.Mowry, M.S.Lam, and A.Gupta. "Design and Evaluation of a Compiler Algorithm for Prefetching." In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS-V)*, October 1992.

[VWW00] R. Viswanath, V. Wakharkar, A. Wathe, and V.Lebonheur. "Thermal performance challenges from silicon to systems." In *Intel Technology Journal Q3*, 2000.

[WF97a] K. Wang and M. Franklin. "Highly Accurate Data Value Prediction using Hybrid Predictors." In *30th Annual International Symposium on Microarchitecture*, pp. 281–290, December 1997.

[WF97b] K. Wang and M. Franklin. "Highly Accurate Data Value Prediction using Hybrid Predictors." In *30th Annual International Symposium on Microarchitecture*, December 1997.

[Wil95] M.V. Wilkes. "The Memory Wall and the CMOS End-Point." In *ACM Computer Architecture News, Vol.23, No.4*, 1995.

[WJS00] W.Huang, J.Renau, S-M.Yoo, and J. Torrellas. "A framework for dynamic energy effiency and temperature management." In *33rd International Symposium on Microarchitecture*, pp. 202–213, December 2000.

[WS95] Wm.A. Wulf and S.A.Mckee. "Hitting the Memory Wall: Implications of the Obvious." In *ACM Computer Architecture News, Vol.23, No.1*, 1995.

[YP92] T. Yeh and Y. Patt. "A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution." In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 129–139, December 1992.

[ZPS03] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. "HotLeakage: A Temperature-aware model of subthreshold and gate Leakage for architects." In *University of Virginia Dept of Computer Science Tech Report CS-2003-05*, March 2003.