Dynamic Management of Microarchitecture Resources in Future Microprocessors

by

Rajeev Balasubramonian

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Sandhya Dwarkadas

Department of Computer Science
The College
Arts and Sciences

University of Rochester
Rochester, New York

2003

Curriculum Vitae

Rajeev Balasubramonian was born in Pune, India on May 26, 1977. He attended the Indian Institute of Technology, Bombay, receiving the Bachelor of Technology degree in Computer Science and Engineering in 1998. He arrived in Rochester in the Fall of 1998 and has been involved in computer architecture research under the direction of Professor Sandhya Dwarkadas. He was supported in part by a University of Rochester Sproull Fellowship and an IBM Ph.D. Fellowship. He received a Master of Science degree in Computer Science in 2000. His research interests include clustered architectures, memory hierarchy bottlenecks, and complexity-effective designs.

Acknowledgments

I have been very fortunate to learn about computer architecture and academic life from wonderful teachers like Sandhya Dwarkadas and Dave Albonesi. I am eternally grateful to Sandhya for having motivated me through the tough times and for having given me the independence that made the journey so enjoyable. I am indebted to Dave for enthusiastically participating in and steering the direction of all my projects. I thank my committee members, Michael Scott and Chen Ding, for many helpful discussions and the valuable feedback that has helped shape my dissertation. I also thank Pradip Bose and Viji Srinivasan at IBM for helping me through and beyond my internship. I thank the co-authors in my papers, especially Alper Buyuktosunoglu, who was responsible for the circuit-level analysis in the design of the reconfigurable cache. I am grateful to IBM, the University of Rochester, NSF, and DARPA, that have funded my stint in grad school.

I thank my friends who helped me preserve my sanity – the Tribals, the TC Thugs, Ali Rangwala – my classmates at UR, who propped me up in my first year – Isaac Green, who failed miserably at getting me thrown out of grad school :-) – the systems and architecture groups, especially Grigoris, Alper, and Srini, for many stimulating discussions – Umit, for tolerating my pesky simulations – Rob Stets, for having convinced me to join UR and for having been the perfect role model – Brandon, my hockey, golf, and basketball buddy – the highly-efficient URCS staff – Thakur, for being a perfect roommate and accomplice in all crimes.

I am grateful to my family for their support in all my endeavors. My brother, Rajesh, was a great role model and sparked my competitive fire. Many thanks to my wife, Deepthi, whose love and support helped me breeze through a gruelling final year in grad school. My parents deserve all credit for instilling the spirit of discovery and academics in me. Their love and encouragement has been instrumental in my success at every level.

Abstract

Improvements in technology have resulted in steadily improving microprocessor performance. However, the shrinking of process technologies and increasing clock speeds introduce new bottlenecks to performance, viz, long wire delays on the chip and long memory latencies. We observe a number of trade-offs in the design of various microprocessor structures and the gap between the different trade-off points only widens as technologies improve and latencies of wires and memory increase. The emergence of power as a first-order design constraint also introduces trade-offs involving performance and power consumption. Microprocessor designs are optimized to balance these trade-offs in the average case, but are highly sub-optimal for most programs that run on the processor. The dissertation evaluates hardware reconfiguration as a means to providing a program with multiple trade-off points, thereby allowing the hardware to match the program's needs at run-time. In all cases, hardware reconfiguration exploits technology trends and is relatively non-intrusive.

We examine a reconfigurable cache layout that varies the L1 data cache size and helps handle the trade-off between cache capacity and access time. We also study a highly clustered and communication-bound processor, where a subset of the total clusters yields optimal performance by balancing the extraction of distant parallelism with the inter-cluster communication costs. In a processor with limited resources, distant parallelism can be mined with the help of a pre-execution thread and the allocation of resources between the primary and pre-execution thread determines the trade-off between nearby and distant parallelism. In all of these cases, the dynamic management of on-chip resources can balance the different trade-offs. We propose and evaluate dynamic adaptation algorithms that detect changes in program behavior and select optimal hardware configurations. Our results demonstrate that the adaptation algorithms are very effective in adapting to changes in program behavior, allowing improved processor efficiency through hardware reconfiguration. Performance is improved and power consumption is reduced when compared with a static hardware design.

Table of Contents

List of Tables

List of Figures

1 Introduction

Microprocessors today are employed for a wide range of applications, including servers, desktops, laptops, palm-pilots, automobiles, washing machines, etc. Most microprocessors used in low-end devices (home appliances) have relatively simple designs and are not very sensitive to clock speed and performance specifications. However, performance is the primary concern for most high-end microprocessors designed today. Microprocessor performance has been steadily improving because of innovations in microarchitectural design and because of shrinking process technologies. The reduction in transistor feature sizes has increased transistor speeds and chip capacity, but has also introduced new problems in the design of modern processors. Faster transistors have enabled faster clocks, but wire delays have not improved at the same rate. Since many microprocessor structures are communication-bound rather than compute-bound, this seriously impacts the instruction-level parallelism (ILP) that these structures can help extract. Higher chip capacity and the growing number of structures on the chip also increases power consumption and design and verification complexity.

This dissertation studies the effect of recent technology trends and how they introduce trade-offs in the design of different structures on the processor. We show that by reconfiguring the hardware organization at run-time, the processor can provide many different trade-off points to the program it is executing. We examine on-line mechanisms to automatically match the hardware to the application's needs.

1.1 Recent Technology Trends

Early RISC microprocessors had short and simple pipelines and generally executed instructions in order. Over the past decade, architects and circuit designers have focused on maximizing performance. The following approaches have resulted in steadily improving microprocessor performance in the CMOS era [Hennessy and Patterson, 1996]:

- The feature widths of transistors have been shrinking due to improvements in process technology. This has resulted in a shrinking of logic delays, which makes it possible to implement the basic operations (integer arithmetic) within a very short cycle. The net result is an increase in clock frequency.
- The improvement in clock frequency has also been greatly helped by the implementation of deeper pipelines. By breaking up operations across multiple stages, the critical path for the cycle time can be shortened.
- There have been improvements in the number of instructions being executed in parallel, thanks to innovations in out-of-order execution, speculation, caching, etc.

These advances have resulted in the emergence of new challenges for architects:

- **Design Complexity.** To extract a high degree of instruction-level parallelism (ILP), a modern-day microprocessor employs a wide variety of specialized structures such as branch predictors, caches, issue queues, re-order buffers, etc. [Smith and Sohi, 1995]. This increases the number of pipeline stages and the number of cycle-time critical paths [Palacharla *et al.*, 1997] and results in a dramatic increase in the design and verification effort. For each of these structures, designers have to make a number of choices to balance the demands for a high clock speed and high ILP.
- **Power Consumption.** The power consumption in a processor is roughly proportional to the clock frequency and the amount of switching activity. Increased clock speeds and large numbers of structures on chip have made power consumption a first-order design constraint [Gowan *et al.*, 1998; Tiwari *et al.*, 1998].

- **Memory Wall.** The speeds of DRAM memories have not been improving at the same rate as microprocessor clock frequencies [Wulf and McKee, 1995]. As a result, the number of microprocessor clock cycles required to access main memory has been steadily increasing and represents one of the biggest bottlenecks to overall performance.

- **Dominant Wire Delays.** Reduced feature sizes can reduce logic delays, but do not reduce wire delays at the same rate [Agarwal *et al.*, 2000; Palacharla *et al.*, 1997]. The resistance per unit length of a wire increases roughly as a quadratic function of the reduction in feature size, due to the reduction in its cross-sectional area. At the same time, the capacitance per unit length also increases due to an increase in the coupling capacitance between neighboring wires. The result of these trends is that the delay of a wire of fixed length increases as we decrease the feature widths. While the wire length for a microprocessor structure is likely to scale down roughly linearly with feature size, the clock cycle time also scales down roughly linearly. The net result is that the delay of that wire *in cycles* increases in future microprocessors. Hence, more cycles within the program's execution are spent communicating data between different parts of the processor.

1.2 Management of Trade-Offs

In the design of modern microprocessors, architects have to find solutions for all of the challenges described above. The primary emerging goals are a reduction in design complexity, power consumption, and the effect of long communication and memory latencies. Unfortunately, not all of these new emerging goals can be simultaneously optimized. This results in a proliferation of trade-offs in the design of the different structures in a microprocessor. For example, a large data cache can support a high degree of ILP by providing a high hit rate and by avoiding the memory wall. However, a large cache is composed of long wires, which implies that every access to the cache costs a number of cycles, and this could be detrimental to performance in a number of applications. A large cache consumes a large amount of power. Implementing a large cache as a multi-stage pipeline also results in high design complexity. Similarly, a large physical register file can help map a large number of in-flight instructions, which can help extract more instructions to execute in every cycle and also issue memory requests early. Unfortunately, a large register file is energy-inefficient and difficult to pipeline. The long wires in the register file also introduce cycle-time critical paths that limit the maximum attainable clock speed.

Further, because most processor components (typically storage structures) are composed of long wires, their access times (in cycles) increase with each improvement in process technology. As a result, the gap between different design alternatives for a given structure is also widening. For example, in the past, doubling the cache size would have increased the access latency by only one cycle, but in the future, this could increase the access time by three or four cycles.

Thus, current technology trends are going to necessitate a more thorough consideration of trade-offs in different parts of the processor.

Most microprocessor structures are designed to balance different trade-offs and maximize a single metric. In designing a cache, we could optimize the overall performance of the system by balancing the trade-off between cache capacity and cache access time. However, this optimization is done while evaluating an entire workload or benchmark suite. The structure design is selected based on what works best, on average, across a wide range of programs. However, as we shall observe in later chapters, different programs, and even different parts of the same program, have widely varying characteristics and display dramatically different behavior across different hardware designs. Since the gap between different trade-off points is widening, a design that might be optimal on average might be highly sub-optimal for a large number of programs within the benchmark set. We propose hardware reconfiguration to deal with the problem of hardware designs that do not match program needs. By adapting the hardware design at run-time, trade-offs can be balanced for each individual program behavior.

1.3 Thesis Statement

Our thesis is that technology trends necessitate the management of trade-offs at run-time. Technology trends also facilitate the adaptation of the hardware, allowing the processor to choose among multiple trade-off points. Simple control mechanisms that do not involve excessive hardware overhead are very effective at matching the hardware to the program's dynamic needs.

First, we demonstrate the emergence of trade-offs in a number of different settings. Second, we introduce novel techniques to adapt the hardware at run time and provide different trade-off options. These techniques exploit technology trends to provide this adaptivity in an extremely low-intrusive manner. Finally, we define and evaluate a framework of control mechanisms that can dynamically and reliably predict program characteristics and the optimal hardware organization for each distinct program phase. While we focus on maximizing performance and, in some instances, power, the framework can be employed to maximize any metric.

Our framework of control mechanisms explores design options in detecting reconfiguration points and in the techniques used to select the optimal reconfiguration. Reconfiguration points can occur at regular periodic intervals or at specific instructions. An optimal organization can be selected by profiling each of the candidate organizations or by using hardware metrics to predict the behavior of the candidates. The most reliable algorithm, which achieves most of the potential available performance, attempts reconfiguration at interval boundaries and selects the best organization by exploring the candidate organizations. A variable interval length makes this algorithm very reliable and allows it to apply to programs with dramatically different characteristics. Note that there are potentially many other dynamic adaptation techniques that could fruitfully be employed. We consider here, a set of representative techniques.

These techniques can be universally employed over a wide range of reconfigurable hardware. The first such application we study is a reconfigurable cache design [Balasubramonian *et al.*, 2000a; Balasubramonian *et al.*, 2000b; Balasubramonian *et al.*, 2003a]. The trade-off between L1 cache capacity and access time can be optimized at run time by allocating an amount of cache that is just enough to contain the program working set. The next application studies a highly clustered processor that is communication-bound [Balasubramonian *et al.*, 2002; Balasubramonian *et al.*, 2003b]. We demonstrate that the optimal number of clusters at any given time is a function of the degree of parallelism in the program. By employing only this subset of clusters, we balance the trade-off between communication and parallelism. Finally, we study how limited processor resources can be allocated across two threads, one of which attempts to jump ahead in the execution and warm up various processor structures [Balasubramonian *et al.*, 2001b; Balasubramonian *et al.*, 2001c]. There exists an optimal allocation of resources depending on whether the program is more limited by nearby or distant parallelism.

In all of these different problem domains, the best hardware organization is selected by employing the different adaptation algorithms. Our results show that these techniques are easily applicable and very effective at improving performance. We also demonstrate for a subset of the problem domains that power can be reduced in addition to performance improvements.

1.4 Dissertation Organization

The dissertation is organized as follows. Chapter 2 studies variability in program behavior and evaluates different algorithms that can predict program characteristics with little overhead. Chapter 3 describes trade-offs in the design of the data cache and presents an adaptive cache layout. The algorithms described in Chapter 2 are evaluated on this reconfigurable cache. We then discuss the emergence of trade-offs in a highly clustered and communication-bound microprocessor of the future in Chapter 4. We demonstrate that our control mechanisms are equally effective in this setting. Finally, we discuss trade-offs and their management in a microprocessor implementing pre-execution threads in Chapter 5. We conclude in Chapter 6.

2 Dynamic Adaptation Algorithms

Hardware adaptation can provide a number of options to an executing program. It allows the use of a hardware design that can balance various trade-offs and maximize a particular metric. In order to match the hardware to the program's requirements at run-time, the hardware has to estimate the optimal design at any point in the program. There are two major components in such a run-time mechanism. First, it must identify every new *program phase*. A *program phase* is a part of a program that has its own distinct behavior and differs from the other parts of the program in terms of various statistics relevant to the hardware being adapted. Second, it must efficiently determine the hardware organization that is optimal for each *program phase*. The next two sections deal with these issues in detail.

2.1 Phase Detection

At a low level, a *program phase* could be defined by its effects on the microarchitecture. For example, a variation in the data cache miss frequency could signal a new *phase*. At a higher level, each *program phase* can be considered as defined by its functionality. For example, every subroutine could be considered a new *phase*. In the first case, a *phase change* is signaled by observing hardware statistics over an interval of time. In the second case, a *phase change* is signaled by the execution of a particular instruction (a subroutine call/return or a branch).

2.1.1 Phase Changes at Interval Boundaries

A *phase* is characterized by the values of certain hardware metrics observed over a fixed interval of committed instructions. A *phase change* occurs every time these metrics vary significantly across successive intervals. Candidate metrics that could signal a phase change are the branch frequency, the cache miss rate, the frequency of memory operations, the cycles per instruction, the branch misprediction rate, the average window of in-flight instructions, etc. At the start of each phase, the values of these metrics over an interval are recorded and they serve as a reference point. After each subsequent interval, the values over that interval are compared against the reference point. If any of these values are significantly different from the reference point, a new phase is signaled. To reduce the hardware overhead of monitoring these metrics, we restrict ourselves to using only a subset of these metrics. We observed that a change in the branch frequency, or the frequency of memory operations, or the cycles per instruction were good enough to signal a new phase. We examined a wide range of statistics for each interval of execution and found that the above three metrics had a strong correlation with a change in a more exhaustive set of metrics.

Variable Interval Lengths

Once a phase change has been detected, a particular hardware organization, deemed to be optimal, is selected. This organization is used till the next phase change is detected. Since the selection of the optimal organization requires at least one interval to monitor the program behavior, during which the hardware organization might be sub-optimal, every phase change entails some overhead. Hence, a frequent occurrence of phase changes is not desirable. The success of such an interval-based mechanism is heavily reliant on the program's ability to sustain a phase over a number of intervals. Different programs are likely to encounter phase changes at different granularities. Hence, a single fixed interval length results in dramatic variations in phase change frequencies across an entire benchmark set. Consequently, for each program, it is important to use an interval length that keeps the occurrence of a phase change to a minimum.

We studied the applicability of this approach to a wide mix of program types. These programs are drawn from suites like SPEC2k (Int and FP) and the UCLA Mediabench [Lee *et al.*,

Benchmark	Input dataset
gzip (SPEC2k Int)	ref
vpr (SPEC2k Int)	ref
crafty (SPEC2k Int)	ref
parser (SPEC2k Int)	ref
swim (SPEC2k FP)	ref
mgrid (SPEC2k FP)	ref
galgel (SPEC2k FP)	ref
cjpeg (Mediabench)	testimg
djpeg (Mediabench)	testimg

Table 2.1: Benchmark set, including programs from SPEC2k and Mediabench.

1997]. The details on these programs are listed in Table 2.1.

To study the variability of program behavior over different intervals, we ran each of the programs for billions of instructions to generate a trace of various statistics at regular 10K instruction intervals [Balasubramonian *et al.*, 2003b]. We used three metrics to define a program phase - the instructions per cycle (IPC), the branch frequency, and the frequency of memory references. At the start of each program phase, the statistics collected during the first interval were used as the reference point. For each ensuing interval, if the three metrics for that interval were similar to the reference points, the interval was termed ‘stable’. If any of the three metrics was significantly different (as defined by a set of reasonable thresholds), we declared the interval as ‘unstable’ and started a new program phase. This analysis was done for various instruction interval lengths. We define the *instability factor* for an interval length as the percentage of intervals that are considered ‘unstable’, *i.e.*, the frequency of the occurrence of a phase change.

Assuming that an *instability factor* of 5% would result in tolerable overhead, Table 2.2 shows the smallest interval length that affords an *instability factor* of less than 5% for each of our programs. As can be seen, the interval lengths that emerge as the best vary from 10K to 40M. We also show the *instability factor* for a fixed interval length of 10K instructions. Clearly, this interval length works poorly for a number of programs and would result in quite unacceptable performance. Most programs usually show consistent behavior across intervals

Benchmark	Minimum acceptable interval length and its *instability factor*	*Instability factor* for a 10K instruction interval
gzip	10K / 4%	4%
vpr	320K / 5%	14%
crafty	320K / 4%	30%
parser	40M / 5%	12%
swim	10K / 0%	0%
mgrid	10K / 0%	0%
galgel	10K / 1%	1%
cjpeg	40K / 4%	9%
djpeg	1280K / 1%	31%

Table 2.2: *Instability factors* for different interval lengths.

for a coarse enough interval length, making interval-based schemes very robust and universally applicable. Even a program like *parser*, whose behavior varies dramatically based on the input data, has a low *instability factor* for a large 40M instruction interval.

In addition to selecting the optimal hardware organization at any point in the program, the hardware has to also select an interval length that allows a low *instability factor*. This is achieved at run-time with a simple mechanism. We start with the minimum instruction interval length. A counter is incremented or decremented after every interval to estimate the instability factor. When this counter value exceeds a certain threshold, the interval length is doubled. This process repeats until we either experience a low *instability factor* or until we reach a pre-specified limit (say, a billion instructions). If we reach the limit, we cease to employ the selection algorithm and pick the configuration that was picked most often.

Once we pick an interval length, we need not remain at that interval length forever. The program might move from one large *macrophase* to another that might have a completely different optimal instruction interval. To deal with this, we can continue to hierarchically build phase detection algorithms. An algorithm that inspects statistics at a coarse granularity (say, every 100 billion instructions) could trigger the detection of a new *macrophase*, at which point, we would restart the selection algorithm with a 10K interval length and find the optimal interval length all over again.

Thus, the interval-based mechanism is very robust – it applies to every program in our benchmark set as there is usually a coarse enough interval length such that behavior across those intervals is fairly consistent. However, the downside is the inability to target relatively short phases. We experimented with smaller initial interval lengths, but found great instability at these small interval lengths, and hence, the interval lengths were increased to a larger value just as before. This is caused by the fact that measurements become noisier as the interval size is reduced and it is harder to detect the same program metrics across intervals.

Adaptive Thresholds

In some programs, successive instances of the same phase might be separated by a short period of time, during which the behavior is slightly different. This triggers a brief period of instability, after which the same optimal configuration is selected again. In other programs, the mismatch between the length of a phase and the interval length might result in occasional noisy measurements that might trigger a phase change. To reduce the occurrence of these phenomena, the thresholds triggering a phase change can be adapted dynamically. Every phase change that results in the selection of the same optimal configuration as before increases the value of the phase change thresholds. These thresholds can be gradually reduced over a long period of time. Such a technique eliminates the need for pre-defined fixed thresholds that might be tuned for a certain benchmark set. Note that such a phenomenon is likely to not result in heavy overhead if the interval length is already selected to maintain a low instability factor.

2.1.2 Positional Adaptation

Phase changes can be signaled every time the program passes through specific locations in the code [Balasubramonian *et al.*, 2000a; Balasubramonian *et al.*, 2000b]. Recent work by Huang et al [Huang *et al.*, 2003] refers to this as *positional adaptation*. Every branch or subroutine, or even every instruction could be considered a new phase. If the behavior following a specific instruction is often consistent, this behavior could be monitored to predict the optimal hardware organization to be used every time this instruction is encountered. These predictions can be recorded in a table indexed by the program counter value.

The primary advantage from this mechanism is the ability to react as soon as there is a transition to new behavior – the interval-based mechanisms have to wait for interval boundaries before they can react. Further, every new phase is signaled by a single instruction. This instant recognition of the phase and the use of a prediction table ensure that a phase transition results in almost no overhead, allowing hardware reconfigurations at a very fine granularity. With the interval-based schemes, the recognition of a new phase requires at least one interval.

Very frequent reconfigurations might have to be discouraged if boundary effects influence the process of monitoring the behavior for a particular phase. Frequent reconfigurations are also limited by the ability of the hardware to adapt quickly. The rate at which reconfigurations happen can be tuned by allowing reconfigurations only in certain cases. For example, reconfigurations could be allowed at only every 5th branch, or only at subroutine call and returns for long subroutines, or only at specific instructions indicated by the compiler. To reconfigure at every N^{th} branch [Balasubramonian *et al.*, 2003b], a single hardware counter is enough to detect reconfiguration points. Reconfiguration for long subroutines requires that we gather statistics for each subroutine. For example, to measure the cycles spent in each subroutine, a stack is used to checkpoint the cycle count every time a subroutine is entered. While exiting the subroutine, an inspection of the stack value reveals the number of cycles spent in that subroutine and its callees. In order to exclude the cycles spent in the callees, the stack can instead checkpoint the number of cycles spent in each subroutine. Every time a subroutine invokes a callee, the number of cycles spent in that subroutine is pushed on the stack. When the callee returns, the value on the stack is popped. This value reflects the number of cycles spent so far in that subroutine and this value continues to be incremented until it returns or invokes another callee. Given this information, we can keep track of the long subroutines in the program and only create entries for them in the prediction tables.

We found that the behavior was more stable if we included the callees while gathering information for a subroutine. In case of a recursive program, statistics are recorded for only the outermost invocation of a subroutine. If the stack used to checkpoint statistics overflows, we assume that subroutines will be unable to update the table. This mechanism entails additional hardware, primarily to maintain a stack of cycle counts and subroutine call PCs. This is in addition to the table that maintains information on configuration selection (explored configurations, predictions, etc.) for each candidate subroutine.

While the experiments in the earlier subsection were able to roughly estimate the frequency of phase changes (and hence, the potential reconfiguration overhead), it is harder to carry out a similar evaluation for positional adaptation. While a phase transition can be signaled at every instruction, hardware reconfiguration will occur only if two successive phases have different optimal hardware organizations. Hence, the evaluation can not be carried out independently of the particular hardware adaptation being considered.

2.2 Configuration Selection

Once a phase has been identified, the hardware has to employ the organization that is optimal. The selection of this organization can be performed in one of many ways.

2.2.1 Exploration

The most reliable way of selecting the optimal organization is to implement the candidate organizations and select the best based on actual hardware numbers. While such an *exploration* ensures that the optimal is always selected (assuming that current behavior reflects the near future), it incurs additional overhead. If employed very often, it can result in a large portion of the execution being spent in organizations that are sub-optimal.

With the interval-based mechanism, every time a new phase is signaled, for the next few intervals, the hardware can employ a new candidate organization, one for each interval. Based on this *exploration*, the optimal organization is selected and used till the next phase is signaled. If there are three candidate hardware organizations, every new phase triggers an exploration process that lasts four intervals – one to detect the phase change and three to profile each of the candidates. Thus, each phase change can result in a maximum of three intervals being spent in sub-optimal organizations (since one of the explored organizations is the optimal one). Hence, an *instability factor* of 5% results in 15% of the intervals being spent in sub-optimal hardware organizations. If we assume that each sub-optimal organization is within 10% of the optimal organization on average, the interval-based scheme is likely to be within 1.5% of an optimal interval-based mechanism that uses an oracle to select the best hardware organization for each interval. Thus, the efficiency of this mechanism is a function of the number of candidate organizations, the *instability factor*, and the average distance between the optimal and sub-optimal organizations. Also note that the "optimal" behavior indicated by the oracle method is a function of the interval length – the shorter the interval length, the better its ability to target small variations in program behavior.

Exploration can be used in positional adaptation as well. Every time a new phase is encountered, a candidate organization is employed till the next phase. This behavior is recorded in a table indexed by the program counter value. After the same phase has been encountered a few times and all candidate organizations have been profiled, the best organization is used for that phase in the future.

2.2.2 Prediction

As discussed, the exploration process accurately selects the optimal organization, but incurs non-negligible overhead if there are many candidate organizations to choose from. In order to minimize this overhead, the hardware can estimate the optimal organization without profiling every single candidate. For example, if we are trying to pick a cache size that can keep the miss rate within a certain allowed value, exploration would profile the miss rate for each different cache organization. In order to predict the appropriate cache size, one could monitor usage statistics within a large cache to estimate the working set size and hence, the miss rate for a given cache size, e.g., using LRU statistics [Dropsho *et al.*, 2002]. While this technique can reduce the overhead of exploration, its success depends on the ability to efficiently determine and implement metrics that can accurately predict the optimal organization for the hardware being considered.

Both in positional adaptation and in interval-based mechanisms, either the first time a phase is encountered or when a phase change is detected, the behavior has to be profiled for one phase or interval in order to determine the optimal organization.

2.3 Case Studies

This chapter has discussed orthogonal issues in the design of dynamic selection mechanisms which will be applied to different hardware contexts in the rest of the dissertation. Phase changes can be detected at time-based interval boundaries or at specific program points. The optimal organization can be selected through exploration or by prediction based on hardware metrics. The combinations of these design choices result in a number of meaningful dynamic algorithms. We discuss some of the possible algorithms in this section.

It must be pointed out that all of these algorithms are based on certain heuristics and assumptions. For example, we are using history to predict behavior for the future. Hence, none of these algorithms are guaranteed to work on all programs – given an algorithm, it is always possible to construct a program where the algorithm would behave highly sub-optimally. However, the nature of our assumptions is such that they hold for most common programs and exceptions will be noted in the evaluations in subsequent chapters. The algorithms have features that minimize the damage when these exceptions are encountered.

2.3.1 Interval-Based Adaptation with Exploration

In our evaluations, we found that a mechanism with little overhead was the interval-based algorithm with exploration. Behavior across successive intervals is monitored to detect phase changes. At the start of each phase, an exploration process profiles the behavior for all of the different hardware organizations before selecting the best. The interval length is selected at run-time to keep the *instability factor* at a tolerable overhead.

The entire algorithm is listed in Figure 2.1. At the start of a phase, the statistics collected in the first interval serve as a reference point against which to compare future statistics and detect a phase change. The branch and memory reference frequencies are microarchitecture-independent parameters and can be used to detect phase changes even during the exploration process. After exploration, the best performing configuration is picked and its IPC is also used as a reference. A phase change is signaled if either the number of branches, the number of memory references, or the IPC differs significantly from the reference point. Occasionally, there is a slight change in IPC characteristics during an interval (perhaps caused by a burst

```
Initializations and definitions:
interval_length = 10K;              (number of committed instructions before invoking the algorithm)
discontinue_algorithm = FALSE;      (if this is set, no more reconfigurations are attempted until the next macrophase)
have_reference_point = FALSE;       (the first interval in a new phase provides a reference point against which to copare future intervals)
significant_change_in_ipc;          (this is set if the IPC in the current interval differs from that in the reference point by more than 10%)
significant_change_in_memrefs;      (this is set if the memory references in the current interval differs from the reference point by more than interval_length/100)
significant_change_in_branches;     (this is set if the number of branches in the current interval differs from the reference point by more than interval_length/100)
num_ipc_variations = 0;             (this indicates the number of times there was a significant_change_in_ipc)
stable_state = FALSE;               (this is set only after all configurations are explored and the best is picked)
instability = 0;                    (a number indicating the frequency of occurrence of a phase change)


Inspect statistics every 100 billion instructions.
If (new macrophase)
  Initialize all variables;

If (not discontinue_algorithm) execute the following after every interval_length instructions.
If (have_reference_point)
  If (significant_change_in_memrefs or significant_change_in_branches or ((significant_change_in_ipc and num_ipc_variations > THRESH1))
    have_reference_point = stable_state = FALSE;
    num_ipc_variations = 0;
    use first configuration for the next interval;
    instability = instability + 2;
    if (instability > THRESH2)
      interval_length = interval_length * 2;
      instability = 0;
      if (interval_length > THRESH3)
        Pick most popular configuration;
        discontinue_algorithm = TRUE;
  else
    if (significant_change_in_ipc)
      num_ipc_variations = num_ipc_variations + 2;
    else
      num_ipc_variations = MAX(−2,num_ipc_variations−0.125);
    instability = instability − 0.125;
else
  have_reference_point = TRUE;
  Record branches and memrefs.

If (have_reference_point and not stable_state)
  record IPC;
  use the next configuration in the next interval;
  if (all configurations have been profiled)
    pick the best performing configuration;
    make its IPC the IPC_reference_point;
    stable_state = TRUE;
```

Constants and Thresholds

The constant increments/decrements for num_ipc_variations and instability were chosen to roughly allow about 5% instability.

The other thresholds were picked to be reasonable round numbers.
THRESH1 = THRESH2 = 5.
THRESH3 = 1 billion instructions.

Figure 2.1: Run-time algorithm to select the best of many candidate hardware organizations. This is an interval-based mechanism with exploration. The constant increment/decrements for num_ipc_variations and instability were chosen to allow about 5% instability. The thresholds were picked to be reasonable round numbers. Note that THRESH1 has to be greater than the number of candidate configurations. We assume a significant_change_in_memrefs if the memory references differ from the reference point by more than 5% of the interval length. A significant_change_in_branches is signaled if it differs from the reference point by more than 2% of the interval length. A significant_change_in_IPC refers to a variation of more than 10% compared to the reference point.

of branch mispredicts or cache misses), after which, behavior returns to that of the previous phase. To discourage needless explorations in this scenario, we tolerate some noise in the IPC measurements (with the *num_ipc_variations* parameter). In addition, if phase changes are frequent, the *instability* variable is incremented and eventually, the interval length is doubled.

This entire process of run-time reconfiguration can be implemented in software with support from hardware event counters. A low-overhead software routine (like that used for software TLB miss handling) that inspects various hardware counters before making a decision on the subsequent configuration is invoked at every interval. The algorithm amounts to about 100 assembly instructions, only a small fraction of which are executed at each invocation. Even for the minimum interval length of 10K instructions, this amounts to an overhead of much less than 1%. Implementing the selection algorithm in software allows greater flexibility and opens up the possibility for application-specific algorithms. Algorithms at higher levels that detect changes in *macrophases* have an even lower overhead. Since the algorithm runs entirely in software, most program-specific state resides in memory as opposed to hardware registers. Hence, apart from the event counters, no additional state has to be saved and restored on a context switch.

The algorithm in Figure 2.1 assumes a dynamic interval length. As an alternative, below we detail an algorithm that uses adaptive thresholds to tolerate noise and the mismatch between the length of a program phase and the interval length. This algorithm was employed for cache reconfiguration in [Balasubramonian *et al.*, 2000b; Balasubramonian *et al.*, 2003a] and uses the number of branches and the number of cache misses as metrics to detect a phase change.

```
Initializations and definitions:
  base_br_noise = 4500; base_miss_rate_noise = 450;
  br_incr = 1000; br_decr = 50;
  miss_rate_incr = 100; miss_rate_decr = 5;
  miss_rate_noise = base_miss_rate_noise;
  br_noise = base_br_noise;
  state = UNSTABLE;

Repeat the following every 100K cycles:
(inputs: num_miss, num_br, CPI)
if (state == STABLE)
    if ((|num_miss-last_num_miss|) < miss_rate_noise &&
        (|num_br-last_num_br|) < br_noise)
        miss_rate_noise = max(miss_rate_noise-miss_rate_decr,
                              base_miss_rate_noise);
        br_noise = max(br_noise - br_decr, base_br_noise);
    else
        last_cache_size = cache_size;
        cache_size = SMALLEST; state = UNSTABLE;

else if (state == UNSTABLE)
    record CPI, num_miss, num_br;
    if ((num_miss > THRESHOLD) && (cache_size != MAX))
        Increase cache_size;
    else
        cache_size = that with best CPI; state = STABLE;
        last_num_miss = num_miss recorded for selected size;
        last_num_br = num_br recorded for selected size;
        if (cache_size == last_cache_size)
          miss_rate_noise= miss_rate_noise + miss_rate_incr;
          br_noise = br_noise + br_incr;
```

In order to reduce the overhead of the exploration process, behavior of past phases could be recorded and re-used when the same phase is again encountered. Dhodapkar and Smith [Dhodapkar and Smith, 2002] suggest using working set signatures to characterize a given phase. Instructions executed during an interval are combined to create a signature for that particular phase. The optimal organization selected for a phase is recorded in a table. Every phase change results in an interval during which the working set signature is computed to index into the table and retrieve the optimal organization. This ensures that each phase change results in a modest overhead – one interval to detect the phase change and one to compute the working set signature. However, if program behavior changes over time, an initial estimate of the optimal organization might be incorrect. Hence, at periodic intervals, the prediction tables have to be cleared and re-computed.

In all the interval-based algorithms, statistics collected during the interval that flags a phase change are discarded. The assumption is that it takes an interval to eliminate boundary effects (cache misses, branch mispredictions, surrounding code, etc.).

2.3.2 Interval-Based Adaptation with Prediction

This mechanism is identical to the one in the previous subsection, except that at the start of each phase, instead of exploring across multiple candidate organizations, one interval is spent at an organization that can help reveal the optimal hardware organization. This organization is then employed till the next phase change. The IPC reference point is the IPC for the first interval with the selected organization. The overhead per phase change potentially equals two intervals – one to detect the phase change and one to compute the optimal organization. Since the overhead is already at a minimum, it is not very meaningful to employ working set signatures to remember past behavior.

2.3.3 Positional Adaptation with Exploration

An algorithm based on positional adaptation requires a table to maintain state for every program counter value that signals a new phase. As an example, consider an algorithm that defines every N^{th} branch as a phase change (Figure 2.2). The table is cleared at the start of the execution and at periodic intervals (say, every 10 million instructions). At every N^{th} branch, the program counter is used to look up the table. If the table has no entry for this branch, the hardware has to detect which organization is optimal with an exploration process. Every time this branch is encountered, a new hardware organization is implemented and profiled (this organization is maintained till the program moves to the next phase.). This information is stored in the table just before the start of the next phase. After all the organizations have been profiled, the table is inspected to determine the best. Accordingly, the next time this phase is encountered, the table recommends that this optimal organization be used. While every N^{th} branch signals a phase change, successive phases might continue to use the same organization – this is unlike the interval-based mechanism, where every new phase requires at least one interval of profiling a particular hardware organization.

A few important details have to be taken into consideration. An initial exploration or monitoring process updates a table that is used for the remainder of the execution. This results in initial overhead that gets amortized over the whole program. However, if program behavior changes over time, an initial estimate of the optimal organization might be incorrect. Hence, at

```
Initializations and definitions.  (All the tables are indexed by using some bits of the program counter)
valid_entry[]=FALSE;  The bits that indicate if a given branch has a valid entry in the tables
stable_state[];    The bits that indicate if a given branch has completed the exploration process
prediction[];      The table that indicates the predicted optimal organization for each phase
statistics[];      The table that maintains exploration statistics for each phase

Every 10 million instructions,
    Reset all the valid_entry[] bits to FALSE;


At every Nth branch,
    For the phase just exited,
    If (no valid_entry[branch_pc])
        valid_entry[branch_pc] = TRUE; Record statistics for the first organization;
    else if (not stable_state[branch_pc])
        Record statistics for the organization just profiled;
        if (all organizations profiled)
            stable_state[branch_pc] = TRUE;
            prediction[branch_pc] = optimal organization;
    branch_pc = the current branch encountered;

    If (no valid_entry[branch_pc])
        Start exploration process;
        Select the first candidate hardware organization;
    else if (not stable_state[branch_pc])
        Select the next organization to be explored;
    else
        Select prediction[branch_pc] as the next organization;
```

The length of the exploration process depends on the number of candidate hardware organizations and the number of samples required for each. The table has to be large enough to record these data points and this has to be taken into account while determining the next organization to be explored.

Figure 2.2: Run-time algorithm to select the best of many candidate hardware organizations. This uses the positional adaptation approach at the branch level and employs an exploration process.

periodic intervals, the prediction tables have to be cleared and re-computed. If phases are signaled frequently, the behavior of neighboring phases might influence the statistics collected for a phase. To make the mechanism robust to these boundary effects, statistics could be collected for a number of samples before any decision-making. This could also handle cases where the behavior within a phase is not always consistent.

Since this algorithm is invoked at every N^{th} branch, it has to be implemented in hardware, so as to not slow down the processor with software interrupts.

The prediction table has finite capacity. Assuming that some of the program counter bits are used to index into this table, it is possible that multiple phases might conflict for the same entry in the table. Hence, the table has to be made large enough to minimize these conflicts. These tables may maintain tags so that a conflict for an entry results in an eviction of an earlier entry and the creation of a new one. If tags are not maintained, the statistics collected for two different phases may interfere with each other.

2.3.4 Positional Adaptation with Prediction

Again, this mechanism is very similar to the one described in the earlier subsection. When there is no entry in the prediction table for a particular phase, instead of going through an exploration process, a particular organization is selected that helps the hardware profile the program behavior. Based on this profiling, the behavior of different hardware organizations is predicted and the estimated optimal organization is selected and recorded in the prediction table. To minimize boundary effects, a number of samples may have to be taken before the program behavior is accurately profiled. Otherwise, the algorithm is as described before.

2.4 Related Work

Many recent bodies of work [Albonesi, 1998; Bahar and Manne, 2001; Buyuktosunoglu *et al.*, 2000; Dhodapkar and Smith, 2002; Folegnani and Gonzalez, 2000; Ghiasi *et al.*, 2000; Huang *et al.*, 2000; Ponomarev *et al.*, 2001; Yang *et al.*, 2001] have looked at hardware units with multiple configuration options and algorithms for picking an appropriate configuration at run-time. Many of these algorithms are interval-based, in that, they monitor various statistics over a fixed interval of instructions or cycles and make configuration decisions based on that information. These algorithms usually employ reactive metrics (throughput, utilization, miss rates) to select an organization. Huang et al. [Huang *et al.*, 2003] study adaptation at subroutine boundaries and demonstrate that this can be more effective than using fixed instruction intervals. Magklis et al. [Magklis *et al.*, 2003] also employ positional adaptation and use profiled information to identify subroutines and loops that are likely to benefit the most. Sherwood et al. [Sherwood *et al.*, 2003] classify the program into multiple phases based on the basic blocks executed as well as their frequency of execution in different parts of the program. They also design a run length encoding Markov predictor to predict the next phase in the program. Our results lead us to believe that simpler metrics such as branch frequency, miss rates, memory reference frequency, and IPC are sufficient to detect phase changes. Our proposals [Balasubramonian *et al.*, 2000a; Balasubramonian *et al.*, 2000b] have been one of the early contributions in this field. We have also been one of the first to identify the importance of a variable-length instruction interval length [Balasubramonian *et al.*, 2003b], as well as adaptive thresholds [Balasubramonian *et al.*, 2000b; Balasubramonian *et al.*, 2003a]. Since we do not rely on profiled information, the algorithms behave well regardless of the input data set used for the program.

2.5 Summary

In this chapter, we have examined a number of dynamic adaptation algorithms that attempt to predict program behavior at run time. Unlike static analysis or profiling, these algorithms perform well in spite of dynamic changes in input sets and data structures. In the subsequent chapters, we introduce techniques to adapt the hardware that allows us to optimize various trade-offs at run-time. In each scenario, one of many candidate hardware organizations yields optimal performance or power. The algorithms described in this chapter help us detect new behavior and then select the optimal organization for this new phase. All the adaptation algorithms are based on the premise that past behavior is likely to repeat in the future. A new program behavior can be detected by either examining statistics across successive periods of execution (Interval-based) or by the entry into a different basic block or subroutine (Positional adaptation). Once such a *phase change* has been detected, the optimal hardware configuration can be selected either by profiling the behavior of each candidate organization (Exploration) or by gathering program metrics that can instantly predict the optimal configuration (Prediction). The interval-based mechanisms have low implementation complexity, but capture program behavior at a fairly coarse granularity. It can also entail heavy overheads if the number of candidate organizations is very large. This is likely to be the case if there are a number of configurable hardware units that interact with each other. Positional adaptation reacts faster to a phase change, but can occasionally be inaccurate and entails hardware overhead. Exploration is reliable, but can consume many cycles if the search space is very large. The effectiveness of the prediction technique relies on the existence of a program metric that can accurately predict the behavior of the program on the given hardware.

3 Cache Reconfiguration

In Chapter 2, we introduced a number of techniques to detect program behavior changes at run-time so that optimal hardware organizations could be employed at any given point in the program's execution. Modern microprocessors have fixed designs and do not allow the hardware to adapt dynamically. The widening gap between different trade-off points in future technologies necessitates the use of adaptive structures on the chip. The primary focus of the thesis is to study how the different components of a microprocessor lend themselves to such adaptation and how the control mechanisms described in Chapter 2 behave in these different scenarios. In this chapter, we examine trade-offs in the design of the on-chip data cache and re-organize the layout to allow for multiple configurations with differing trade-off characteristics.

3.1 Cache Design Issues

Microprocessor clock speeds have been improving at a much faster rate than memory speeds. As a result, large caches are employed in modern processors and their design has a significant impact on processor performance and power consumption. The most common conventional memory system today is the multi-level memory hierarchy. The rationale behind this approach, which is used primarily in caches but also in some TLBs (*e.g.*, in the MIPS R10000 [Yeager, 1996]), is that a combination of a small, low-latency L1 memory backed by a higher capacity, yet slower, L2 memory and finally by main memory provides the best trade-off between optimizing hit time and miss time. The fundamental issue with these designs is that they are targeted to the average application — no single memory hierarchy organization proves to be the best for all applications. When running a diverse range of applications, there will inevitably be significant periods of execution during which performance degrades and energy is needlessly expended due to a mismatch between the memory system requirements of the application and the memory hierarchy implementation. As an example, programs whose working sets exceed the L1 capacity may expend considerable time and energy transferring data between the various levels of the hierarchy. If the miss tolerance of the application is lower than the effective L1 miss penalty, then performance may degrade significantly due to instructions waiting for operands to arrive. For such applications, a large, single-level cache (as used in the HP PA-8X00 series of microprocessors [Gwennap, 1997; Kumar, 1997; Lesartre and Hunt, 1997]) may perform better and be more energy-efficient than a two-level hierarchy for the same total amount of memory.

To meet the needs of a diverse set of applications, we evaluate a cache layout that is flexible. Key to our approach is the exploitation of the properties of conventional caches and future technology trends in order to provide cache configurability in a low-intrusive and low-latency manner. Our cache is logically designed and laid out as a *virtual two-level, physical one-level* non-inclusive hierarchy, where the partition between the two levels is dynamic. The non-inclusive nature of the hierarchy minimizes the overheads at the time of repartitioning. To maximize performance, an organization that balances hit and miss latency intolerance is chosen.

3.2 The Reconfigurable Cache Layout

3.2.1 Circuit Structures

The cache layouts (both conventional and configurable) that we model follow that described by McFarland in his thesis [McFarland, 1997]. McFarland developed a detailed timing model for both the cache and TLB that balances both performance and energy considerations in subarray partitioning, and which includes the effects of technology scaling.

We took into account several considerations in choosing the cache layout as well as parameters such as size and associativity for our configurable cache and the L2 cache in a conventional processor. First, we determined that the cache should be at least 1MB in size. We based this on the size of on-chip L2 caches slated to be implemented in modern processors (such as the Alpha 21364 [Bannon, 1998] which has 1.5MB of on-chip cache). Based on performance simulation results with our benchmark suite, we picked 2MB as the target size for our configurable cache as well as for the L2 (or combined L1 and L2 in the case of an exclusive cache) of the conventional baseline memory hierarchies.

To further define the number of subarrays and associativity, we calculated (following Bakoglu [Bakoglu and Meindl, 1985]) the SRAM array wordline delay as a function of the number of array columns and the number of wire sections (separated by repeater switches) using the 0.1 micrometer parameters of McFarland [McFarland and Flynn, 1995]. The best delay was achieved with four repeater switches for 2048 columns, and eight for 4096 columns.

Based on the above constraints, on delay calculations using various numbers of subarrays and layouts, and on the need to make the cache banked to obtain sufficient bandwidth, we arrived at an organization in which the cache is structured as two 1MB interleaved banks[1]. In order to reduce access time and energy consumption, each 1MB bank is further divided into two 512KB SRAM structures (with data being block interleaved among the structures), one of which is selected on each bank access.

The data array section of the configurable structure is shown in Figure 3.1 in which only the details of one subarray are shown for simplicity. (The other subarrays are identically organized). There are four subarrays, each of which contains four ways. Each of these subarrays has 512 rows and 2048 columns. In both the conventional and configurable cache, two address bits (*Subarray Select*) are used to select only one of the four subarrays on each access in order to reduce energy dissipation. The other three subarrays have their local wordlines disabled and their precharge, sense amp, and output driver circuits are not activated. The TLB virtual to real page number translation and tag check proceed in parallel and only the output drivers for the way in which the hit occurred are turned on. Parallel TLB and tag access can be accomplished if the operating system can ensure that *index_bits-page_offset_bits* bits of the virtual and physical addresses are identical, as is the case for the four-way set associative 1MB dual-banked L1 data cache in the HP PA-8500 [Fleischman, 1999].

[1]The banks are word-interleaved when used as an L1/L2 cache hierarchy and block interleaved when used as L2/L3.

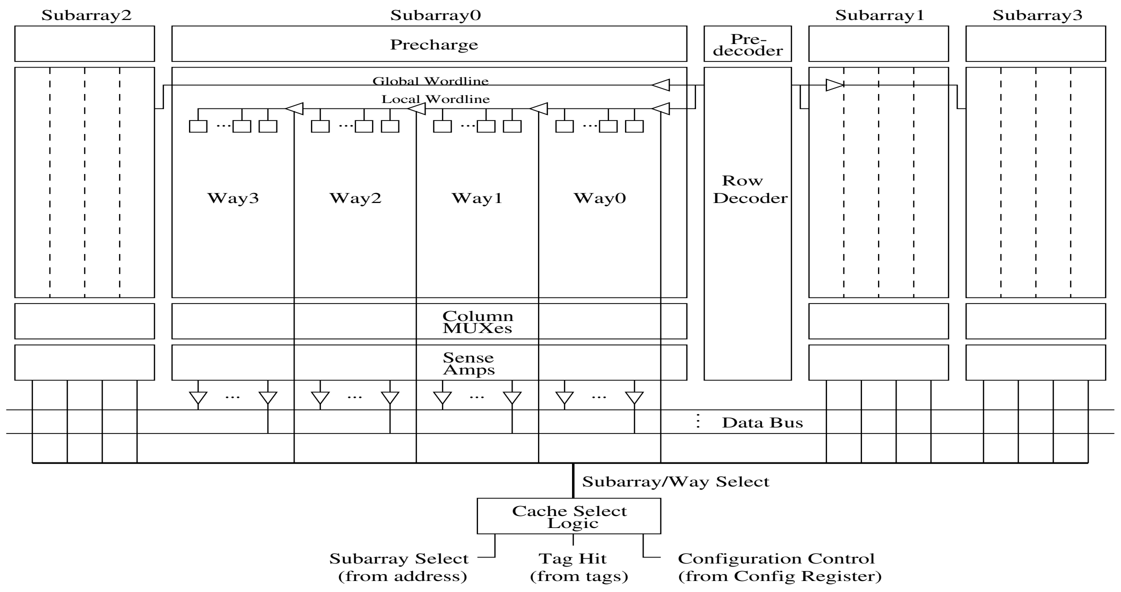


Figure 3.1: The organization of the data array section of one of the 512KB cache structures.

The key modifications to McFarland's baseline design that allow for multiple configuration options are the conscious placement of repeaters in the local and global wordlines. Repeater switches are used in the global wordlines to electrically isolate each subarray. That is, subarrays 0 and 1 do not suffer additional global wordline delay due to the presence of subarrays 2 and 3. Providing switches as opposed to simple repeaters also prevents wordline switching in disabled subarrays thereby saving dynamic power. Repeater switches are also used in the local wordlines to electrically isolate each way in a subarray. The result is that the presence of additional ways does not impact the delay of the fastest ways. Dynamic power dissipation is also reduced by disabling the wordline drivers of disabled ways. *Configuration Control* signals from the *Configuration Register* provide the ability to disable entire subarrays or ways within an enabled subarray. Local wordline and data output drivers and precharge and sense amp circuits are not activated for a disabled subarray or way. More details on the circuit structures can be found in [Balasubramonian *et al.*, 2000b].

3.2.2 Configurable Cache Operation

With these modifications, the sizes, associativities, and latencies of the resulting virtual two-level, physical one-level non-inclusive cache hierarchy are *dynamic*. That is, a single large cache organization serves as a configurable two-level non-inclusive cache hierarchy, where the ways within each subarray that are initially enabled for an L1 access are varied to match application characteristics. The latency of the two sections is changed on half-cycle increments according to the timing of each configuration. Half cycle increments are required to provide the granularity to distinguish the different configurations in terms of their organization and speed. Such an approach can be implemented by capturing cache data using both phases of the clock, similar to the double-pumped Alpha 21264 data cache [Kessler *et al.*, 1998], and enabling the appropriate latch according to the configuration. The advantages of this approach is that the timing of the cache can change with its configuration while the main processor clock remains unaffected, and that no clock synchronization is necessary between the pipeline and cache/TLB.

However, because a constant two-stage cache pipeline is maintained regardless of the cache configuration, cache bandwidth degrades for the larger, slower configurations. Furthermore, the implementation of a cache whose latency can vary on half-cycle increments requires two pipeline modifications. First, the dynamic scheduling hardware must be able to speculatively issue (assuming a data cache hit) load-dependent instructions at different times depending on the currently enabled cache configuration. Second, for some configurations, running the cache on half-cycle increments requires an extra half-cycle for accesses to be caught by the processor clock phase.

The possible configurations for a 2MB L1/L2 on-chip cache hierarchy at 0.1μm technology are shown in Figure 3.2. Although multiple subarrays may be enabled as L1 in an organization, as in a conventional cache, only one is selected each access according to the *Subarray Select* field of the address. When a miss in the L1 section is detected, all tag subarrays and ways are read. This permits hit detection to data in the remaining portion of the cache (designated as L2 in Figure 3.2). When such a hit occurs, the data in the L1 section (which has already been read out and placed into a buffer) is swapped with the data in the L2 section. In the case of a miss to both sections, the displaced block from the L1 section is placed into the L2 section. This

Subarray/Way Allocation (L1 or L2)

Cache Configuration	L1 Size	L1 Assoc	L1 Acc Time	Subarray 2				Subarray 0				Subarray 1				Subarray 3			
				W3	W2	W1	W0	W3	W2	W1	W0	W0	W1	W2	W3	W0	W1	W2	W3
256-1	256KB	1 way	2.0	L2	L2	L2	L2	L2	L2	L2	***L1***	***L1***	L2	L2	L2	L2	L2	L2	L2
512-2	512KB	2 way	2.5	L2	L2	L2	L2	L2	L2	***L1***	***L1***	***L1***	***L1***	L2	L2	L2	L2	L2	L2
768-3	768KB	3 way	2.5	L2	L2	L2	L2	L2	***L1***	***L1***	***L1***	***L1***	***L1***	***L1***	L2	L2	L2	L2	L2
1024-4	1024KB	4 way	3.0	L2	L2	L2	L2	***L1***	***L1***	***L1***	***L1***	***L1***	***L1***	***L1***	***L1***	L2	L2	L2	L2
512-1	512KB	1 way	3.0	L2	L2	L2	***L1***	L2	L2	L2	***L1***	***L1***	L2	L2	L2	***L1***	L2	L2	L2
1024-2	1024KB	2 way	3.5	L2	L2	***L1***	***L1***	L2	L2	***L1***	***L1***	***L1***	***L1***	L2	L2	***L1***	***L1***	L2	L2
1536-3	1536KB	3 way	4.0	L2	***L1***	***L1***	***L1***	L2	***L1***	***L1***	***L1***	***L1***	***L1***	***L1***	L2	***L1***	***L1***	***L1***	L2
2048-4	2048KB	4 way	4.5	***L1***	***L1***	***L1***	***L1***	***L1***	***L1***	***L1***	***L1***	***L1***	***L1***	***L1***	***L1***	***L1***	***L1***	***L1***	***L1***

Figure 3.2: Possible L1/L2 cache organizations that can be configured shown by the ways that are allocated to L1 and L2. Only one of the four 512KB SRAM structures is shown. Abbreviations for each organization are listed to the left of the size and associativity of the L1 section, while L1 access times in cycles are given on the right. Note that the TLB access may dominate the overall delay of some configurations. The numbers listed here simply indicate the relative order of the access times for all configurations and thus the size/access time trade-offs allowable.

prevents thrashing in the case of low-associative L1 organizations.

The direct-mapped 512KB and two-way set associative 1MB cache organizations are lower energy, and lower performance, alternatives to the 512KB two-way and 1MB four-way organizations, respectively. These options activate half the number of ways on each access for the same capacity as their counterparts. For execution periods in which there are few cache conflicts and hit latency tolerance is high, the low energy alternatives may result in comparable performance yet potentially save considerable energy. These configurations can be used in an energy-aware mode of operation.

Note that because some of the configurations span only two subarrays, while others span four, the number of sets is not always the same. Hence, it is possible that a given address might map into a certain cache line at one time and into another at another time (called a *mis-map*). However, the non-inclusive nature of our cache helps prevent aliasing and guarantees correctness. The high-order *Subarray Select* signal is replicated as an extra tag bit. This extra tag bit is used to detect mis-maps. In order to minimize the performance impact of mis-mapped data, an L2 look-up examines twice as many tags as the conventional L2 in order to examine all subarrays where mis-mapped data may reside. Mis-mapped data is handled the same way as a L1 miss and L2 hit, *i.e.*, it results in a swap. Our simulations indicate that such events are infrequent.

In conventional two-level hierarchies, the L2 implements additional hardware to ensure cache coherence with the L2 caches of other chips. This involves replication of its tags and a snooping protocol on the system bus. The use of a non-inclusive L1-L2 does not affect this in any way. The tags for the entire 2MB of on-chip cache are replicated (just as for the 2MB L2 in the conventional hierarchy) and the snooping protocol ensures that data in the L1 and L2 are coherent with other caches. There is potential for added interference with the processor to L1 datapath when compared to a conventional design in the presence of inactive shared data that would only be present in the L2 in the case of a conventional design.

3.2.3 Configurable L2-L3 Cache

In sub-0.1μm technologies, the long access latencies of a large on-chip L2 cache [Agarwal *et al.*, 2000] may be too expensive for those applications which make use of only a small fraction of the L2 cache. Thus, for performance reasons, a three-level hierarchy with a moderate size (*e.g.*, 512KB) L2 cache will become an attractive alternative to two-level hierarchies at these feature sizes. However, the cost may be a significant increase in energy dissipation due to transfers involving the additional cache level. The use of the aforementioned configurable cache structure as a replacement for conventional L2 and L3 caches can significantly reduce energy dissipation without any compromise in performance as feature sizes scale below 0.1μm.

3.2.4 The Hot-and-Cold Cache

The primary focus in this chapter has been on dynamic cache reorganization to improve performance and power consumption. We have also explored static organizations that can reduce power consumption in the cache.

A significant saving in leakage and dynamic power is possible if circuit techniques are leveraged, for example, high threshold voltage devices for low leakage, transistor sizing to reduce capacitance, etc. However, by doing this, it is harder to switch between different modes of operation and the cache organization has to be statically defined. Unfortunately, most of these energy-saving techniques incur a cost in access times, which can significantly impact performance.

We proposed a banked cache organization, where half the cache banks (the *hot* banks) are designed to optimize performance, while the remaining banks (the *cold* banks) are designed to minimize power consumption. The *cold* banks could have an access time that is longer than the *hot* banks, but they potentially consume a small fraction of the power consumed by the *hot* banks. A number of instructions, which are not on the program critical path, can be slowed down with a minimal impact on performance. If these *non-critical* instructions and the data they access are restricted to the *cold* cache banks, a large saving in energy consumption could be had, while minimally impacting performance. We studied if instructions and data readily lend themselves to such a classification and found that this is often the case. On average, 90%

Branch predictor	comb. of bimodal & 2-level gshare; Combining pred. entries - 1024; bimodal/Gshare Level1/2 entries - 2048, 1024 (hist. 10), 4096 (global); RAS entries - 32; BTB - 2048 sets, 2-way
Branch mispred. latency	8 cycles
Fetch, decode, issue width	4
RUU and LSQ entries	64 and 32
L1 I-cache	2-way; 64KB (0.1μm), 32KB (0.035μm)
Memory latency	80 cycles (0.1μm), 114 cycles (0.035μm)
Integer ALUs/mult-div	4/2
FP ALUs/mult-div	2/1

Table 3.1: Architectural parameters.

of all instructions were steered to the bank that cached their data and 80% of all loads and stores were serviced by a cache bank that matched their criticality characteristics.

3.3 Methodology

For our simulations, we used Simplescalar-3.0 [Burger and Austin, 1997] for the Alpha AXP ISA. We modeled an aggressive 4-way superscalar out-of-order processor. The architectural parameters used in the simulation are summarized in Table 3.1. The data memory hierarchy is modeled in great detail. For example, contention for all caches and buses in the memory hierarchy as well as for writeback buffers is modeled. A line size of 128 bytes was chosen because it yielded a much lower miss rate for our benchmark set than smaller line sizes. We use cycles per instruction (CPI) to represent program performance. This metric has the nice property that the execution cycles can be intuitively broken down as those spent accessing memory and those spent in the CPU.

As a benchmark set, we select a variety of programs drawn from different suites like SPEC95, SPEC2000, and the Olden suite [Rogers *et al.*, 1995]. Since we are evaluating a cache layout that provides configurations of sizes between 256KB and 2MB, our benchmark set primarily consists of programs that have working set sizes in this range or higher. Including programs with smaller working sets would result in the smallest organization being used con-

Benchmark	Suite	Datasets	Simulation window (instructions)	256KB-1way L1 miss rate
em3d	Olden	20,000 nodes, arity 20	1000M-1100M	16%
health	Olden	4 levels, 1000 iters	80M-140M	11%
mst	Olden	256 nodes	entire program 14M	5%
compress	SPEC95 INT	ref	1900M-2100M	6%
hydro2d	SPEC95 FP	ref	2000M-4000M	4%
apsi	SPEC95 FP	ref	2200M-4200M	7%
swim	SPEC2000 FP	ref	2500M-3500M	9%
art	SPEC2000 FP	ref	300M-2300M	14%

Table 3.2: Benchmarks.

stantly, thus proving to be uninteresting. Note that the cache layout is one potential example at a single technology point. Other layouts with different cache options might target a different class of benchmark programs. The programs were compiled with the Compaq cc, f77, and f90 compilers at an optimization level of O3. Warm-up times were determined for each benchmark, and the simulation was fast-forwarded through these phases. A further million instructions were simulated in detail to prime all structures before starting the performance measurements. In almost all cases, simulation windows have been picked to represent overall program behavior. Table 3.2 summarizes the benchmarks and their miss rates for the smallest cache.

For the 0.1μm design point, we use the cache and TLB timing model developed by McFarland [McFarland, 1997] to estimate timings for both the configurable and conventional caches and TLBs. McFarland's model contains several optimizations, including the automatic sizing of gates according to loading characteristics, and the careful consideration of the effects of technology scaling down to 0.1μm technology [McFarland and Flynn, 1995]. The model integrates a fully-associative TLB with the cache to account for cases in which the TLB dominates the L1 cache access path. For the global wordline, local wordline, and output driver select wires, we recalculate cache and TLB wire delays using RC delay equations for repeater insertion [Dally and Poulton, 1998]. Repeaters are used in the configurable cache as well as in the conventional L1 cache whenever they reduce wire propagation delay. The energy dissipation of these repeaters was accounted for, and they add only 2-3% to the total cache energy. For our experiments at the

0.035μm design point, we use the cache latency values of Agarwal et al. [Agarwal *et al.*, 2000] whose model parameters are based on projections from the Semiconductor Industry Association Technology Roadmap [Association, 1999].

3.4 Results

3.4.1 Dynamic Selection Mechanisms

Our configurable cache permits picking appropriate configurations and sizes based on application requirements. The different configurations spend different amounts of time and energy accessing the L1 and the lower levels of the memory hierarchy. Our primary focus is to study the impact of these selection mechanisms on performance by balancing hit latency and miss rate for each application phase. We also demonstrate how the mechanisms can be modified to opportunistically trade off a small amount of performance for significant energy savings.

Chapter 2 presents a number of selection mechanisms that differ in how they select the optimal configurations and the configuration points. We evaluate how some of these selection mechanisms perform in the context of the reconfigurable cache.

The interval-based mechanism with exploration entails the least amount of overhead. Chapter 2 illustrates that it is applicable to a wide class of applications. Every phase change requires an exploration process, during which different configurations are profiled. A property of this specific problem domain that has to be considered while designing the algorithm is the relatively long time it takes to warm up a cache and profile it accurately. In our study, it takes an interval of at least 100K instructions to allow the cache enough time to react to the reconfiguration and demonstrate its representative behavior. The following five cache configurations are the most meaningful from the point of view of performance: 256KB 1-way L1, 768KB 3-way L1, 1MB 4-way L1, 1.5MB 3-way L1, and 2MB 4-way L1. The 512KB 2-way L1 configuration provides no performance advantage over the 768KB 3-way L1 configuration (due to their identical access times in cycles) and thus this configuration is not used. For similar reasons, the two low-energy configurations (direct-mapped 512KB L1 and two-way set associative 1MB L1) are not used while maximizing performance. Hence, every phase change results in six intervals of overhead – one to detect the phase change and five to explore all the candidate organizations. The algorithm is exactly like that described in Chapter 2, with the thresholds attempting to pick an interval length that allows an *instability factor* of 5%.

The mechanism using positional adaptation and exploration is also just as described in Chapter 2. The primary advantage of positional adaptation is its ability to react quickly to short program phases. Unfortunately, the adaptation of the hardware itself is not very quick in this case. While it takes only a few cycles to change the hardware organization, it can take many hundreds of cycles for the L1 cache to contain relevant data and improve performance. Hence, it is meaningful to attempt reconfiguration at a relatively coarse granularity (at every 100 branches or only for large subroutines). The results in the next subsections deal with these issues in greater detail.

As an alternative to the exploration process, we also evaluate the use of a metric to predict the optimal organization. This metric, which is based on those proposed by Dropsho et al. [Dropsho *et al.*, 2002], attempts to estimate the working set size of the executing program. While employing this in tandem with the interval-based mechanism, at the start of each phase, we select the largest 2MB 4-way set-associative L1 cache for an interval. During this interval, we keep track of the number of accesses to the different ways in the cache. Combined with the LRU state maintained in the tags, we have the information to predict the miss rate for any given cache size. For example, if the most recently used way is accessed 90% of the time, the miss rate for a 1-way cache is about 10%. To simplify the hardware overhead, we maintain a single counter to estimate the required cache size (the LRU counter). If the least recently used way is accessed, the counter is incremented by three, if the next least recently used way is accessed, the counter is incremented by two, and so on. At the end of the interval, we compare the counter value against empirically pre-determined thresholds to predict the working set size and hence, the cache organization to be used. A low counter value indicates that the most recently used way is accessed most of the time and a small cache is good enough to maintain a low miss rate. In our experiments, while using a 100K instruction interval, cache sizes of 256KB, 768KB, 1MB, and 1.5MB were used if the LRU counter values were less than 0.5K, 6K, 9K, and 12K, respectively. The same predictive metric can also be employed with positional adaptation.

3.4.2 Interval-Based Adaptation

The performance afforded by a given cache organization is determined greatly by the L1 miss rate and to a lesser extent by the L1 access time. A number of programs have working sets that do not fit in today's L1 caches. For our chosen memory-intensive benchmark set, half of the total execution time can be attributed to memory hierarchy accesses. Increasing the size of the L1 and thereby reducing the miss rate has a big impact on cycles per instruction (CPI) for such programs. At the same time, the increased access time for the L1 results in poorer performance for other non-memory-intensive programs. For example, we observed that for most SPEC95 integer programs, each additional cycle in the L1 access time resulted in a 4-5% performance loss.

The reconfigurable L1/L2 cache provides a number of attractive design points for both memory-intensive and non-memory-intensive applications. Programs that do not have large working sets and do not suffer from many conflict misses can use the smaller and faster 256KB direct-mapped L1. Programs with large working set sizes, whose execution times are dominated by accesses to the L2 and beyond can use large L1 sizes so that most accesses are satisfied by a single cache look-up. While each access now takes longer, its performance effect is usually smaller than the cost of a higher miss rate. Moving to a larger cache size not only handles many of the capacity misses, it also takes care of a number of conflict misses as the associativity is increased in tandem. In our experiments, the combined L1-L2 hierarchy has a 2MB capacity. If the working set of the program is close to 2MB, the entire cache can be used as the L1. This not only reduces the miss rate, it also eliminates the L2 look-up altogether, reducing the effective memory access time. Our benchmark set represents programs with various working set sizes and associativity needs (even for different phases of the same program) and the dynamic selection mechanisms adapt the underlying L1-L2 cache hierarchy to these needs.

In Figure 3.3, we demonstrate how the different adaptation algorithms influence the performance of the reconfigurable cache. As a base case, we use a fixed L1-L2 non-inclusive cache hierarchy, where the L1 is a 256KB direct-mapped 2-cycle cache and is backed by a 14-way 1.75MB L2. Each access to the L2 takes 15 cycles and is pipelined to allow a new request every four cycles. This organization is exactly like a reconfigurable cache where the smallest cache

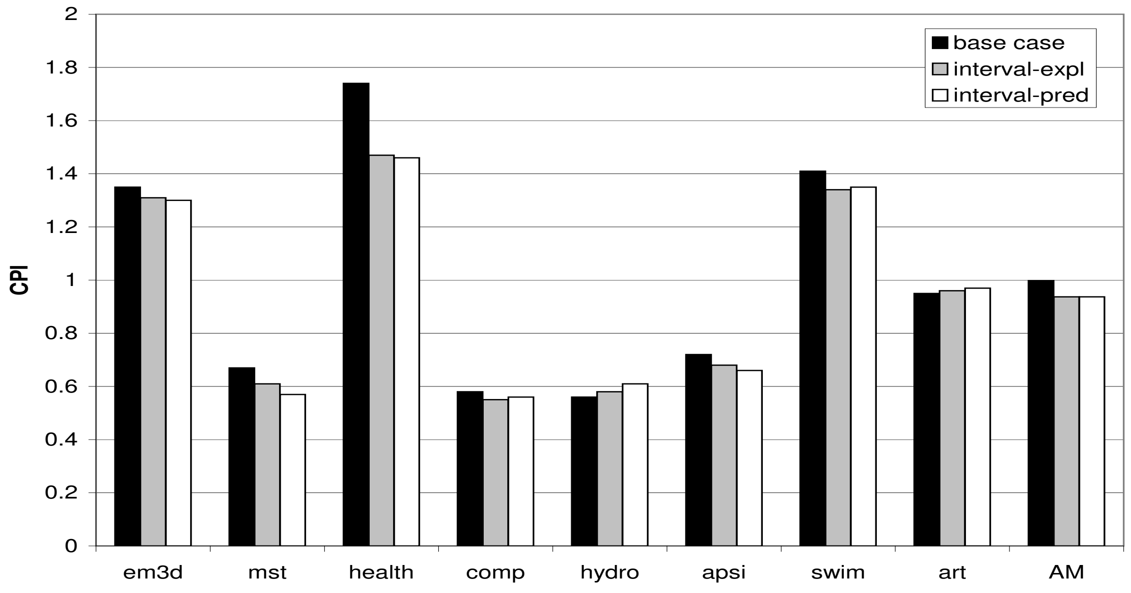


Figure 3.3: Cycles Per Instruction (CPI) results for the base case and the interval-based adaptation schemes. The second bar employs the exploration process and the third employs prediction.

size is always used. The second bar in Figure 3.3 represents an interval-based mechanism with exploration while the third bar represents the interval-based mechanism while employing the LRU counter to predict the optimal organization.

We see that in terms of overall performance (arithmetic mean of CPIs), the interval-based mechanisms work best. We found that the branch and memory reference frequency and the CPI were excellent metrics to detect phase changes. Half the programs did not display much variation and worked well with a 100K instruction interval length. *Em3d* and *Mst* spent most of their execution time with a 768KB cache, while *health* employed a 1.5MB cache for most of the time and *swim* used all 2MB as its L1 cache. *Swim* and *health* showed different behaviors over different program phases, but ended up selecting the same optimal cache size. *Compress* was another program that worked well with a 100K instruction interval length and had two distinct phases, one that did well with the smallest 256KB cache and the other with a 768KB cache. *Apsi* is a program that needs a set-associative cache to contain its working set size. Hence, it spends most of its time with a 1MB 4-way cache. However, it moves through a number of subroutines,

Benchmark	Number of phase changes	Most commonly selected configurations
em3d	0	1MB
health	9	1.5MB
mst	5	768KB, 1MB, 1.5MB
compress	41	256KB, 768KB
hydro2d	29	256KB, 768KB, 2MB
apsi	29	256KB, 768KB
swim	5	2MB
art	24	256KB, 1MB, 1.5MB, 2MB

Table 3.3: Number of phase changes encountered for each program during cache reconfiguration.

each lasting a few million instructions, and experiences frequent phase changes at small interval lengths. The interval length is increased to 12.8M before the phase changes are reduced to a tolerable extent. A similar phenomenon is also observed for *art* – the interval length has to be increased to 25.6M before the behavior is consistent across intervals, at which point the 768KB cache is selected as the optimal organization. Since the results include the cycles spent initially to select the appropriate interval length, the overall performance is slightly poorer than the base case.

Table 3.3 details the number of phase changes observed for each benchmark. This includes the phase changes encountered at the start of the simulation, while the appropriate interval length is still being selected.

The only program that we were unable to target was *hydro2d*. Even at the end of the two billion instruction simulation, the appropriate interval length could not be computed. As a result, performance degraded by about 4%. Note that if the dynamic scheme is unable to detect a reasonable interval length quickly, it can turn itself off and use the smallest cache organization so that it does no worse than the base case. Thus the negative performance impact can be controlled. Overall, we observed a 6.5% improvement in performance by using the interval-based technique with exploration, with the performance improvement ranging from -4% to

16%.

In prior work [Balasubramonian *et al.*, 2000a; Balasubramonian *et al.*, 2003a], we also examined the use of hit and miss intolerance metrics to limit the exploration process. When the stall cycles associated with a cache miss are fairly low, a larger cache is unlikely to yield improved performance. This was exploited to limit the search space of the exploration process and it yielded marginal performance improvements.

The third bar in Figure 3.3 represents the interval-based mechanism where every phase change triggers an interval at the largest L1 cache size that helps compute the LRU counter value to predict the optimal cache size. In most cases, this technique performs marginally better than with the exploration process. In *mst* the improvement is as high as 7% because of its ability to react quickly to a short phase. In the other programs, since phase changes might not occur very frequently, the improvement is marginal, if at all. *Hydro2d* was the only program where this technique did noticeably worse (5%), but this can be attributed to the fact that its behavior is not consistent across successive intervals and any mechanism that uses history to select a configuration for the future is likely to exhibit unpredictable performance. We observed that the predictions made with the LRU counter were quite accurate – often, the predicted optimal organization was either identical to or within a few per cent of the optimal organization selected by the exploration process. The overall improvement across the benchmark set was 6.5%.

Summary. Our conclusion from this portion of the study is that an interval-based technique is likely to provide most of the available benefits from a reconfigurable cache. Except for one example, we were able to keep the instability factor to a minimum and thus, reduce the overhead of detecting the optimal organization. Prediction with the LRU counter is effective, though, empirically, not much better than using exploration.

3.4.3 TLB Reconfiguration

The discussion so far has assumed that only the cache size is adapted at run-time. This helps isolate the effect of the adaptation algorithms on a single processor structure. However, the TLB is another storage structure that is accessed in parallel with the cache. A large working set size can not only result in a large cache miss rate, but also a large TLB miss rate. The access

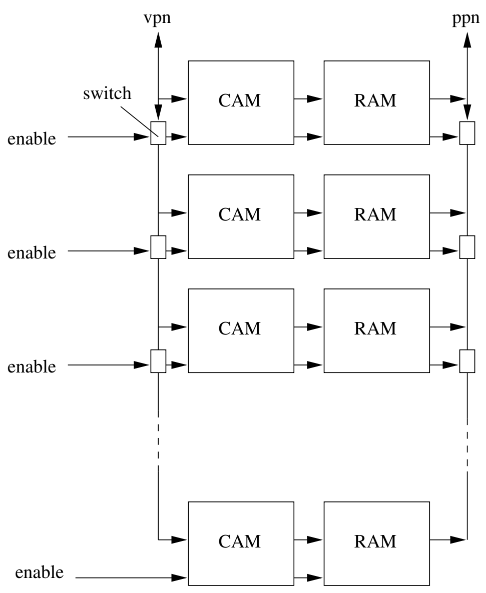


Figure 3.4: The Reconfigurable TLB Organization.

time and capacity trade-off observed for the cache also applies in the design of the TLB.

Figure 3.4 illustrates how a 512-entry, fully-associative TLB can be laid out to allow for reconfigurability. There are eight TLB increments, each of which contains a CAM of 64 virtual page numbers and an associated RAM of 64 physical page numbers. Switches are inserted on the input and output buses to electrically isolate successive increments. Thus, the ability to configure a larger TLB does not degrade the access time of the minimal size (64 entry) TLB. Similar to the cache design, TLB misses result in a second access but to the back-up portion of the TLB.

After every million-cycle interval, we examine hardware counters to determine the TLB size in the next interval. One counter keeps track of the TLB miss handler cycles and the L1 TLB size is doubled if this counter exceeds a threshold (3% in this study) of the total execution time. A single bit is also added to each TLB entry that is set to indicate if it has been used in an interval (and is flash cleared in hardware at the start of an interval). At the end of each interval, the number of TLB entries that have their bit set is counted. This can be done in hardware with fairly simple and energy-efficient logic. Similar logic that aggregates usage information within the issue queue has been proposed by Buyuktosunoglu et al. [Buyuktosunoglu *et al.*, 2001]. The

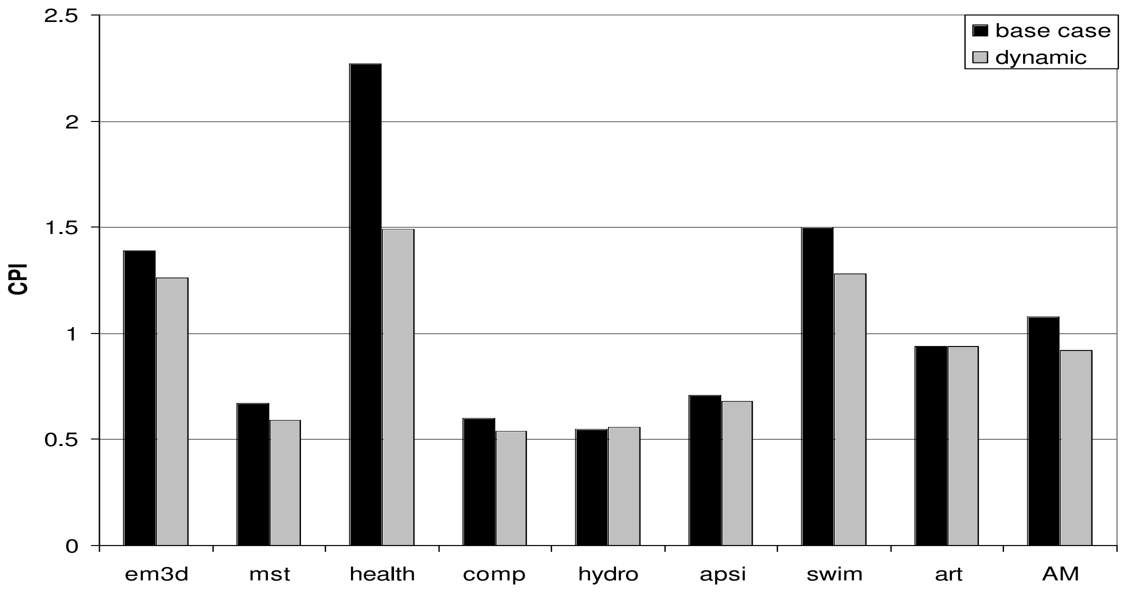


Figure 3.5: Cycles Per Instruction (CPI) results for the base case and for an interval-based mechanism for the cache and the TLB.

L1 TLB size is halved if the TLB usage is less than half. This mechanism can be classified as an interval-based technique using prediction. Note that an interval length of a million cycles has to be used so that there is sufficient time to collect an accurate estimate of TLB usage. For the cache, we use an interval-based mechanism with exploration. This algorithm uses the adaptive thresholds and a fixed interval length of 100K cycles. The simulation windows used in these experiments were also slightly smaller in some cases.

Figure 3.5 shows performance results when cache and TLB reconfiguration are simultaneously employed. The base case has a 64-entry L1 TLB and a 448-entry L2 TLB, while the configurable TLB has 64 entries when in its smallest configuration. We see that the overall processor performance improves by 15%. Table 3.4 attributes this improvement to either cache or TLB reconfiguration and lists the number of TLB size changes in each case. We see that the most significant improvements from TLB reconfiguration happen in *health*, *compress*, and *swim*. Health and compress perform best with 256 and 128 entries, respectively, and the dynamic scheme settles at these sizes. Swim shows phase change behavior with respect to TLB

	Cache contribution	TLB contribution	TLB changes		Cache contribution	TLB contribution	TLB changes
em3d	73%	27%	2	hydro2d	100%	0%	0
health	33%	67%	2	apsi	100%	0%	27
mst	100%	0%	3	swim	49%	51%	6
compress	64%	36%	2	art	100%	0%	5

Table 3.4: Contribution of the cache and the TLB to speedup or slow down in the dynamic scheme and the number of explorations.

usage, resulting in five stable phases requiring either 256 or 512 TLB entries.

3.4.4 Positional Adaptation

The primary advantage of using positional adaptation is to be able to target fine-grained phase changes. However, the cache does not react quickly to a reconfiguration. Due to this mismatch, we noticed that these techniques do not perform as well as the interval-based schemes. We attempted reconfigurations for every N^{th} branch and determined that best performance was seen when a phase change was signaled after every 100 branches or every 50 subroutine calls/returns. Because of this relatively coarse granularity, it takes a while for the predictions to be computed and used (we also record three samples of each event to minimize noise and boundary effects). In order to improve the granularity match, we attempted reconfigurations based on the length of the subroutine. We maintained a stack to keep track of the instructions executed within each subroutine and recorded statistics in a table only if the subroutine length exceeded 1000 dynamic instructions. Similar approaches have been employed by Huang et al. [Huang *et al.*, 2003] and Magklis et al. [Magklis *et al.*, 2003].

Figure 3.6 shows the use of positional adaptation with exploration for cache reconfiguration and also includes dynamic TLB management. For comparison, we also show results for the base case and the interval-based mechanism with exploration. As the results show, the simpler interval-based scheme usually outperforms the subroutine-based approach. If the application phase behavior is data or time-dependent rather than code location dependent, the subroutine-based scheme will be slower to adapt to the change. In addition, there is potential for instability

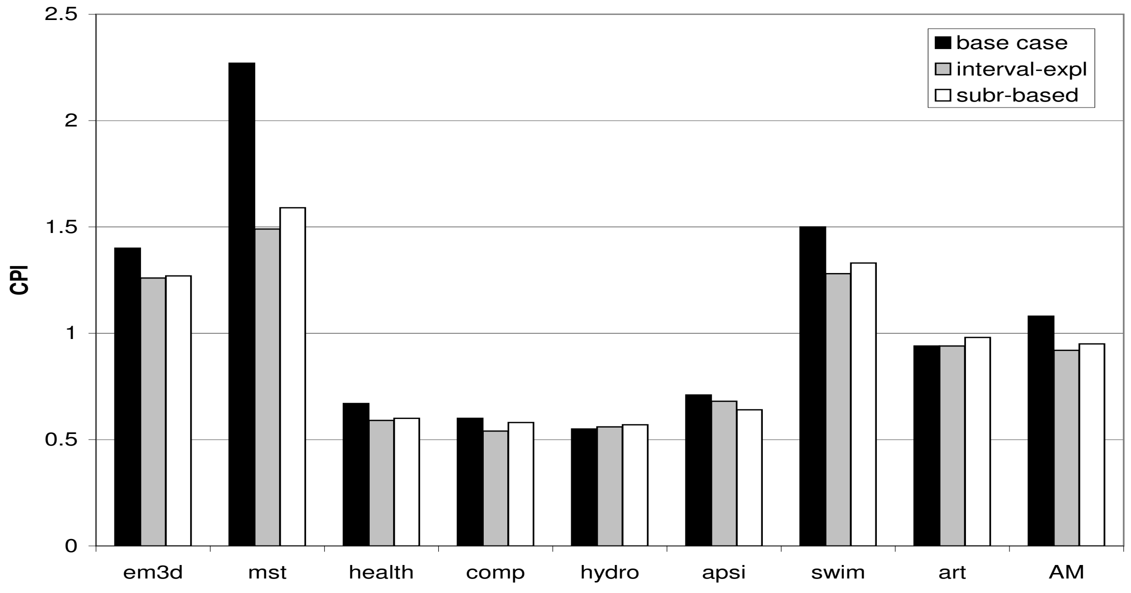


Figure 3.6: Cycles Per Instruction (CPI) results for the base case, for the interval-based mechanism with exploration and the subroutine-based technique with exploration.

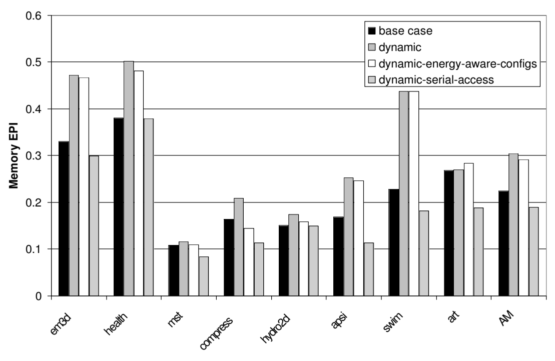
across subroutine invocations especially if the same procedure is called from multiple locations or phases in the program. The exception in our benchmark suite is *apsi*, for which the subroutine-based scheme improves performance relative to the interval-based approach as each subroutine exhibits consistent behavior across subroutine invocations. With the interval-based scheme, *apsi* shows inconsistent behavior across intervals, causing it to thrash between a 256KB L1 and a 768KB L1. However, the interval-based scheme is better able to capture application behavior on average than the subroutine-based scheme, in addition to being more practical since it requires simpler hardware.

3.4.5 Energy Consumption

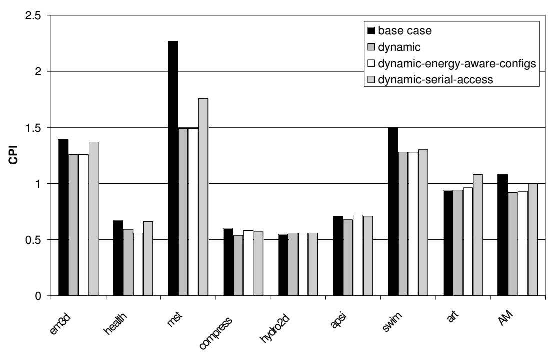
In our study so far, the dynamic schemes try to maximize performance. This can also have implications for energy consumption. If some of the cache organizations are more energy-efficient than the base case, the use of those organizations may result in improved performance and improved energy utilization. There are two energy-aware modifications to the selection mechanisms that we consider. The first takes advantage of the inherently low-energy configurations (those with direct-mapped 512KB and two-way set associative 1MB L1 caches). With this approach, the selection mechanism simply uses these configurations in place of the 768KB 3-way L1 and 1MB 4-way L1 configurations. The second approach is to serially access the tag and data arrays of the L1 data cache, as is the case for the L2 caches. Conventional L1 caches always perform parallel tag and data look-up to reduce hit time, thereby reading data out of multiple cache ways and ultimately discarding data from all but one way. By performing the tag and data look-up in series, only the data way associated with the matching tag can be accessed, thereby reducing energy consumption. Hence, if we assume that the base case is direct-mapped and all the set-associative configurations use serial tag and data access, every organization consumes the same energy for every read and write. However, the set-associative organizations have relatively longer access times because of the serialization. In spite of this increased access time, using these larger organizations may improve performance because of the reduced miss times that they afford. Further, because of the reduced miss rate and the fewer transfers between L1 and L2, the larger cache organizations can also reduce energy consumption.

We estimate cache and TLB energy dissipation using a modified version of the analytical model of Kamble and Ghose [Kamble and Ghose, 1997]. This model calculates cache energy dissipation using similar technology and layout parameters as those used by the timing model (including voltages and all electrical parameters appropriately scaled for 0.1μm technology). Note that we assume serial tag and data access for every L2 look-up, so the bitlines for at most one way are precharged and discharged. The TLB energy model was derived from this model and included CAM match line precharging and discharging, CAM wordline and bitline energy dissipation, as well as the energy of the RAM portion of the TLB. For main memory, we include only the energy dissipated due to driving the off-chip capacitive busses. Detailed event counts were captured during the simulations of each benchmark. These event counts include all cache and TLB operations and are used to obtain final energy estimations.

Figure 3.7(a) shows the memory energy per instruction (memory EPI) consumed for the base case and for three interval and exploration-based techniques. The first is for the best-performing parallel tag and data access organizations as before, the second is for the most energy-efficient parallel tag and data access organizations, and the third uses serial tag and data



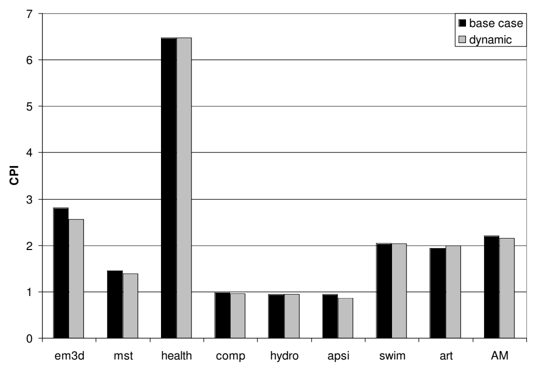
(a) Memory EPI results.



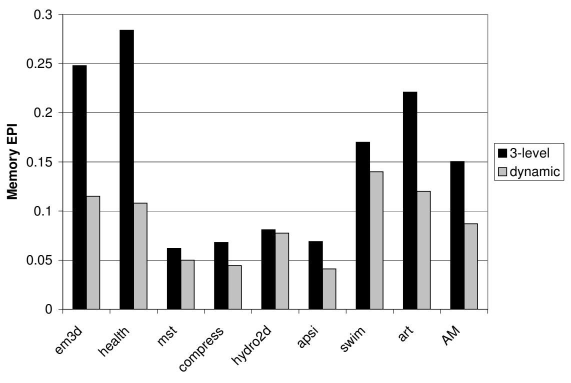
(b) CPI results.

Figure 3.7: (a) Memory Energy per Instruction (EPI) results for the base case, for the dynamic scheme with the best performing cache configurations, for the dynamic scheme with the most energy-efficient cache configurations, and for the dynamic scheme that employs serial tag and data access. (b) CPI results for the same four cases.

access for the set-associative caches to make them more energy-efficient. In the first case, the use of the larger caches results in higher energy consumption than the base case (35%) as the set-associativity increases the energy consumed per access, compared to the base case. Every memory access results in all ways being accessed for larger L1s, while for the base case, misses in the L1 result in accesses to the L2, which use serial tag and data access. In the second case, we observe that merely selecting the energy-aware configurations has only a nominal impact on energy. In the third case, by using serial tag and data access, the memory EPI reduces by 38% relative to the dynamic mechanism and by 16% relative to the base case. Note that the exploration process continues to pick the best performing organization, so this improvement in energy consumption happens in tandem with improved performance. Figure 3.7(b) shows the performance results for these organizations. In spite of the longer access times for the set-associative caches with serial access, the overall performance is 7% better than the base case. Thus, the use of the energy-efficient cache organizations with serial access provides balanced improvements in both performance and energy in portable applications where design constraints such as battery life are of utmost importance. Furthermore, as with the dynamic voltage and frequency scaling approaches used today, this mode may be switched on under particular environmental conditions (*e.g.*, when remaining battery life drops below a given threshold), thereby providing on-demand energy-efficient operation.



(a) CPI results.



(b) Memory EPI results.

Figure 3.8: (a) CPI results for the base three-level cache and for the dynamic scheme with the dynamic L2-L3. (b) Memory Energy per Instruction (EPI) results for the same two experiments.

3.4.6 L2/L3 Reconfiguration

In sub-0.1μm technologies, the long access latencies of a large on-chip L2 cache [Agarwal *et al.*, 2000] may be prohibitive for those applications which make use of only a small fraction of the L2 cache. Thus, for performance reasons, a three-level hierarchy with a moderate size (*e.g.*, 512KB) L2 cache will become an attractive alternative to two-level hierarchies at these feature sizes. However, the cost may be a significant increase in energy dissipation due to transfers involving the additional cache level. The use of the reconfigurable cache structure as a replacement for conventional L2 and L3 caches can help match the L2 size to the program's working set size. This helps reduce the number of transfers between L2 and L3, thereby significantly reducing energy dissipation without compromising performance. Note that L2s typically employ serial tag and data access, so using a larger set-associative L2 does not incur an energy cost. To further reduce the energy consumption, we modified the search mechanism to pick a larger sized cache if its performance was within 95% of the best performing cache during the exploration.

We used latencies corresponding to a model at 0.035μm technology. The L1 caches are 32KB 2-way with a three cycle latency, the L2 is 512KB 2-way with a 21 cycle latency, and the L3 is 2MB 16-way with a 60 cycle latency. We do not attempt TLB reconfiguration in these experiments so as to not affect the latency for the L1 cache access. Figure 3.8 compares the performance and energy of the conventional three-level cache hierarchy with the configurable scheme. Since the L1 cache organization has the largest impact on cache hierarchy performance, as expected, there is little performance difference between the two, as each uses an identical conventional L1 cache. However, the ability of the dynamic scheme to adapt the L2/L3 configuration to the application results in a 42% reduction in memory EPI on average.

3.5 Related Work

In this section, we briefly describe other approaches to improving the efficiency of on-chip caches. In order to address the growing gap between memory and processor speeds, techniques such as non-blocking caches [Farkas and Jouppi, 1994] and hardware and software-based prefetching [Jouppi, 1990; Callahan and Porterfield, 1990; Mowry *et al.*, 1992] have been proposed to reduce memory latency. However, their effectiveness can be greatly improved by changing the underlying structure of the memory hierarchy.

Recently, Ranganathan, Adve, and Jouppi [Ranganathan *et al.*, 2000] proposed a reconfigurable cache in which a portion of the cache could be used for another function, such as an instruction reuse buffer. Although the authors show that such an approach only modestly increases cache access time, fundamental changes to the cache may be required so that it may be used for other functionality as well, and long wire delays may be incurred in sourcing and sinking data from potentially several pipeline stages.

Dahlgren and Stenstrom [Dahlgren and Stenstrom, 1991] describe a cache whose organization can be changed by the compiler on a per-application basis. To handle conflict misses in a direct-mapped cache, they break the cache into multiple subunits and map different virtual address regions to these different subunits. This changes the way the cache is indexed. They also propose using different cache line sizes for different address ranges. Veidenbaum et al. [Veidenbaum *et al.*, 1999] also talk about such a reconfigurable cache, where the cache line size can be changed dynamically based on the spatial locality exhibited by the program. These changes are not done at the layout level – the cache has a small line size and depending on the program needs, an appropriate number of adjacent cache lines are fetched on a miss.

Albonesi [Albonesi, 1999] proposed the disabling of data cache ways for programs with small working sets to reduce energy consumption. A similar proposal by Yang et al. [Yang *et al.*, 2001] that reduces the number of sets in an instruction cache helps reduce leakage power for programs with small instruction working sets.

The reconfigurable cache is effective at reducing cache energy consumption in certain cases. A number of circuit-level and architectural techniques can be employed to reduce dynamic and leakage energy in caches. At the circuit level, careful transistor sizing and lowering of V_{dd} can be used to reduce dynamic energy. Simultaneous to the above circuit-level techniques, architectural techniques such as banking, serial tag and data access, and way prediction [Powell *et al.*, 2001], help lower dynamic energy consumption by reducing the number of transistors that are switched on each access. All the above techniques usually entail performance penalties.

Several combined circuit-level and architectural techniques have been proposed to reduce leakage energy while minimizing the performance loss. Higher V_t devices help reduce leakage energy [Nii *et al.*, 1998]. When applied statically to the entire cache, especially the L1 cache, these techniques increase the latency of access, however. Kaxiras *et al.* [Kaxiras *et al.*, 2001] present a control scheme that dynamically adjusts when V_{dd} is gated on a per cache line basis. They apply their algorithm to the L1 data cache and the L2 cache. The state of the cell is lost in the above schemes, and hence careful architectural control needs to be exercised in order to ensure that cells are turned off only when the probability of access is very low in order to avoid both energy and access time penalties. Agarwal et al. [Agarwal *et al.*, 2002] propose and employ a gated-ground SRAM cell that retains state even when put in standby mode. By gating ground, leakage energy consumption is reduced in cache lines that are not accessed. However, the cost is an increase in access time as well as in dynamic energy, and additional design complexity in the cache arrays. Multi-threshold CMOS (MT-CMOS [Nii *et al.*, 1998]) uses dynamic V_t scaling to reduce leakage energy consumption in caches. This also incurs additional energy as well as latency in switching between modes and results in increased fabrication complexity. Flautner et al. [Flautner *et al.*, 2002] propose a *drowsy* cache design for the L1 data cache that uses dynamic V_{dd} scaling, resulting in higher leakage current than with gated V_{dd}. Using controlled scaling, the state of the memory cell is maintained, allowing more aggressive use of the sleep mode. However, access to a cell in sleep mode incurs additional latency to restore V_{dd} since the cell can only be read in high V_{dd} mode. Their study concludes that a simple control scheme suffices to achieve most of the energy savings. Heo et al. [Heo *et al.*, 2002] take a novel approach to reduce the static energy associated with the bitlines in a RAM by simply tristating the drivers to the lines.

The reconfigurable cache layout described here was also employed in the *Accounting Cache* design of Dropsho et al. [Dropsho *et al.*, 2002]. The *Accounting Cache* uses LRU information to accurately estimate the miss rate for each cache organization and hence, the time and energy spent. Accordingly, lower-associative and lower-energy caches are selected, while maintaining performance within a specified percentage of the base case. When employed for all the on-chip caches, cache energy savings of 40% were observed, while incurring a performance loss of only 1.1%.

Various works [Srinivasan and Lebeck, 1999; Fisk and Bahar, 1999; Srinivasan *et al.*, 2001] have characterized load latency tolerance and metrics for identifying critical loads. Such metrics could prove useful in determining the cache requirements for a program phase (tolerance to a longer hit latency, tolerance to cache misses, etc), but we found that such hints do not improve the performance of the selection mechanisms [Balasubramonian *et al.*, 2000a; Balasubramonian *et al.*, 2003a].

3.6 Summary

This chapter has described a novel configurable cache and TLB as a higher performance and lower energy alternative to conventional on-chip memory hierarchies. Cache and TLB reconfiguration is effected by leveraging repeater insertion to allow dynamic speed/size trade-offs while limiting the impact of speed changes to within the memory hierarchy. We evaluate the effect of different adaptation algorithms on this cache and our results demonstrate that a simple interval-based technique is sufficient to achieve good performance. The algorithm is able to dynamically balance the trade-off between an application's hit and miss intolerance, thereby maximizing performance for each program phase. Information on the LRU nature of accesses can help predict the optimal organization without profiling each candidate organization. Positional adaptation is not as effective in this context because of the slow reaction of the hardware to any reconfiguration. Its effectiveness can, however, be improved by employing adaptation only for large subroutines [Balasubramonian *et al.*, 2003a]. At 0.1μm technologies, our results show an average 15% reduction in CPI in comparison with a conventional L1-L2 design of comparable total size, with the benefit almost equally attributable on average to the configurable cache and TLB. Furthermore, energy-aware enhancements to the algorithm trade off a more modest performance improvement for a significant reduction in energy. Projecting to a 3-level cache hierarchy potentially necessitated by sub-micron technologies, we show an average 42% reduction in memory hierarchy energy at 0.035μm technology when compared to a conventional design.

4 Trade-Offs in Clustered Microprocessors

In this chapter, we evaluate trade-offs in the design of communication-bound processors of the future. In this context, current technology trends facilitate the implementation of hardware reconfiguration. We employ the adaptation algorithms introduced in Chapter 2 to dynamically manage the trade-offs.

4.1 Technology Trends

The extraction of large amounts of instruction-level parallelism (ILP) from common applications on modern processors requires the use of many functional units and large on-chip structures such as issue queues, register files, caches, and branch predictors. As CMOS process technologies continue to shrink, wire delays become dominant (compared to logic delays) [Agarwal *et al.*, 2000; Matzke, 1997; Palacharla *et al.*, 1997]. This, combined with the continuing trend towards faster clock speeds, increases the time in cycles to access regular on-chip structures (caches, register files, etc.). Not only does this degrade instructions per cycle (IPC) performance, it also presents various design problems in breaking up the access into multiple pipeline stages. In spite of the growing numbers of transistors available to architects, it is becoming increasingly difficult to design large monolithic structures that aid ILP extraction without increasing design complexity, compromising clock speed, and limiting scalability in future process technologies.

A potential solution to these design challenges is a *clustered microarchitecture* [Farkas *et al.*, 1997; Palacharla *et al.*, 1997] in which the key processor resources are distributed across multiple clusters, each of which contains a subset of the issue queues, register files, and the functional units. In such a design, at the time of instruction rename, each instruction is steered into one of the clusters. As a result of decreasing the size and bandwidth requirements of the issue queues and register files, the access times of these cycle-time critical structures are greatly reduced, thereby permitting a faster clock. The simplification of these structures also reduces their design complexity.

An attractive feature of a clustered microarchitecture is the reduced design effort in producing successive generations of a processor. Not only is the design of a single cluster greatly simplified, but once a single cluster core has been designed, more of these cores can be put into the processor for a low design cost (including increasing front-end bandwidth) as the transistor budget increases. Adding more clusters could potentially improve IPC performance because each program has more resources to work with. There is little effect if any on clock speed from doing this as the implementation of each individual cluster does not change. In addition, even if the resources in a large clustered processor cannot be effectively used by a single thread, the scheduling of multiple threads on a clustered processor can significantly increase the overall instruction throughput. The relatively low design complexity and the potential to exploit thread-level parallelism make a highly-clustered processor in the billion transistor era an extremely attractive option.

The primary disadvantage of clustered microarchitectures is their reduced IPC compared to a monolithic design with identical resources. Although dependent instructions within a single cluster can issue in successive cycles, extra inter-cluster bypass delays prevent dependent instructions that lie in different clusters from issuing in successive cycles. While monolithic processors might use a potentially much slower clock to allow a single-cycle bypass among all functional units, a clustered processor allows a faster clock, thereby introducing additional latencies in cycles between some of the functional units. The clustered design is a viable option only if the IPC degradation does not offset the clock speed improvement.

Modern processors like the Alpha 21264 [Kessler, 1999] at 0.35μ technology already employ a limited clustered design, wherein the integer domain, for example, is split into two clusters. A number of recent studies [Aggarwal and Franklin, 2001; Baniasadi and Moshovos, 2000; Canal *et al.*, 2000; Canal *et al.*, 2001; Farkas *et al.*, 1997] have explored the design of heuristics to steer instructions to clusters. Despite these advances, the results from these studies will likely need to be reconsidered in the near future for the following reasons:

- Due to the growing dominance of wire delays [Matzke, 1997; Palacharla *et al.*, 1997] and the trend of increasing clock speeds, the resources in each cluster core will need to be significantly reduced relative to those assumed in prior studies.
- There will be more clusters on the die than assumed in prior studies due to larger transistor budgets and the potential for exploiting thread-level parallelism [Tullsen *et al.*, 1995].
- The number of cycles to communicate data between the furthest two clusters will increase due to the wire delay problem [Agarwal *et al.*, 2000]. Furthermore, communication delays will be heterogeneous, varying according to the position of the producer and consumer nodes.
- The data cache will need to be distributed among clusters, unlike the centralized cache assumed by most prior studies, due to increased interconnect costs and the desire to scale the cache commensurately with other cluster resources.

While the use of a large number of clusters could greatly boost overall throughput for a multi-threaded workload, its impact on the performance of a single-threaded program is not as evident. The cumulative effect of the above trends is that clustered processors will be much more *communication bound* than assumed in prior models.

As the number of clusters on the chip increases, the number of resources available to the thread also increases, supporting a larger window of in-flight instructions and thereby allowing more distant instruction-level parallelism (ILP) to be exploited. At the same time, the various instructions and data of the program get distributed over a larger on-chip space. If data has to be communicated across the various clusters frequently, the performance penalty from this increased communication can offset any benefit derived from the parallelism exploited by additional resources.

In this chapter, we present and evaluate a dynamically tunable clustered architecture that attempts to optimize the communication-parallelism trade-off for improved single-threaded performance in the face of the above trends. The balance is effected by employing only a subset of the total number of available clusters for the thread. Our results show that the performance trend as a function of the number of clusters varies across different programs depending on the degree of distant ILP present in them. This motivates the need for dynamic algorithms that identify the optimal number of clusters for any program phase and match the hardware to the program's requirements. Our evaluation studies the performance benefits and the overhead of employing the algorithms described in Chapter 2.

Disabling a subset of the clusters for a given program phase in order to improve single-threaded performance has other favorable implications. Entire clusters can turn off their supply voltage, thereby greatly saving on leakage energy, a technique that would not have been possible in a monolithic processor. Alternatively, these clusters can be used by (partitioned among) other threads, thereby simultaneously achieving the goals of optimal single and multi-threaded throughput.

4.2 The Base Clustered Processor Architecture

We start by describing a baseline clustered processor model that has been commonly used in earlier studies [Aggarwal and Franklin, 2001; Baniasadi and Moshovos, 2000; Canal *et al.*, 2000; Canal *et al.*, 2001; Farkas *et al.*, 1997]. Such a model with four clusters is shown in Figure 4.1. The branch predictor and instruction cache are centralized structures, just as in a conventional processor. At the time of register renaming, each instruction gets assigned to a specific cluster. Each cluster has its own issue queue, register file, a set of functional units, and its own local bypass network. Bypassing of results within a cluster does not take additional cycles (in other words, dependent instructions in the same cluster can issue in successive cycles). However, if the consuming instruction is not in the same cluster as the producer, it has to wait additional cycles until the result is communicated across the two clusters.

A conventional clustered processor [Aggarwal and Franklin, 2001; Baniasadi and Moshovos, 2000; Canal *et al.*, 2000; Canal *et al.*, 2001; Farkas *et al.*, 1997] distributes only the register file, issue queue, and the functional units among the clusters. The data cache is centrally located. An alternative organization [Zyuban and Kogge, 2001] distributes the cache among the clusters, thereby making the design more scalable, but also increasing the implementation complexity.

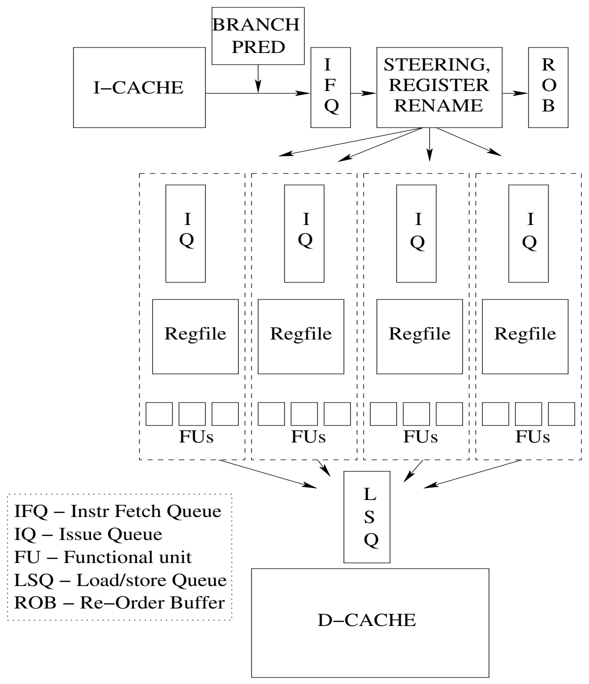


Figure 4.1: The base clustered processor (4 clusters) with the centralized cache.

Since both organizations are attractive design options, we evaluate the effect of dynamic tuning on both organizations.

4.2.1 The Centralized Cache

In the traditional clustered designs, once loads and stores are ready, they are inserted into a centralized load-store queue (LSQ) (Figure 4.1). From here, stores are sent to the centralized L1 cache when they commit and loads are issued when they are known to not conflict with earlier stores. The LSQ is centralized because a load in any cluster could conflict with an earlier store from any of the other clusters.

For the aggressive processor models that we are studying, the cache has to service a number of requests every cycle. An efficient way to implement a high bandwidth cache is to make it word-interleaved [Rivers *et al.*, 1997]. For a 4-way word-interleaved cache, the data array is split into four banks and each bank can service one request every cycle. Data with word addresses (where a word is eight bytes long) of the form 4N are stored in bank 0, of the form 4N+1 are stored in bank 1, and so on. Such an organization supports a maximum bandwidth of four accesses in a cycle so long as these accesses are all to different banks. A word-interleaved banked cache minimizes conflicts to a bank and hence usually outperforms other alternatives like a replicated or line-interleaved cache [Rivers *et al.*, 1997].

In a processor with a centralized cache, the load latency depends on the distance between the centralized cache and the cluster issuing the load. In our study, we assume that the centralized LSQ and cache are co-located with cluster 1. Hence, a load issuing from cluster 1 does not experience any communication cost. A load issuing from cluster 2 takes one cycle to send the address to the LSQ and cache and another cycle to get the data back (assuming that each hop between clusters takes a cycle). Similarly, cluster 3 experiences a total communication cost of four cycles for each load. This is in addition to the few cycles required to perform the cache RAM look-up.

A clustered design allows a faster clock, but incurs a noticeable IPC degradation because of inter-cluster communication and load imbalance. Minimizing these penalties with smart instruction steering has been the focus of many recent studies [Aggarwal and Franklin, 2001; Baniasadi and Moshovos, 2000; Canal *et al.*, 2000; Canal *et al.*, 2001; Capitanio *et al.*, 1992; Farkas *et al.*, 1997]. We use an effective steering heuristic [Canal *et al.*, 2000] that steers an instruction (and its destination register) to the cluster that produces most of its operands. In the event of a tie or under circumstances where an imbalance in issue queue occupancy is seen, instructions are steered to the least loaded cluster. By picking an appropriate threshold to detect load imbalance, such an algorithm can also approximate other proposed steering heuristics like Mod_N and $First\_Fit$ [Baniasadi and Moshovos, 2000]. The former minimizes load imbalance by steering N instructions to one cluster, then steering to its neighbor. The latter minimizes communication by filling up one cluster before steering instructions to its neighbor. We empirically determined the optimal threshold value for load balance. Further, our steering heuristic also uses a criticality predictor [Fields *et al.*, 2001; Tune *et al.*, 2001] to give a higher priority to the cluster that produces the critical source operand. Thus, our heuristic represents the state-of-the-art in steering mechanisms.

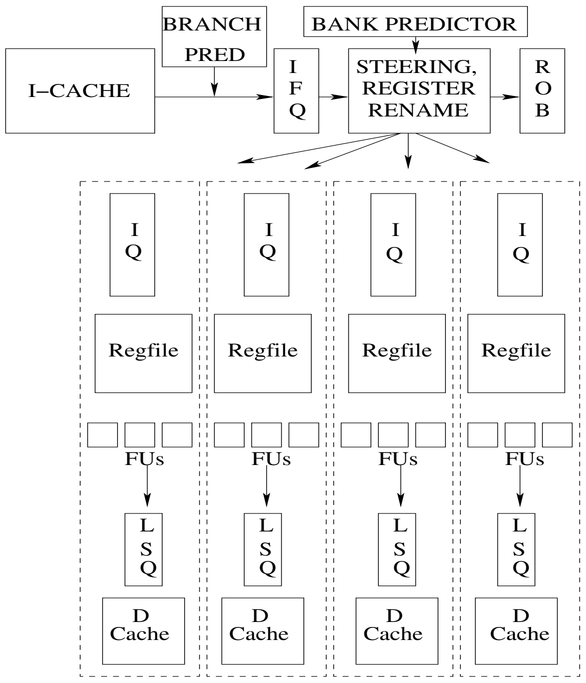


Figure 4.2: The clustered processor (4 clusters) with the decentralized cache.

4.2.2 The Decentralized Cache

In a highly clustered processor, the centralized cache can be a major bottleneck - (i) The centralized structure has to service requests from a number of clusters and implementing a large, high bandwidth cache imposes access time penalties. (ii) The average load latency goes up because of the distance between the cache and the requesting cluster. (iii) The contention for the interconnect increases because each access requires two transfers. Hence, a distributed cache model [Zyuban and Kogge, 2001] represents an attractive design option.

For an N-cluster system, we assume that the L1 cache is broken into N word-interleaved banks. Each bank is associated with its own cluster. The LSQ is also split across the different clusters. The example in Figure 4.2 shows an organization with four clusters. Because they are word-interleaved, the various banks cache mutually exclusive data and do not require any cache coherence protocol between them. The goal of the steering mechanism is to steer a load or store to the cluster that caches the corresponding memory address. That way, the instruction naturally progresses from its issue queue, to the corresponding LSQ, and then to its own cache bank. It need not bother with loads and stores in other clusters that are necessarily handling different memory addresses. We discuss the additional steering complexities arising from the distributed nature of the cache in Section 4.4.3.

The L2 cache continues to be co-located with cluster 1 and a miss in any of the L1 cache banks other than that associated with this cluster incurs additional latency depending on the number of hops.

4.2.3 Interconnects

As process technologies shrink and the number of clusters is increased, attention must be paid to the communication delays and interconnect topology between clusters. Cross-cluster communication occurs at the front-end as well as when communicating register values across clusters or when accessing the cache. Since the former occurs in every cycle, we assume a separate network for this purpose and model non-uniform dispatch latencies as well as the additional latency in communicating a branch mispredict back to the front-end. Since the latter two (cache and register-to-register communication) involve data transfer to/from registers, we assume that the same (separate) network is used.

In our study, we focus on a ring interconnect because of its low implementation complexity. Each cluster is directly connected to two other clusters. We assume two uni-directional rings, implying that a 16-cluster system has 32 total links (allowing 32 total transfers in a cycle), with the maximum number of hops between any two nodes being 8.

In a later section, as part of our sensitivity analysis, we also show results for a grid interconnect, which has a higher implementation cost but higher performance. The clusters are laid out in a two-dimensional array. Each cluster is directly connected to up to four other clusters. For 16 clusters, there are 48 total links, with the maximum number of hops being 6, thus reducing the overall communication cost.

Fetch queue size	64
Branch predictor	comb. of bimodal and 2-level
Bimodal predictor size	2048
Level 1 predictor	1024 entries, history 10
Level 2 predictor	4096 entries
BTB size	2048 sets, 2-way
Branch mispredict penalty	at least 12 cycles
Fetch width	8 (across up to two basic blocks)
Dispatch and commit width	16
Issue queue size	15 in each cluster (int and fp, each)
Register file size	30 in each cluster (int and fp, each)
Re-order Buffer (ROB) size	480
Integer ALUs/mult-div	1/1 (in each cluster)
FP ALUs/mult-div	1/1 (in each cluster)
L1 I-cache	32KB 2-way
L2 unified cache	2MB 8-way, 25 cycles
TLB	128 entries, 8KB page size (I and D)
Memory latency	160 cycles for the first chunk

Table 4.1: Simplescalar simulator parameters.

4.3 Methodology

4.3.1 Simulator Parameters

Our simulator is based on Simplescalar-3.0 [Burger and Austin, 1997] for the Alpha AXP instruction set and is similar to the one used in Chapter 3. The processor parameters have been modified to reflect projections of future designs and are summarized in Table 4.1. The simulator has been modified to represent a microarchitecture resembling the Alpha 21264 [Kessler, 1999]. The register update unit (RUU) is decomposed into issue queues, physical register files, and the reorder buffer (ROB). The issue queue and the physical register file are further split into integer and floating-point. Thus, each cluster in our study is itself decomposed into an integer and floating-point cluster. The memory hierarchy is also modeled in detail (including word-interleaved access, bus and port contention, writeback buffers, etc).

This base processor structure was modified to model the clustered microarchitecture. To represent a wire-delay constrained processor at future technologies, each cluster core was assumed to have one functional unit of each type, 30 physical registers (int and fp, each), and 15 issue queue entries (int and fp, each). As many instructions can issue in a cycle as the number of available functional units. We assume that each hop on the interconnect takes a single cycle. While we did not model a trace cache, we assumed that instructions could be fetched from up to two basic blocks at a time.

The number of resources in each cluster and the latency for each hop on the interconnect are critical parameters in such a study as they determine the amount and cost of inter-cluster communication. These parameters are highly technology, layout, and design-dependent, and determining them is beyond the scope of this study. Our results include a sensitivity analysis to see how the results change as our assumptions on the number of registers, issue queue entries, functional units, and cycles per hop are varied. We use instructions per cycle (IPC) to represent program performance as it is directly indicative of the instruction-level parallelism in the program.

Our study focuses on wire-limited technologies of the future and we pick latencies according to projections for 0.035μ. We used CACTI-3.0 [Shivakumar and Jouppi, 2001] to estimate access times for the cache organizations. We used the methodology in [Agarwal *et al.*, 2000] to

Parameter	Centralized cache	Decentralized cache each cluster	Decentralized cache total
Cache size	32 KB	16 KB	16N KB
Set-associativity	2-way	2-way	2-way
Line size	32 bytes	8 bytes	8N bytes
Bandwidth	4 words/cycle	1 word/cycle	N words/cycle
RAM look-up time	6 cycles	4 cycles	4 cycles
LSQ size	15N	15	15N

Table 4.2: Cache parameters for the centralized and decentralized caches. All the caches are word interleaved. N is the number of clusters.

estimate clock speeds and memory latencies, following SIA roadmap projections [Association, 1999]. With Simplescalar, we simulated cache organizations with different size and port parameters (and hence different latencies) to determine the best base cases. These parameters are summarized in Table 4.2. The centralized cache yielded best performance for a 4-way word-interleaved 32KB cache. Such a cache has a bandwidth of four accesses per cycle and an access time of six cycles. The best decentralized cache organization has a single-ported four-cycle 16KB bank in each cluster.

4.3.2 Benchmark Set

Our proposed design exploits the fact that different programs in most benchmark sets have vastly different parallelism characteristics. To limit the simulation effort, we restrict ourselves to a subset of programs that are representative of these diverse characteristics. These programs that are described in Table 4.3 include four SPEC2K Integer programs, three SPEC2K FP programs, and two programs from the UCLA Mediabench [Lee *et al.*, 1997]. The programs represent a mix of various program types, including high and low IPC codes, and those limited by memory, branch mispredictions, etc. We focus our discussions on these programs because they show high variability in interval lengths, as discussed in Chapter 2. We also verify some of our results for all of SPEC2k. They were compiled with Compaq's cc, f77, and f90 compilers for the Alpha

Benchmark	Input dataset	Simulation window	Base IPC	Mispred branch interval
cjpeg (Mediabench)	testimg	150M-250M	2.06	82
crafty (SPEC2k Int)	ref	2000M-2200M	1.85	118
djpeg (Mediabench)	testimg	30M-180M	4.07	249
galgel (SPEC2k FP)	ref	2000M-2300M	3.43	88
gzip (SPEC2k Int)	ref	2000M-2100M	1.83	87
mgrid (SPEC2k FP)	ref	2000M-2050M	2.28	8977
parser (SPEC2k Int)	ref	2000M-2100M	1.42	88
swim (SPEC2k FP)	ref	2000M-2050M	1.67	22600
vpr (SPEC2k Int)	ref	2000M-2100M	1.20	171

Table 4.3: Benchmark description. Baseline IPC is for a monolithic processor with as many resources as the 16-cluster system. ”Mispred branch interval” is the number of instrs before a branch mispredict is encountered.

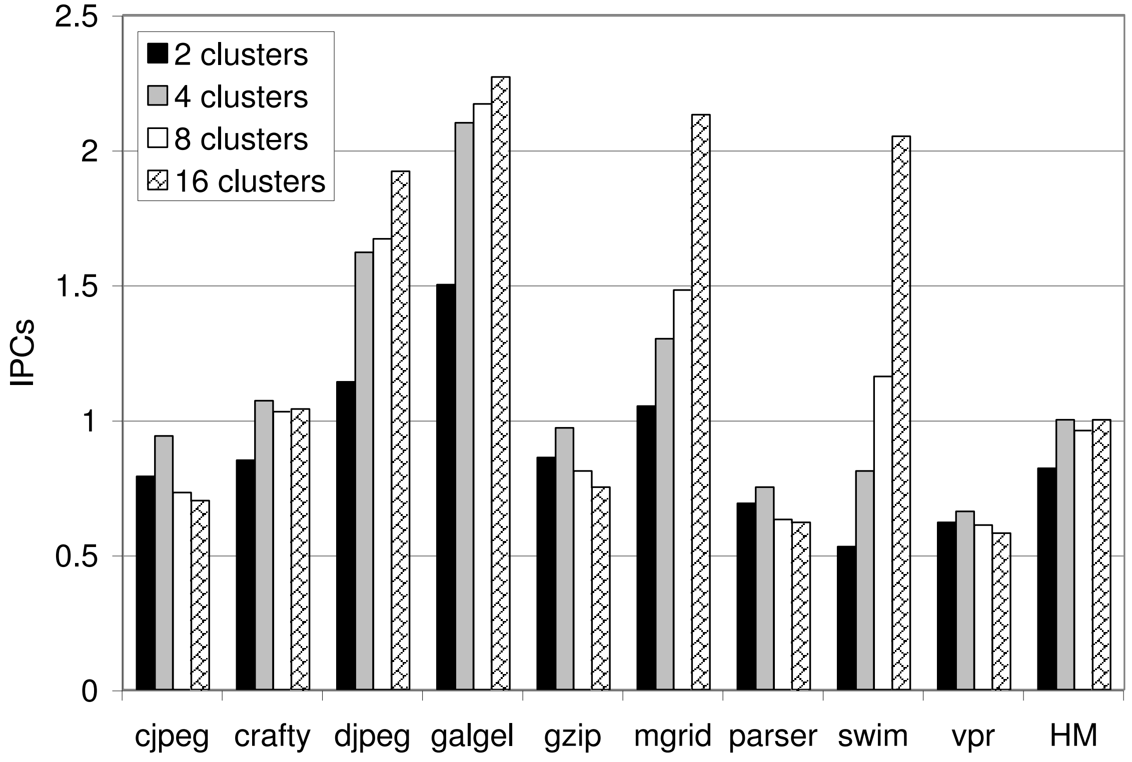


Figure 4.3: IPCs for fixed cluster organizations with 2, 4, 8, and 16 clusters.

21164 at the highest optimization level. Most of these programs were fast forwarded through the first two billion instructions and simulated in detail to warm the various processor structures before measurements were taken. Because of the high complexity in simulating 16 clusters, we restrict our simulation windows to at most 300M instructions for any program. While we are simulating an aggressive processor model, not all our benchmark programs have a high IPC. Note that an aggressive processor design is motivated by the need to run high IPC codes and by the need to support multiple threads. In both cases, the quick completion of a single low-IPC thread is still important – hence the need to include such programs in the benchmark set.

4.4 Evaluation

4.4.1 The Dynamically Tunable Clustered Design

For brevity, we focus our initial analysis on the 16-cluster model with the centralized cache and the ring interconnect. Figure 4.3 shows the effect of statically using a fixed subset of clusters for a program. Increasing the number of clusters increases the average distance of a load/store instruction from the centralized cache and the worst-case inter-cluster bypass delay, thereby greatly affecting the overall communication cost. Assuming zero inter-cluster communication cost for loads and stores improved performance by 31%, while assuming zero cost for register-to-register communication improved performance by 11%, indicating that increased load/store latency dominates the communication overhead. This latency could be reduced by steering load/store instructions to the cluster closest to the cache, but this would increase load imbalance and register communication. The average latency for inter-cluster register communication in the 16-cluster system was 4.1 cycles. At the same time, using more clusters also provides the program with more functional units, registers, and issue queue entries, thus allowing it to dispatch a larger window of in-flight instructions. Depending on which of these two conflicting forces dominates, performance either improves or worsens as the number of clusters is increased. Programs with distant ILP, like *djpeg* (JPEG decoding from Mediabench), *swim*, *mgrid*, and *galgel* (loop-based floating-point programs from SPEC2K) benefit from using many resources. On the other hand, most integer programs with low branch prediction accuracies can not exploit a large window of in-flight instructions. Hence, increasing the resources only degrades performance because of the additional communication cost. This is a phenomenon hitherto unobserved in a clustered processor (partly because very few studies have looked at more than four clusters and partly because earlier studies assumed no communication cost in accessing a centralized cache).

Our goal is to tune the hardware to the program's requirements by dynamically allocating clusters to the program. This can be very trivially achieved by modifying the steering heuristic to disallow instruction dispatch to the disabled clusters. In other words, disabling is equivalent to not assigning any new instructions to the cluster. Instructions already assigned to the disabled clusters are allowed to complete, resulting in a natural draining of the cluster.

4.4.2 Comparing the Dynamic Algorithms

Chapter 2 describes four different adaptation algorithms, based on the reconfiguration points and on how they pick an optimal configuration. Figure 4.4 shows the IPC performance for these different algorithms and for two base cases. The first two bars represent the base cases – clustered organizations that have a fixed set of 4 and 16 clusters, respectively. Note that in terms of overall harmonic mean performance across the benchmark set, both base cases are comparable. Next, we discuss the behavior of each of the dynamic algorithms.

Using Intervals and Exploration. The third bar in Figure 4.4 illustrates the impact of using the interval-based selection mechanism with exploration at the start of each program phase. The exploration process lasts four intervals and profiles behavior with 2, 4, 8, and 16 clusters before selecting the best. We see that in almost all cases, the dynamic scheme does a very good job in approximating the performance of the best static organization. For floating-point programs with little instability (*galgel, mgrid, swim*), the dynamic scheme easily matches the hardware to the program's requirements. For the integer programs, in most cases, there is an initial unstable period when the interval size is inappropriate. Consistent with our analysis in Chapter 2, the interval size is increased until it settles at one that allows an *instability factor* of less than 5%. In *parser*, the simulation interval was not long enough to allow the dynamic scheme to settle at the required 40M instruction interval.

In *djpeg*, it takes a number of intervals for the interval size to be large enough (1.28M instructions) to allow a small *instability factor*. Further, since the interval length is large, many opportunities for reconfiguration are missed. There are small phases within each interval where the ILP characteristics are different. For these two reasons, the dynamic scheme falls short of the performance of the fixed static organization with 16 clusters for *djpeg*.

In the case of *gzip*, there are a number of prolonged phases, some with distant ILP characteristics, and others with low amounts of distant ILP. Since the dynamic scheme picks the best configuration at any time, its performance is better than even the best static fixed organization. Table 4.4 enumerates the number of phase changes encountered for each program and the most commonly selected organizations.

On average, 8.3 of the 16 clusters were disabled at any time across the benchmark set. In

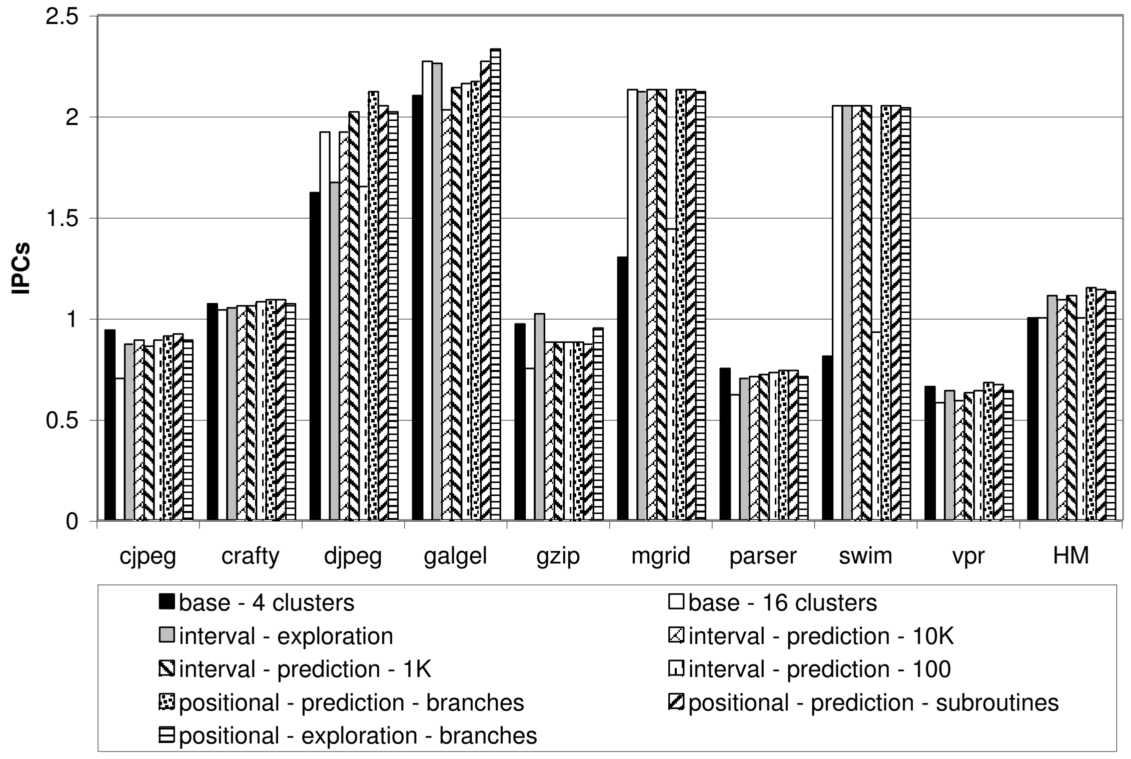


Figure 4.4: IPCs for the base cases and all the dynamic adaptation algorithms. The first two bars have a fixed set of clusters (4 and 16, respectively). The third bar represents the interval and exploration based mechanism. The fourth, fifth, and sixth bars represent interval-based mechanisms with prediction, for interval lengths of 10K, 1K, and 100 instructions, respectively. The seventh and eighth bars represent positional adaptation techniques with prediction, while reconfiguring at every 5th branch and every subroutine, respectively, while the ninth bar employs positional adaptation and exploration at every 5th branch.

Benchmark	Number of phase changes	Most commonly selected configurations
cjpeg	286	4, 16 clusters
crafty	53	4, 16 clusters
djpeg	24	16 clusters
galgel	4	16 clusters
gzip	295	4, 16 clusters
mgrid	0	16 clusters
parser	34	4 clusters
swim	0	16 clusters
vpr	422	4 clusters

Table 4.4: Number of phase changes encountered for each program.

the absence of any other workload, this produces a great savings in leakage energy, provided the supply voltage to these unused clusters can be turned off. Likewise, for a multi-threaded workload, even after optimizing single-thread performance, more than eight clusters still remain for use by the other threads.

Overall, the dynamic interval-based scheme with exploration performs about 11% better than the best static fixed organization. It is also very robust – it applies to every program in our benchmark set as there is usually a coarse enough interval length such that behavior across those intervals is fairly consistent. However, the downside is the inability to target relatively short phases. We experimented with smaller initial interval lengths, but found that the dynamic scheme encountered great instability at these small interval lengths, and hence, the interval lengths were increased to a larger value just as before. This is caused by the fact that measurements become noisier as the interval size is reduced and it is harder to detect the same program metrics across intervals and accordingly identify the best configuration for any phase.

Using Intervals and Prediction. To alleviate these problems, we investigate the use of hardware metrics to predict the optimal organization without going through an exploration process. Instead of exploring various configurations at the start of each program phase, we used a 16-cluster configuration for an interval and based on the degree of available distant ILP, we selected either a four or 16-cluster configuration for subsequent intervals until the next phase change (our earlier results indicate that these are the two most meaningful configurations and cover most cases). An instruction is marked as *distant* if it is at least 120 instructions younger than the oldest instruction in the ROB. At the time of issue, the instruction sets a bit in its ROB entry if it is *distant*. At the time of commit, this bit is used to increment the 'degree of distant ILP'. Since each cluster has 30 physical registers, four clusters are enough to support about 120 in-flight instructions. If the number of *distant* instructions issued in an interval exceeds a certain threshold, it indicates that 16 clusters would be required to exploit the available distant ILP. In our experiments, we use a threshold value of 160 for an interval length of 1000. Because there is no exploration phase, the hardware reacts quickly to a program phase change and reconfiguration at a finer granularity becomes meaningful. Hence, we focus on small fixed instruction intervals and do not attempt to increase the interval length at run-time. However, since the decision is based on program metrics instead of exploration, some accuracy is compromised. Further, the smaller the interval length, the faster the reaction to a phase change, but the noisier the measurements, resulting in some incorrect decisions.

The fourth, fifth, and sixth bars in Figure 4.4 represent such a mechanism for three different fixed interval lengths. An interval length of 1K instructions provides the best trade-off between accuracy and fast reactions to phase changes. Overall, it shows the same 11% improvement over the best static base case. However, in a program like *djpeg*, it does much better (21%) than the interval-based scheme with exploration because of its ability to target small phases with different requirements. Unfortunately, it takes a performance hit in programs like *galgel* and *gzip* because the small interval-length and the noisy measurements result in frequent phase changes and inaccurate decision-making.

One of the primary reasons for this is the fact that the basic blocks executed in successive 1000 instruction intervals are not always the same. As a result, frequent phase changes are signaled and each new phase change results in an interval with 16 clusters, to help determine the distant ILP.

Positional Adaptation with Prediction. To allow reconfiguration at a fine granularity, we attempt positional adaptation at branch and subroutine boundaries. We have to determine if a branch is followed by a high degree of distant ILP, in which case, dispatch should continue freely, else, dispatch should be limited to only the first four clusters. Exploring various configurations is not a feasible option as there are likely to be many neighboring branches in different stages of exploration resulting in noisy measurements for each branch. Hence, until we have enough information, we assume dispatch to 16 clusters and compute the distant ILP characteristics following every branch. This is used to update a *reconfiguration table* so that when the same branch is later encountered, it is able to pick the right number of clusters. If we encounter a branch with no entry in the table, we assume a 16-cluster organization so that we can determine its degree of distant ILP.

Assuming that four clusters can support roughly 120 instructions, to determine if a branch is followed by distant ILP, we need to identify how many of the 360 committed instructions following a branch were *distant* when they issued. Accordingly, either four or 16 clusters would be appropriate. To effect this computation, we keep track of the *distant* ILP nature of the 360 last committed instructions. A single counter can be updated by the instructions entering and leaving this queue of 360 instructions so that a running count of the distant ILP can be maintained. When a branch happens to be the oldest of these 360 instructions, its degree of distant ILP is indicated by the value in the counter.

There is likely to still be some interference from neighboring branches. To make the mechanism more robust, we sample the behavior for a number of instances of the same branch before creating an entry for it in the reconfiguration table. Further, we can fine-tune the granularity of reconfiguration by attempting changes only for specific branches. For example, we found that best performance was achieved when we attempted changes for only every fifth branch. We also show results for a mechanism that attempts changes only at subroutine calls and returns.

As discussed in Chapter 2, the downside of the approach just described is the fact that initial measurements dictate future behavior. The nature of the code following a branch could change over the course of the program. It might not always be easy to detect such a change, especially if only four clusters are being used and the degree of distant ILP is not evident. To deal with this situation, we flush the reconfiguration table at periodic intervals. We found that re-constructing the table every 10M instructions resulted in negligible overheads.

The seventh and eighth bars in Figure 4.4 represent two positional adaptation algorithms based on the *distant ILP* metrics. The first of these two algorithms attempts reconfiguration at every 5th branch and creates an entry in the table after collecting 10 samples for each branch. To eliminate effects from aliasing, we use a large 16K-entry table, though, in almost all cases, a much smaller table works as well. The second scheme attempts changes at every subroutine call and return and uses three samples. The figure indicates that the ability to quickly react to phase changes results in improved performance in programs like *djpeg, cjpeg, crafty, parser,* and *vpr*. The maximum number of configuration changes was observed for *crafty* (1.5 million). Unlike in the interval-based schemes with no exploration, instability is not caused by noisy measurements. However, *gzip* fails to match the performance achieved by the interval-based scheme. This is because the nature of the code following a branch changes over the course of the program. Hence, our policy of using initial measurements to pick a configuration for the future is not always accurate. The same behavior is observed to a lesser extent in *galgel*. Overall, the fine-grained schemes yield a 15% improvement over the base cases, compared to the 11% improvements seen with the interval-based schemes.

Positional Adaptation with Exploration. Finally, we study an example of positional adaptation while using an exploration process to pick the best organization. As described in Chapter 2, phase changes are signaled at specific branches or subroutines. Initially (and at regular 10M instruction intervals), the prediction table is cleared. For each branch PC encountered, we go through an exploration process, where we profile the performance of each candidate hardware organization for the instructions following the branch. A number of samples of each measurement are taken. Once all the organizations have been profiled, the best performing one is recorded in the prediction table and it is used every time the branch is encountered. We empirically determined that the best performance was observed when phase changes were signaled at every 20th branch and when 10 samples were recorded for each event.

The last bar in Figure 4.4 represents such an adaptation mechanism. Overall, the improvement is roughly 13%, slightly poorer than when using prediction with the *distant ILP* metric. This is primarily because measurements are being made across fairly short periods and neighboring phases might employ completely different configurations. This results in inaccuracies in the exploration process. However, the ability to react quickly to a new phase allows it to outperform the interval-based mechanisms.

Summary. We see that the interval-based schemes are reliable in that they are able to target all of the studied programs. They also achieve most of the improvements possible with run-time adaptation and entail negligible hardware overheads. The techniques involving positional adaptation are able to provide additional benefits of 4% beyond those achieved by the interval-based schemes. This is made possible by their ability to react quickly to short phases. However, because of the predictor tables required, the hardware overhead is not negligible.

By matching the hardware to the program's parallelism needs, overall processor efficiency can be improved. We see appreciable single thread speed-ups and more than half the clusters are freed up. These clusters can be employed to improve the performance of other threads or they can be turned off to reduce leakage energy consumption on the chip.

4.4.3 Evaluating a Decentralized Cache Model

The earlier subsection has focused on a processor model where the cache is centrally located. The potential bottleneck that this creates can be alleviated by distributing the cache across the clusters. We next study the implications of such an organization on the system's ability to adapt.

Clustered LSQ Implementation. In the decentralized cache model, if an effective address is known when a memory instruction is renamed, then it can be directed to the cluster that caches the corresponding data. However, the effective address is generally not known at rename time, requiring that we predict the bank that this memory operation is going to access. Based on this prediction, the instruction is sent to one of the clusters. Once the effective address is computed, appropriate recovery action has to be taken in the case of a bank misprediction.

If the operation is a load, recovery is simple - the effective address is sent to the correct cluster, where memory conflicts are resolved in the LSQ, data is fetched from the cache bank, and returned to the requesting cluster. If the memory operation is a store, the mis-direction could result in correctness problems. A load in a different cluster could have proceeded while being unaware of the existence of a mis-directed store to the same address. To deal with this problem, we adopt a policy similar to that in [Zyuban and Kogge, 2001]. While renaming, a store whose effective address is unknown is assigned to a particular cluster (where its effective address is computed), but at the same time, a dummy slot is also created in the other clusters. Subsequent loads behind the dummy slot in other clusters are prevented from proceeding because there is an earlier store with an unresolved address that could potentially cause conflicts. Once the effective address is computed, the information is broadcast to all the clusters and the dummy slots in all the LSQs except one are removed. This ensures that no recovery has to be initiated on a store mis-direction. However, this policy does reduce the effective size of the LSQ as some of the entries are replicated until the effective address is computed. The broadcast increases the traffic on the interconnect for register and cache data (which we model).

Bank prediction. Earlier work by Yoaz et al. [Yoaz *et al.*, 1999] had proposed the use of branch-predictor-like tables to predict the bank accessed by a load or store, with the intention of using it to improve instruction scheduling. They had also suggested its use in steering memory operations into different memory pipelines. We employ the use of similar bank predictors to determine which bank the instruction will access. For a 16-cluster system, we need a 4-bit prediction and each of these bits requires an independent predictor. In our simulations, we use a two-level bank predictor with 1024 entries in the first level and 4096 entries in the second. At the first level, bank histories of the predicted banks are maintained for a number of different instructions (indexed by program counter (PC)). The history is xor-ed with some of the PC bits to index into the second level that maintains 2-bit saturating counters that indicate the prediction for the bit.

Steering heuristics. In a processor with a decentralized cache, the steering heuristic has to handle three data dependences for each load or store – the two source operands and the bank that caches the data. Since the transfer of cache data involves two communications (the address and the data), performance is maximized when a load or store is steered to the cluster that is predicted to cache the corresponding data (note that unlike in the centralized cache model, doing so does not increase load imbalance as the cache is not at a single location). Even so, frequent bank mispredictions and the increased traffic from store address broadcasts seriously impact performance. Ignoring these effects improved performance by 29%. At the same time, favoring the dependence from the cache bank results in increased register communication. Assuming free register communication improved performance by 27%. Thus, register and cache traffic contribute equally to the communication bottleneck in such a system.

Disabling clusters. So far, our results have assumed a clustered processor with a centralized

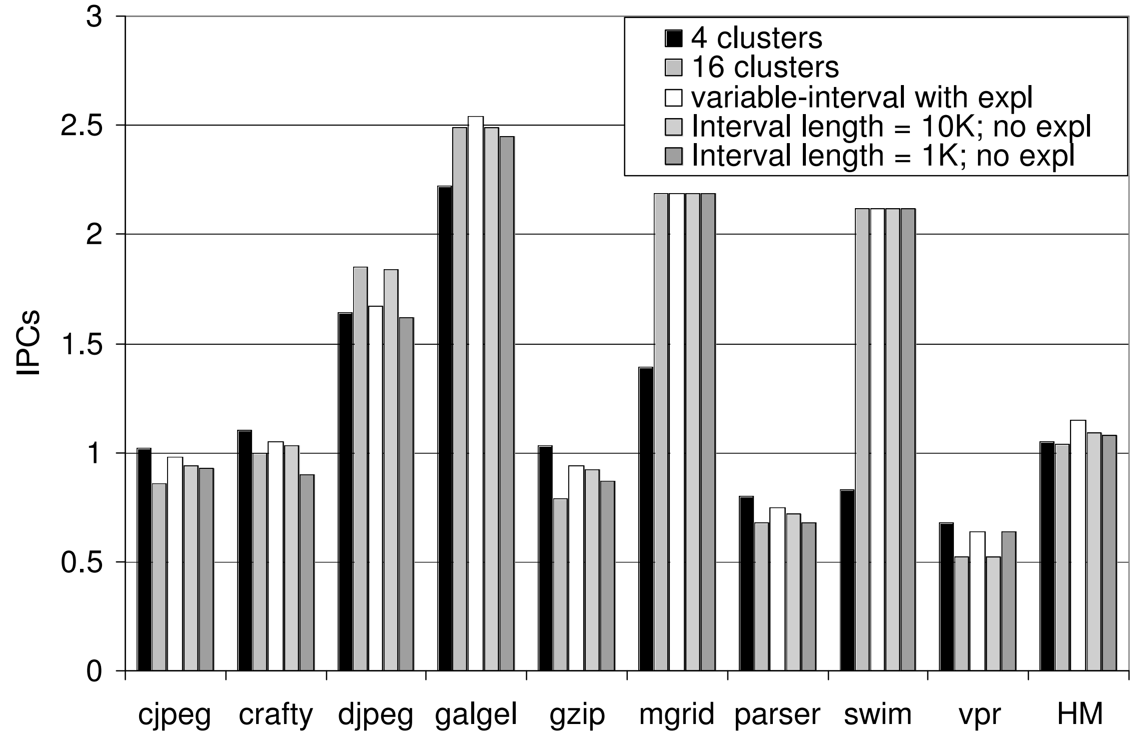


Figure 4.5: IPCs for dynamic interval-based mechanisms for the processor model with the decentralized cache.

cache. Hence, reconfiguration is only a matter of allowing the steering heuristic to dispatch to a subset of the total clusters. With a decentralized cache, each cluster has a cache bank associated with it. Data is allocated to these cache banks in a word-interleaved manner. In going from 16 to four clusters, the number of cache banks and hence, the mapping of data to physical cache lines changes. To fix this problem, the least complex solution is to stall the processor while the L1 data cache is flushed to L2. Alternatively, we could allow the processor to make progress while the flush took place in the background, but then, any cache miss would have to probe the other clusters that could cache that data, in addition to probing the L2. We chose to go with the former implementation because of its simplicity. Fortunately, the bank predictor need not be flushed. With 16 clusters, the bank predictor produces a 4-bit prediction. When four clusters are used, the two lower order bits of the prediction indicate the correct bank.

Results. Because the indexing of data to physical cache locations changes, reconfiguration is not as seamless as in the centralized cache model. Every reconfiguration requires a stall of the processor and a cache flush. Hence, the fine-grained reconfiguration schemes from the earlier subsection do not apply. Figure 4.5 shows IPCs for the base cases and the interval-based mechanisms. The third bar shows the scheme with exploration and a minimum interval length of 10K instructions. The fourth and fifth bars show interval-based schemes with no exploration and the use of distant ILP metrics to pick the best configuration. The simulation parameters for the decentralized cache are summarized in Table 4.2. We find that the results trend is similar to that seen before for the centralized cache model. Except in the case of *djpeg*, there is no benefit from reconfiguring using shorter intervals. Overall, the interval-based scheme with exploration yielded a 10% speedup over the base cases. More importantly, the performance achieved for individual benchmarks is comparable to that for the best static organization.

Since the dynamic scheme attempts to minimize reconfigurations, cache flushes are kept to a minimum. *Vpr* encountered the maximum number of writebacks due to flushes (400K), which resulted in a 1% IPC slowdown. Overall, these flushes resulted in a 0.3% IPC degradation.

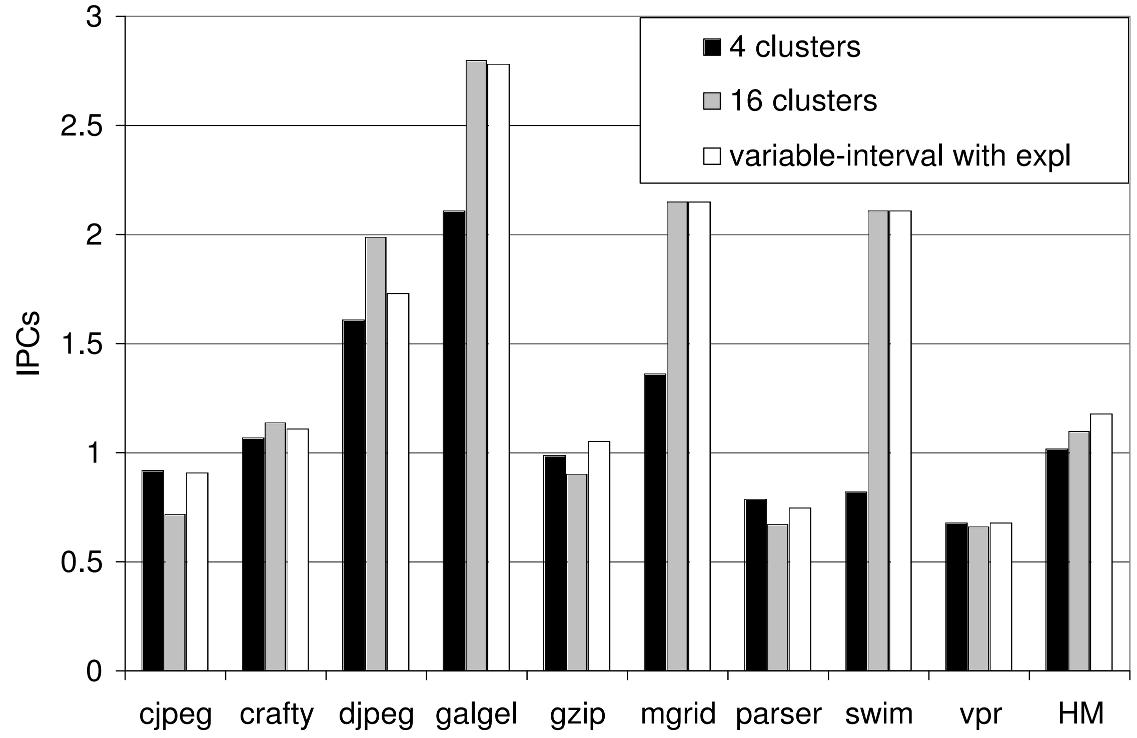


Figure 4.6: IPCs for the dynamic interval and exploration-based mechanism for the processor model with the grid interconnect.

4.4.4 Sensitivity Analysis

Our results have shown that the communication-parallelism trade-off greatly affects the scalability of different programs as the number of clusters is increased for two important cache organizations. In this section, we confirm the applicability of our dynamic reconfiguration algorithms to other meaningful base cases. Some of the key parameters that affect the degree of communication and the degree of distant ILP are the choice of interconnect between the clusters, the latency of communication across a hop, the number of functional units in each cluster, and the number of instructions that can be supported by each cluster (the number of registers and issue queue entries per cluster).

Figure 4.6 shows the effect of using a grid interconnect as described in Section 4.2.3 with a centralized cache model. Because of the better connectivity, the communication is less of a bottleneck and the performance of the 16-cluster organization is 8% better than that of the 4-cluster system. For brevity, we only show results with the interval-based scheme with exploration. The

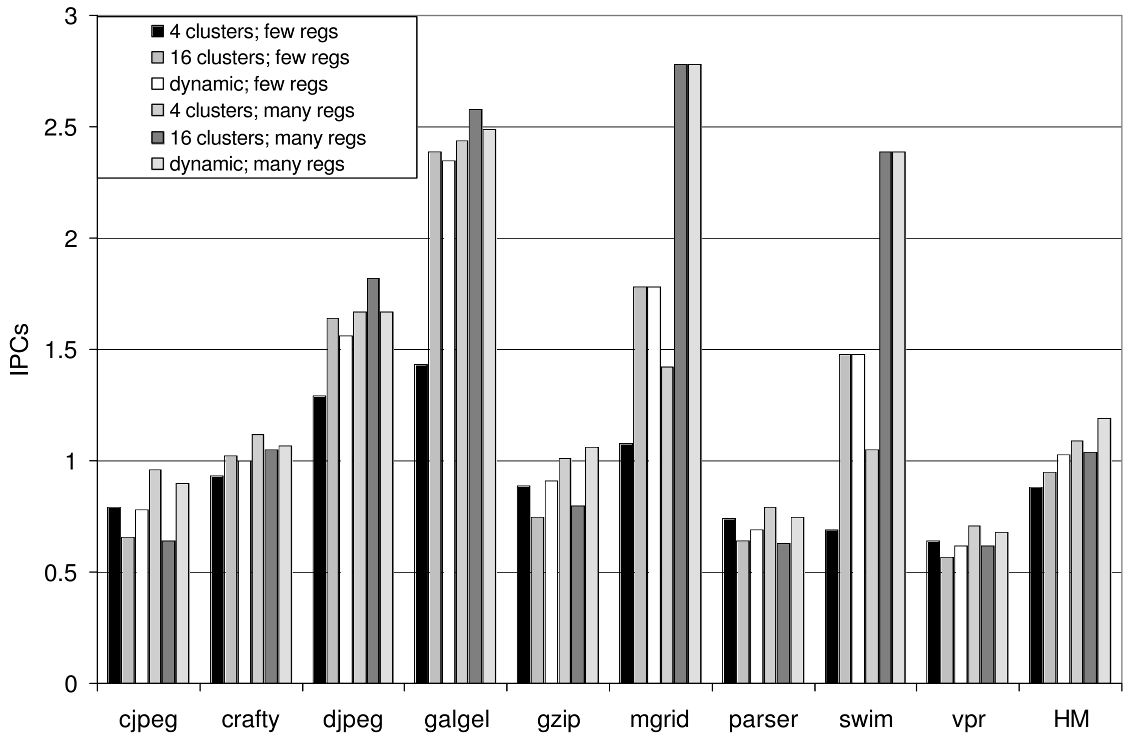


Figure 4.7: Dynamic interval and exploration-based reconfiguration for processor models with different resources. The first three bars represent a processor with a total of 320 registers and 160 issue queue entries (int and fp, each), while the latter three bars represent a processor with 640 registers and 320 issue queue entries.

trend is as seen before, but because the communication penalty is not as pronounced, the overall improvement over the best base case is only 7%. The use of fine-grained reconfiguration techniques yields qualitatively similar results as with the ring interconnect.

Next, we study the sensitivity of the results to the size of various resources within a cluster. Figure 4.7 shows IPCs for a processor model that reduces the number of registers to 20 and the number of issue queue entries to 10 per cluster, and also for a model that increases the number of registers to 40 and the number of issue queue entries to 20. With fewer resources per cluster, the 16 cluster model performs best on an average (since more clusters are needed to exploit the ILP) assuming a fixed configuration, whereas the performance of the 4 and 16 cluster models were nearly identical for the baseline set of resources. Thus, the improvement relative to the 16-cluster model for the dynamic scheme is less (8%). With more resources per

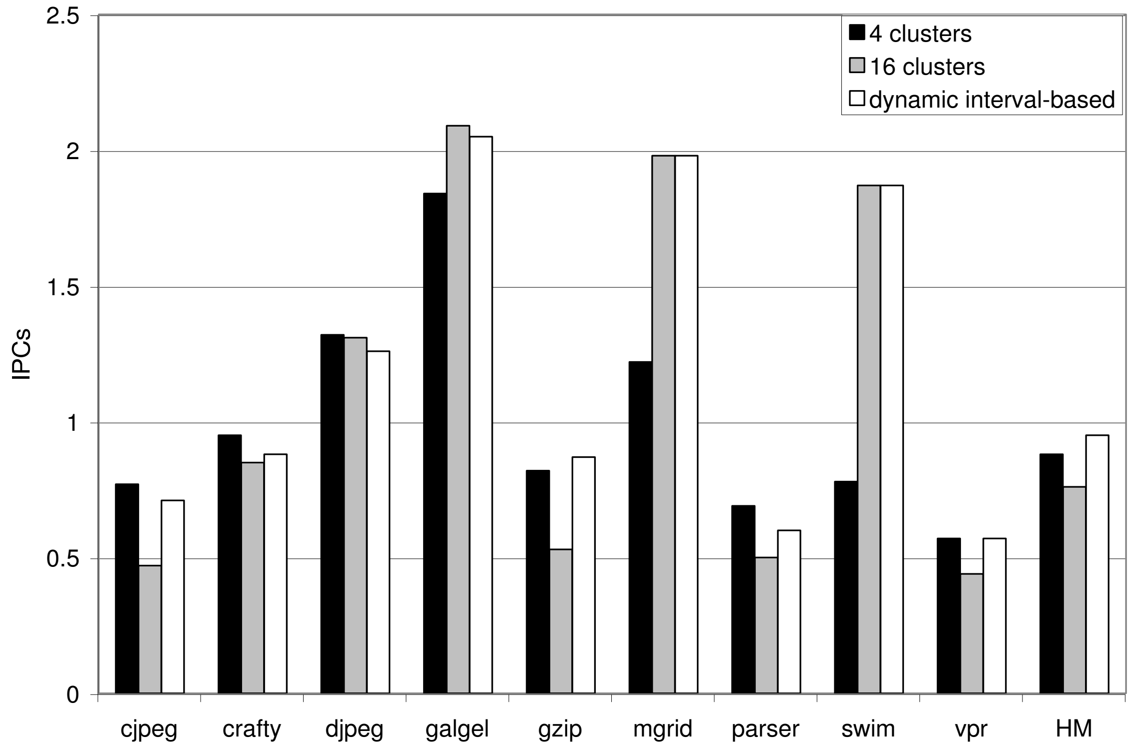


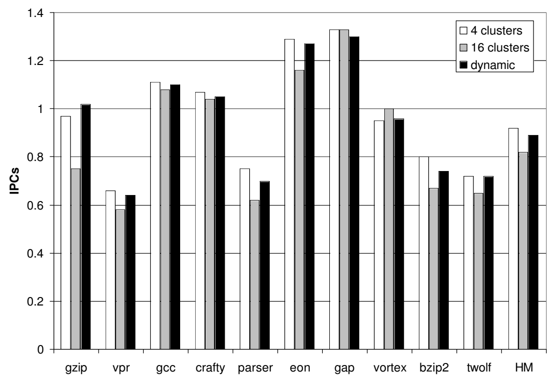
Figure 4.8: Dynamic interval and exploration-based reconfiguration while assuming a latency of two cycles for each hop on the interconnect.

cluster, the 4-cluster fixed configuration outperforms the 16-cluster configuration. The dynamic scheme yields overall improvements of 9% relative to the 4-cluster base and 13% relative to the 16-cluster base.

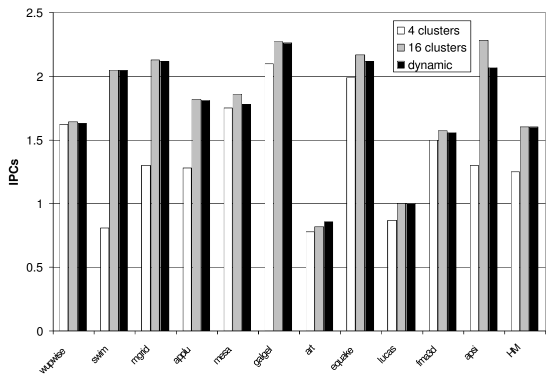
In Figure 4.8, we show results for a model that is highly communication-bound and assumes two cycles for each hop on the interconnect. While the 16-cluster base system performs 14% worse than the 4-cluster system, the 16-cluster system with the interval-based dynamic scheme performs 8% better than the 4-cluster system (and correspondingly, 23% better than the 16-cluster base).

Finally, in Figure 4.9, we show the behavior of the interval and exploration based mechanism on all of SPEC2k[1]. We see that for most SPECInt programs, because of high branch mispredict rates, 4 clusters are enough to exploit most of the available parallelism. The longest

[1]We exclude three programs that did not run with our simulator and two others that had extremely low IPCs and dominated the overall HM numbers. Our algorithms do not adversely affect the performance of these two programs.



(a) SPEC2k Integer results.



(b) SPEC2k FP results.

Figure 4.9: Performance results for all of SPEC2k for the 4 and 16-cluster fixed base cases and the dynamic interval and exploration-based mechanism.

selected interval lengths were observed for *bzip* (2.56M) and *gcc* (640K). For most SPECFP programs, 16 clusters yield optimal performance because of the high parallelism in these codes. The dynamic mechanism is able to detect these differences in behaviors and adapts the hardware so that performance closely matches the best static organization. Over the entire benchmark set, this not only improves processor utilization, freeing up 8.2 clusters on average, but also improves performance by 6% when compared to the overall best base case.

These results are qualitatively similar to the improvements seen with the interval-based schemes in the earlier subsections, indicating that the dynamically tunable design can help improve performance significantly across a wide range of processor parameters and applications. Thus, the communication-parallelism trade-off and its management are likely to be important in most processor settings of the future.

4.5 Related Work

A number of proposals based on clustered processors have emerged over the past decade [Akkary and Driscoll, 1998; Baniasadi and Moshovos, 2000; Canal *et al.*, 2000; Canal *et al.*, 2001; Capitanio *et al.*, 1992; Farkas *et al.*, 1997; Keckler and Dally, 1992; Lowney *et al.*, 1993; Nagarajan *et al.*, 2001; Parcerisa *et al.*, 2002; Ranganathan and Franklin, 1998; Rotenberg *et al.*, 1997; Sohi *et al.*, 1995]. These differ in the kinds of resources that get allocated, the instruction steering heuristics, and the semantics for cross-cluster communication. The cache is a centralized structure in all these models. These studies assume a small number of total clusters with modest communication costs. The Multiflow architecture [Lowney *et al.*, 1993] and the Limited Connectivity VLIW [Capitanio *et al.*, 1992] were among the earlier works that distributed the register file among groups of ALUs. Farkas et al [Farkas *et al.*, 1997] deal with a dynamically scheduled processor and also distribute the issue queue across the clusters. They do a compile-time assignment of instructions to clusters by looking at neighboring static instructions. Canal et al [Canal *et al.*, 2000; Canal *et al.*, 2001] use run-time statistics to do a dynamic steering of instructions to clusters. The best performing heuristic uses information on register dependences and load imbalance. Baniasadi and Moshovos [Baniasadi and Moshovos, 2000] do a similar study for a quad-clustered processor model. Their results show that a simple heuristic that assigns a series of n instructions to a cluster, where the cluster is picked in a round-robin manner, performs as well as other more complicated heuristics that take dependences and past history into account. The Alpha 21264 [Kessler, 1999] is a clustered microarchitecture with two clusters, but replicates the entire register file. Ranganathan and Franklin [Ranganathan and Franklin, 1998] cluster the functional units and the issue queue, but maintain a centralized register file.

Various other architectures like Multiscalar [Sohi *et al.*, 1995], Trace processors [Rotenberg *et al.*, 1997], and DMT [Akkary and Driscoll, 1998] have been proposed that use multiple execution units (clusters). Each cluster is assigned a sequential segment of the program, with all except one of them being speculative in nature. Palacharla [Palacharla *et al.*, 1997] proposes a clustered issue queue, where successively dependent instructions are steered into a single cluster. The head entries of each queue are the only instructions that could be made ready in a cycle, hence the complexity of the wakeup and select logic is greatly reduced.

All these various bodies of work do not attempt to distribute the cache among the various clusters. Recent studies have looked at the problem of increasing cache bandwidth, but have not placed them in the context of clustered microarchitectures. A truly multi-ported cache would be very expensive to implement, prompting the proposal of various alternatives. One such alternative is double-pumping, which runs the cache at a frequency that is an integral multiple of the processor frequency. Since this technique has limited scalability, banked caches have emerged as the acceptable means of providing high bandwidth [Rivers *et al.*, 1997]. Rivers et al [Rivers *et al.*, 1997] discuss the trade-offs of various designs. Replicating the data cache, with each bank having a single read and write port, can accommodate multiple reads in the same cycle, but only a single write can occur in a cycle as it would have to write simultaneously to each bank. In a line interleaved cache, each bank gets a subset of the total cache lines. Multiple accesses can be supported in a cycle so long as they are to different cache lines. This restriction degrades IPC somewhat as spatial locality dictates that successive accesses are often to the same cache line. Word interleaving tries to alleviate this problem by distributing each cache line across the various banks. This implies that, for example, in a dual-banked cache, at most one odd and one even word can be accessed in a cycle. However, each bank would have to replicate the tag storage, unlike in a line interleaved cache. The banked organizations cluster the cache, but leave the rest of the processor, including the load-store queue (LSQ) untouched.

Cho et al [Cho *et al.*, 1999b; Cho *et al.*, 1999a] cluster the cache and the LSQ, but not the rest of the processor. This helps reduce the size of the LSQ and also serves to increase the bandwidth of the cache system without true multi-porting. They use a different criterion for splitting the cache into banks. They notice that 50% of all data accesses are made to the stack/frame and this can often be detected by just looking at the instruction. Only in rare cases can a load be ambiguous in terms of which data region it is accessing. Loads and stores are split into one of two streams early in the pipeline. They make the optimistic assumption that stores in one stream will not conflict with loads in the other stream. As a result, loads in one stream need not compare their address with the addresses of pending stores in the other stream, and this helps reduce the size of the LSQ. One of the cache banks only contains stack and frame data, while the other bank maintains the rest of the data, allowing one access to each bank per cycle. The authors further make the observation that due to the small size of temporary data allocated on the stack, the cache bank servicing the stack accesses can be as small as 4KB. At the time of actually issuing the ld/st, if it is discovered that the instruction was mis-categorized, recovery is initiated. The authors do not address the problem of how such a partition would scale beyond two clusters.

Yoaz et al [Yoaz *et al.*, 1999] anticipate the importance of splitting data accesses across multiple streams early and propose predictors to do the same. They evaluate the accuracy of branch predictor-like schemes to do the prediction and find them to be about 70% accurate for two streams. They propose two uses for such a predictor - (i) improved scheduling of memory operations, and (ii) the use of multiple memory pipelines like that proposed by Cho et al [Cho *et al.*, 1999b; Cho *et al.*, 1999a]. They mention that predictions with low confidence can direct the instruction to all the memory pipelines.

Recently, Zyuban and Kogge [Zyuban and Kogge, 2001] incorporated a clustered cache in their study on the power efficiency of a clustered processor. Our implementation of the decentralized cache closely resembles theirs. A recent study by Aggarwal and Franklin [Aggarwal and Franklin, 2001] explores the performance of various steering heuristics as the number of clusters scale up. Theirs is the only study that looks at as many as 12 clusters and proposes the use of a ring interconnect. They conclude that the best steering heuristic varies depending on the number of clusters and the processor model. To take this into account, each of our clustered organizations was optimized by tuning the various thresholds in our steering heuristic.

Agarwal et al. [Agarwal *et al.*, 2000] show that processors in future generations are likely to suffer from lower IPCs because of the high cost of wire delays. Ours is the first study to focus on a single process technology and examine the effects of adding more resources. The clustered processor model exposes a clear trade-off between communication and parallelism, and it readily lends itself to low-cost reconfiguration.

4.6 Summary

In this chapter, we evaluated the effects of shrinking process technologies and dominating wire delays on the design of future clustered processors. While increasing the number of clusters to take advantage of the increasing chip densities improves the processor's ability to support multiple threads, the performance of a single thread can be adversely affected. This is because performance on such processors is limited by cross-cluster communication costs. These costs can tend to dominate any increased extraction of instruction-level parallelism as the processor is scaled to large numbers of clusters. We have demonstrated that dynamically choosing the number of clusters using an exploration-based approach at regular intervals is effective in optimizing the communication-parallelism trade-off for a single thread. It is applicable to almost every program and yields average performance improvements of 11% over our base architecture. In order to exploit phase changes at a fine grain, additional hardware has to be invested, allowing overall improvements of 15%. Since 8.3 clusters, on average, are disabled by the reconfiguration schemes, there is the potential to save a great deal of leakage energy in single-threaded mode. The throughput of a multi-threaded workload can also be improved by avoiding cross-thread interference by dynamically dedicating a set of clusters to each thread. We have verified the validity of our results for a number of interesting processor models, thus highlighting the importance of the management of the communication-parallelism trade-off in future processors.

5 The Future Thread

In this chapter, we extend the exploitation of the concept of distant and nearby parallelism introduced in Chapter 4. We show how a processor with limited resources could mine distant ILP by using a helper pre-execution thread. The allocation of the limited processor resources between the two threads significantly impacts performance and introduces a trade-off between nearby and distant parallelism. We employ the adaptation algorithms introduced in Chapter 2 to determine the optimal allocation of these resources.

5.1 Managing the In-Flight Window

Dynamic superscalar processors perform register renaming and out-of-order issue in hardware to extract greater instruction-level parallelism (ILP) from existing programs. A significant performance limitation in such processors is the lack of forward progress in the midst of long latency operations (*e.g.*, cache misses). Ideally, these operations should be overlapped with the execution of other independent instructions, especially other performance-degrading long-latency loads or branch mispredicts. However, in order to continue to keep the processor busy, a sufficiently large instruction window would have to be examined to find these independent instructions.

This problem cannot be solved by simply increasing the number of in-flight instructions, as it would require larger register files and reorder buffers that may impact critical timing paths. The register file, in particular, can often determine the cycle time and several approaches that attempt to balance latency and IPC have been proposed. The Alpha 21264 implements a clustered register file [Kessler, 1999] in an attempt to reduce average latency. Similarly, register file caches have also been proposed [Cruz *et al.*, 2000] in order to access a smaller subset of registers in a single cycle. Both of these techniques, however, cause IPC degradation when compared to a single monolithic register file of the same size. A multi-cycle register file has its own problems - design complexity in pipelining a RAM structure, having two levels of bypass (which is one of the critical factors in determining cycle time [Cruz *et al.*, 2000; Palacharla *et al.*, 1997]), and reduced IPC because of longer branch mispredict penalties and increased register lifetimes. These problems are only exacerbated in an SMT processor, where the register file resources have to be shared by multiple threads. Further, as we move to smaller process technologies, the dominating effect of long wire delays will make it even more prohibitive to implement large register files in wide-issue machines [Farkas *et al.*, 1996; Palacharla *et al.*, 1997].

The fundamental reason why the register file size has such a large impact on the size of the instruction window, and hence performance, is that instructions can be renamed and dispatched only when there are free registers available. Registers are freed only when instructions commit, and instructions are committed in order. A single instruction that takes a long time to complete could stall the commit stage, thereby holding up all the registers and not allowing subsequent instructions to dispatch. During this period, the out-of-order execution core can only look at a restricted window of instructions to extract ILP. As the processor-memory speed gap increases, there will be an increasing number of long-latency loads, causing dispatch to frequently stall as it runs out of physical registers. Thus, there is a need for new approaches that allow for forward progress to be made without increasing the complexity of critical hardware structures.

In this chapter, we present a novel architecture that uses the limited number of physical registers to dynamically trade nearby with distant ILP, while still maintaining precise exceptions and program correctness. The front-end can support fetch from two threads, the second of which is dynamically spawned by the hardware rather than being statically created by the program. Initially, the only thread to run is the main (*primary*) program. The secondary (*future*) thread consists only of a program counter and register state. Out of the available rename registers, we dynamically reserve a certain number for the *future* thread, according to the program's current needs to exploit far-flung ILP. Once the *primary* thread runs out of its allocated registers, it stalls, and the *future* thread gets triggered and starts off from where the *primary* left off. This *future* thread cannot alter program state, *i.e.*, it cannot write to memory or update the *primary* thread's registers. It uses the remaining registers to rename and dispatch its instructions.

The *future* thread serves the purpose of potentially warming up the register file, data and instruction caches, and resolving mispredicted branches early. In order to allow the *future* thread to make progress beyond the instructions to which its registers are allocated, we relax the constraints on when these registers are released back into the free list. First, a register is released as soon as all its consumers have read its value, *i.e.*, we make the optimistic assumption that there will be no branch mispredicts or exceptions raised. Second, in order to avoid holding *future* thread resources that prevent other independent instructions from executing, we also add a timeout mechanism to remove instructions that wait for operands in the issue queue for too long. This frees up registers and issue queue slots so that other productive dependence chains can make progress, thereby allowing the *future* thread to get far ahead of the *primary*. When the *primary* thread ceases to be stalled, it dispatches its subsequent instructions, which are potentially already executed by the future thread. However, progress is faster since its loads have been prefetched and its branches have been correctly predicted by the *future* thread. The use of an Instruction Reuse Buffer (IRB) [Sodani and Sohi, 1997] could speed up the execution even more as some of these instructions would not have to be re-executed.

Thus, we rob the main (*primary*) program thread of some of its resources (which are in effect idle) and allocate them to this opportunistic 'helper' (*future*) thread that seeks independent instructions that are more distant. The benefit of such an approach would depend on the nature of the program, and we present a mechanism that dynamically performs this allocation of resources between the *primary* and *future* threads. As a result, in situations where the *future* thread degrades performance, the processor can always revert back to an organization like the base case, where all resources belong to the *primary* thread. Our simulation results indicate that relative to the base simulated architecture, performance is improved overall by 21% with the dynamic helper thread.

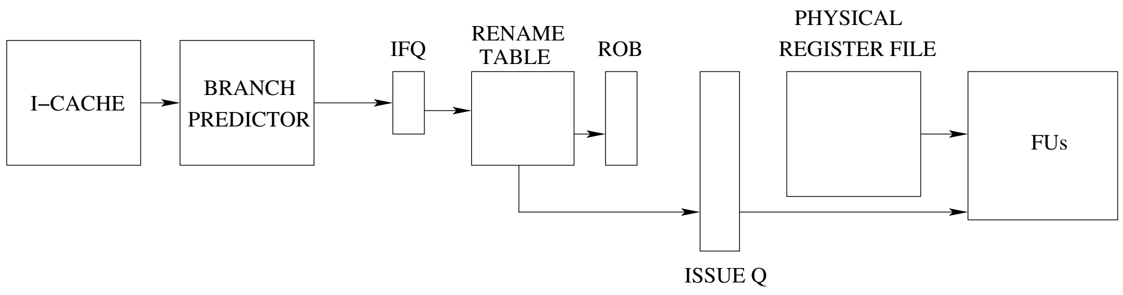


Figure 5.1: The base processor structure

5.2 The Future Thread Microarchitecture

5.2.1 The Base Processor

In a typical processor architecture (Figure 5.1) such as that of the R10000 [Yeager, 1996] and the Alpha 21264 [Kessler, 1999], the processor front-end performs branch prediction, fetches instructions from the instruction or trace cache, and deposits them in the instruction fetch queue (IFQ). The IFQ holds the fetched instructions until they get renamed and dispatched into the issue queue. In the dispatch stage, the logical registers are mapped to the processor's pool of physical registers. The rename table keeps track of logical to physical register mappings and is used to rename instructions before putting them into the issue queue. The destination register is mapped to a new physical register that is picked out of the free list (the list of registers not presently in use). The mapping is also entered into the re-order buffer (ROB), which keeps track of register mappings for all instructions that have been dispatched, but not committed. The issue queue checks for register dependences. As instructions become ready and issue, they free up their issue queue entry. A branch stack within the rename table checkpoints the mappings at every branch so that they can be reinstated in the event of a branch misprediction.

Instructions are issued from the issue queue when their register and memory dependences are satisfied, and they are committed from the ROB in program order as they complete. Consider the following example:

```
Original code          Renamed code
lr7 <- ...             pr15 <- ...
... <- lr7             ...  <- pr15
branch to x            branch to x
lr9 <- lr3             pr31 <- pr19
lr7 <- ...             pr43 <- ...
...                    ...
end                    end
x:                     x:
... <- lr7             ... <- pr15
```

At dispatch, the first write to logical register 7 (lr7) causes it to get mapped to physical register 15 (pr15). This is followed by an instruction that reads lr7. The branch is then predicted to be not taken and the next instructions to be dispatched are a write to lr9 and a write to lr7. At this point, lr7 gets mapped to pr43 and subsequent users of lr7 will now read from pr43. Even if the instruction that reads pr15 has completed, pr15 cannot be released back into the free list unless the write to pr43 has committed. There are two reasons for this: (i) if the write to pr31 raises an exception, to reflect an accurate register file state, lr7 should show the value held in pr15, (ii) if the branch was mispredicted, we would need to jump to x, where the read from lr7 would actually refer to pr15. Hence, pr15 remains live until all instructions prior to the write to pr43 are known to not raise an exception and have all their branches resolved.

In the example shown above, if the write to pr31 was a load instruction that missed in the L2 and had to go to memory, it could occupy the head of the ROB for potentially a hundred cycles. If the processor has 24 rename registers, only up to 23 more instructions that write to registers can be dispatched in this period. This severely limits the ability of the processor to extract ILP.

5.2.2 Overview of the Future Thread

The goal of the proposed architecture is to circumvent the in-order commit process in order to exploit any potential far-flung ILP in addition to nearby ILP. We begin with an overview of the

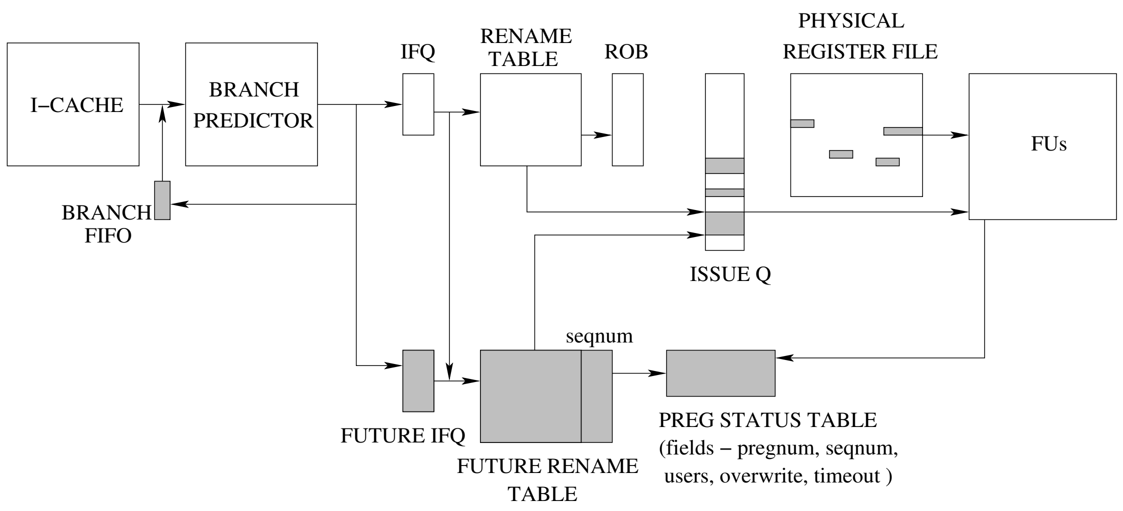


Figure 5.2: The architecture supporting the *future* thread (components belonging to the *future* thread are shaded).

proposed microarchitecture, followed by a more detailed description of the various operations.

As an illustrative example, we begin with a base processor that has 32 int and 32 fp logical registers, and 72 int and 72 fp physical registers (*i.e.*, there are 40 int and 40 fp rename registers). In the *future* thread architecture, the front-end, comprising the IFQ and the register rename table, is replicated (Figure 5.2). While the *primary* thread is not stalled, the *future* thread does not dispatch instructions, but continues to update its rename table to reflect the new mappings in the *primary* thread. Some of the 40 integer rename registers (12, for example), are reserved for the *future* instructions. When the primary thread runs out of registers and stalls, the *future* thread begins to dispatch subsequent instructions using its allocated physical registers. These registers are then freed according to two criteria. Registers are reused as soon as there is no use for them (assuming no mispredicts and exceptions). In addition, if an instruction waits too long in the issue queue, it gets timed out and its register is reused. Instructions waiting in the issue queue for this register are also removed. Application of these two criteria is possible because of the fact that the *primary* thread will re-execute these instructions in order to ensure in-order commit and program correctness. Thus, registers reserved for the *future* thread can be reused much more quickly, potentially allowing the thread to execute far ahead of the *primary*, enabling prefetching of data into the cache, early branch prediction, and value reuse. The *future* thread does not engage in any speculation apart from speculating across branches. It respects register and memory dependences while issuing instructions.

5.2.3 Additional Hardware Structures

The three main additional structures are the *future* IFQ, the *future* rename table, and the Preg Status Table.

There are two program counters, one for the *primary* thread, and one for the *future*. These are identical at first, and fetched instructions are placed in each IFQ. Every cycle, instructions can potentially be renamed by both threads and dispatched into the issue queue. If the same instruction is being handled by both threads, the *future* thread will not dispatch it. The mapping corresponding to that instruction in the *primary* rename table is copied into the *future* rename table.

Each dynamic instruction is assigned a sequence number (this is a counter that wraps around when full and is large enough to ensure that all in-flight sequence numbers are unique — possibly 10 bits long). Sequence numbers are rolled back on a branch mispredict. These sequence numbers make it possible to relate the *primary* instructions to their *future* counterparts.

When the *primary* thread runs out of physical registers, it stalls. The *future* thread continues, using the remaining physical registers to map subsequent instructions. For each instruction that is dispatched by the *future* thread, an entry is added to the Preg Status Table. This is a small CAM structure, the size of the number of registers reserved for the *future* thread (12 entries, in this example, for int and fp each), that keeps track of the current physical registers in use within the *future* thread. The other fields in this structure are: (i) *Seqnum*, the sequence number corresponding to the instruction that has the physical register as destination, (ii) *Users*, indicating how many more consumers of that register still remain in the pipeline, (iii) *Overwrite*, indicating that the corresponding logical register has been remapped by a subsequent instruction, (iv) *Written*, indicating that the result has been written into the physical register, (v) *Timeout*, set to a particular value (30 in our case) at the time of dispatch, and decremented every cycle if the instruction has still not been issued. The *Users* field is incremented every time an instruction is dispatched that sources that physical register. It is correspondingly decremented when that instruction issues. The design considerations of the Preg Status Table are discussed in Section 5.4.

5.2.4 Timeout and Register Reuse

To help the *future* thread use its register resources more efficiently, we eagerly free up registers using the timeout mechanism and the register reuse criteria.

The rationale for the timeout mechanism can be illustrated by Figure 5.3. It shows a histogram of the number of instructions that wait in the issue queue for a given period of time. The histogram is for a 20 million instruction window from the program *perimeter*, and is typical of most memory-intensive programs. It can be seen that instructions are made ready within the first few cycles of their dispatch, or after about 20 cycles, or after 100 cycles. These correspond roughly to the L1, L2, and memory access times. The timeout heuristic models the fact that the non-readiness of an instruction in the first 30 cycles implies that it is waiting on a memory access and is likely to not be woken up for another 70 cycles. Hence, we time it out and allow its register and issue queue entry to be used by other instructions.

Registers get put back into the free list as soon as their overwrite and written bits are set and the number of users becomes zero. Likewise, when the timeout counter becomes zero, the register is put back in the free list, its mappings in the rename table (if still active) and the Preg Status Table are removed, and the instruction is removed from the issue queue. In order to ensure the correct execution of instructions, in the next cycle, the tag of this timed out register is broadcast through the issue queue and all instructions that source it, time themselves out. This not only frees up the issue queue slots but also ensures that the instructions do not wake themselves up when the same register tag (corresponding to the completion of a later instruction) is broadcast as ready. The process is repeated for the newly timed out instructions. *Future* instructions dependent on this value will not be dispatched due to the invalid entry in the rename table. This operation could take a few cycles depending on the length of the dependence chain in the issue queue. To reduce hardware overhead, we could impose the restriction that *future* instructions only occupy certain issue queue slots, thereby having this associative logic

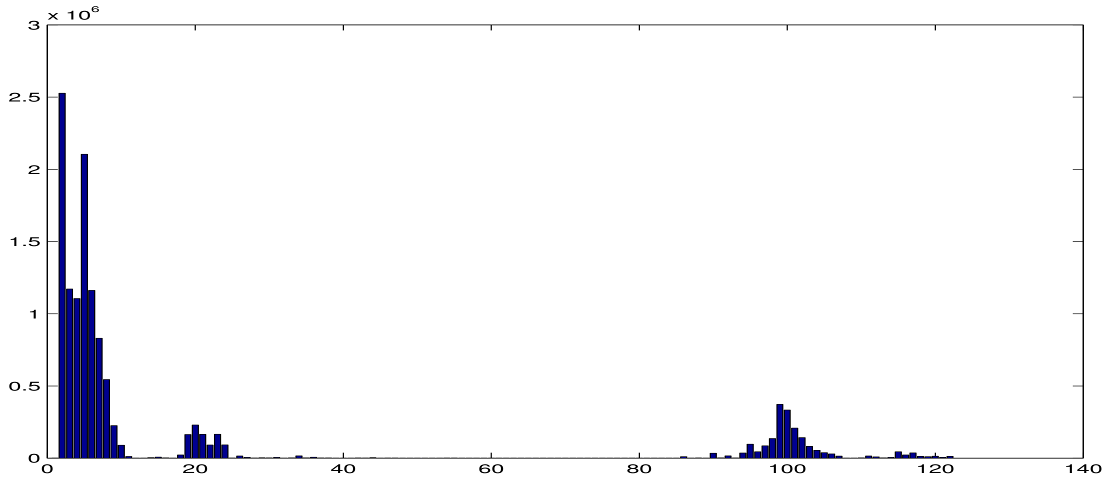


Figure 5.3: Histogram showing waiting time in the issue queue for a portion of the program *perimeter*. The X axis shows the time spent (in cycles) waiting in the issue queue, and the Y axis shows the number of instructions that waited for that period.

for only a subset of the issue queue. While dispatching a *primary* instruction, if the issue queue is full, one of the *future* instructions is explicitly timed out to make room for it. This 'stealing' of issue queue slots ensures that priority is always given to the *primary* instructions.

5.2.5 Redispatching an Instruction in the *Primary*

When the instruction at the head of the ROB completes, the *primary* thread can start making progress again as registers get put in the free list. Instructions are fetched again from the I-cache into the IFQ and then dispatched. While dispatching an instruction, the Preg Status Table and *future* rename table are looked up. The *future* rename table keeps track of the sequence number for the last instruction that mapped the logical register within the *future* thread, while the Preg Status Table includes the sequence number of the instruction writing the physical register. The current instruction's sequence number is used to associatively look up the Preg Status Table. If a physical register mapping still exists for that instruction in the *future* thread, the same physical register is used to map the instruction in the *primary* as well. The corresponding physical register entry is removed from the Preg Status Table, as the register is no longer subject to the rules of the *future* thread. The *future* instructions that source this register need not update their operand tags. Also, the instruction need not be dispatched again into the issue queue, as the earlier dispatch will suffice to produce a result in that physical register. If a result already exists in the physical register, the *future* thread helps speed up the *primary* thread even more. This phenomenon is referred to as *natural reuse*. If a physical register mapping for that instruction does not exist in the Preg Status Table (the register has already been timed out or reused) and if there is a match with the sequence number associated with the *future* rename table's logical register entry, the *future* rename table is updated to reflect the mapping in the *primary* table.

5.2.6 Recovery after a Branch Mispredict

Once triggered, only the *future* thread accesses the branch predictor. It communicates its predictions to the *primary* thread through a FIFO queue. These predictions in the queue are updated when resolved by the *future* thread, so that the *primary* thread need not go along the mispredicted path.

When the *future* thread detects a mispredict, it checkpoints back to the state at the mispredict. However, some values may be lost (as the register might have been reused), thereby disallowing dispatch of instructions along some dependence chains.

As mentioned, the *future* rename table tracks the sequence number corresponding to the logical register mapping. A conventional rename table checkpoints its mapping at every branch. For the *future* thread, the mappings that might have been true at the time of checkpointing need not be true when the checkpoint is reinstated – instructions prior to the branch may have timed out, had their registers reused, or been re-dispatched as part of the *primary* thread. Hence, instead of checkpointing the mapping, we checkpoint the sequence number for the mapping. In addition, the Preg Status Table also checkpoints its overwrite bit. While reinstating the checkpoint, the sequence number is inspected to figure out where the correct mapping can be found. If the sequence number is less than the last sequence number encountered by the *primary* thread, then it means that the *primary* rename table has the correct mapping for that register. If the sequence number is greater, it means that the register, if still valid, should be part of the *future* thread and have a mapping in the Preg Status Table. In the subsequent cycles, these mappings are copied back into the *future* rename table so that it reflects an accurate state, and the overwrite bit is recovered. The Preg Status Table needs to also have an accurate count of the number of active users in the pipeline. As unissued instructions along the mispredicted path are squashed, they decrement the *Users* field for their corresponding source registers, thus ensuring its correctness. In Section 5.4, we suggest a few optimizations that would allow this process to be performed quickly and be overlapped with the time taken to fetch instructions from the correct path.

If the *primary* thread detects a mispredict, the *future* thread starts from scratch after copying the contents of the *primary* rename table.

5.2.7 Exploiting the IRB

In the microarchitecture described thus far, instructions may get executed by both the *primary* and *future* threads. An instruction reuse buffer (IRB) could be used to minimize this redundancy[1]. An implementation scheme like S_n or S_{n+d} [Sodani and Sohi, 1997] could be easily used with minimal modification. In our simulations, we use the S_n scheme because of its simplicity. In this scheme, the reuse buffer keeps track of the program counter, the operand names (register addresses) for an instruction, and the result value it produced when it was last invoked. During dispatch, if a program counter match is found in the IRB and the result value is valid, an instruction can bypass the issue and execute stages of the pipeline. Each instruction creates an entry in the IRB at the time of dispatch, and updates the result value at the time of completion. When an instruction dispatches, it also invalidates all the entries in the IRB that source the same logical register as its destination. Similarly, a store invalidates all loads in the IRB that have the same source address.

To support the *future* thread, two modifications need to be made to the IRB. *Primary* instructions cannot create IRB entries once the *future* thread is triggered (these entries may be invalid because the *future* thread may have dispatched instructions that have modified the operands, which the *primary* has no way of knowing). In addition, the entries in the IRB also keep track

[1]An IRB in a conventional microarchitecture exploits value locality by not re-executing instructions if they have the same operand values.

of the sequence number for the *future* instruction that produced them. The *primary* thread can reuse valid results in the IRB as long as these results were produced by instructions with sequence numbers smaller than or equal to that of the instruction being dispatched. This ensures that the contents of the logical registers that are the operands is the same as that used to generate the result.

5.2.8 Dynamic Partitioning of Registers

The allocation of physical registers between the *primary* and *future* threads need not be set at design time. In fact, a number of programs that do not have distant ILP would be better off using their registers to exploit nearby ILP rather than have the *future* thread throw those results away to advance further. We include a mechanism that dynamically accomplishes this partitioning on the fly. The number of registers allocated to each thread is controlled by stalling the thread's dispatch as soon as it has consumed its allotted registers. A simple counter keeps track of the registers allotted to and freed by each thread. A register that can be dynamically set specifies the maximum allowed counter value.

We employ two of the adaptation algorithms described in Chapter 2 to determine the optimal allocation of registers at run-time. The first is the interval-based mechanism with exploration. If the processor has forty rename registers, during the exploration process, nine different candidate organizations are profiled, with four to forty rename registers being allocated to the *primary* thread (in steps of four). The best performing organization is then used till the next phase change is detected. Most of the programs in this study exhibit fairly low instability factors and we use a fixed 100K instruction interval for all the programs. The second mechanism employs positional adaptation also with an exploration process.

Chapter 2 also describes mechanisms that rely on hardware metrics to predict the optimal organization instead of implementing an exploration process. However, in the context of the *future* thread, no single metric can estimate the benefit of the pre-execution thread. Even if the program has a high degree of distant ILP, pre-executing the distant instructions may not impact the performance of the *primary* thread if there is no prefetching effect, if no branch mispredicts are resolved, or if register values are not re-used. The use of the pre-execution thread also

Fetch queue size	16

Branch predictor	comb. of bimodal and 2-level gshare;
	bimodal size 2048;
	Level1 1024 entries, history 10;
	Level2 4096 entries (global);
	Combining predictor size 1024;
	RAS size 32; BTB 2048 sets, 2-way

Branch mispredict penalty	9 cycles
Fetch, dispatch, issue, and commit width	4
Issue queue size	20 (int), 15 (fp)
L1 I and D-cache	64KB 2-way, 2 cycles, 32-byte line sizes
L2 unified cache	1.5MB 6-way, 15 cycles, 64-byte line size
TLB	128 entries, 8KB page size
Memory latency	70 cycles for the first chunk
Memory ports	2 (interleaved)
Integer ALUs/mult-div; FP ALUs/mult-div	4/2; 2/1

Table 5.1: Simplescalar simulator parameters

reduces the window seen by the *primary* thread and its negative impact is hard to estimate. Hence, we focus our results only on the exploration-based algorithms.

5.3 Results

5.3.1 Methodology

Like in the earlier chapters, we used Simplescalar-3.0 [Burger and Austin, 1997] for the Alpha AXP instruction set to simulate a dynamically scheduled 4-wide superscalar processor. The simulation parameters are summarized in Table 5.1 and are again similar to those seen in previous chapters.

The simulator has been modified to model the memory hierarchy in great detail (including

Benchmark	Input dataset	Simulation window	Base case IPC
em3d (Olden)	20000 nodes, arity 20	500M-525M instructions	0.51
mst (Olden)	256 nodes	9M-14M instructions	0.44
perimeter (Olden)	32Kx32K	1515-1540M instrs	0.39
art (SPEC2k)	ref	500M-550M instrs	0.96
swim (SPEC2k)	ref	1000M-1025M instrs	0.73
lucas (SPEC2k)	ref	2000M-2050M instrs	1.03
sp (NAS)	A, uniprocessor	2500M-2550M	0.98
bt (NAS)	A, uniprocessor	3200M-3250M	0.71
go (SPEC95)	ref	1000M-1025M	1.29
compress (SPEC95)	ref	2000M-2025M	1.53

Table 5.2: Benchmark description

interleaved access, bus and port contention, writeback buffers). We also model a physical register file and an issue queue that is smaller than the ROB size. In Simplescalar, the issue queues and the ROB constitute one single unified structure called the Register Update Unit (RUU). These are further divided into separate integer and floating-point structures.

Our base processor has parameters resembling the Alpha 21264 [Kessler, 1999]. We use 72 integer[2] (int) and 72 floating-point (fp) physical registers (corresponding to 40 rename registers, int and fp, each) and integer and fp issue queues of 20 and 15 entries, respectively. We use a sufficiently large ROB as it is a relatively simple structure and is likely to not be on the critical path. Dispatch gets stalled as soon as either the registers or the issue queue entries get used up, so the ROB occupancy rarely exceeds 80 entries, which is the ROB size in the 21264. Our goal is to demonstrate potential improvements on an existing processor model. In addition, we present results with and without a small 16-entry fully-associative IRB with the S_n implementation scheme.

We ran our simulations on 10 programs from SPEC2000, SPEC95, the NAS Parallel Benchmark [Bailey *et al.*, 1991], and the Olden suite [Rogers *et al.*, 1995]. Eight of these programs

[2]The Alpha has 80 integer registers. We use 72 for uniformity.

are memory-intensive and suffer the most from the problem of a single long latency instruction holding up the commit stage. We have also included two non-memory-intensive programs (*go, compress*) from SPEC95 INT, to illustrate the effect of the *future* thread on this class of applications. To reduce simulation time, we studied cache miss rate traces to identify program warm-up phases and smaller instruction windows that were representative of the program behavior[3]. The programs were also run for 1M instructions in detail to warm up the various structures before measuring performance. Details on the benchmarks are listed in Table 5.2. The programs were compiled with Compaq's cc, f77, and f90 compilers for the Alpha 21164 with full optimizations.

5.3.2 Analysis

We start by examining the effect of the allocation of registers between the two threads. This demonstrates the trade-off between nearby and distant ILP and we then show the efficiency of our adaptation algorithms in handling this trade-off. Finally, we identify the contributions of the different features of the *future* thread to the performance improvement and study the effect of various parameters like the IRB, issue queue, and register file size. We also evaluate future processor models with projected parameters and study the combination of the *future* thread with a hardware stride prefetcher.

Dynamic partitioning of registers

Figure 5.4 shows speedups with the *future* thread for various fixed allocations of registers between the *primary* and *future* threads. For all figures, the IPCs have been normalized with respect to an identical base case that has no *future* thread (*i.e.*, all rename registers are allocated to the *primary* thread). Of these various static organizations, the 28::12 allocation that reserves 28 registers for the *primary* thread has the best overall speedup (when comparing the harmonic mean (HM) of IPCs). However, we see that different allocations do well for different programs. This depends on whether the program has distant or nearby ILP and whether the number of registers reserved for the *future* thread are enough to allow it to advance far enough to exploit

[3]Because each iteration in *bt* is very long, we used a smaller window than was representative of the whole program. However, the results were selectively verified to be indicative of the performance over longer windows.

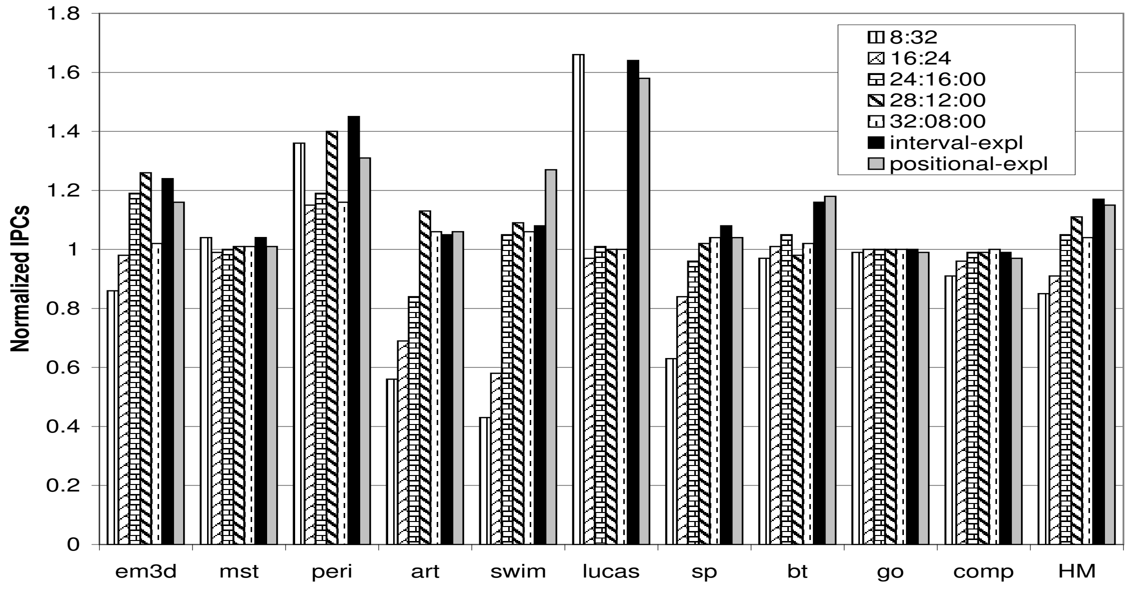


Figure 5.4: Performance of the *future* thread for various fixed register allocations between the *primary* and *future* thread. For example, '8::32' represents an allocation where 8 rename registers are reserved for the *primary* thread and the remaining 32 are reserved for the *future*. The second last bar shows performance with the interval and exploration based scheme that dynamically picks the best allocation. The last bar represents the performance with positional adaptation and exploration. IPCs have been normalized with respect to a base case that has no *future* thread and uses all 40 rename registers for the *primary*.

	em3d	mst	peri	art	swim	lucas	sp	bt	go	comp
Num timeouts	0.29	1.12	0.56	0.31	0.42	0.59	0.37	0.16	0.00	0.03
Num eager reg release	0.45	0.03	0.65	0.30	0.11	0.06	0.13	0.28	0.01	0.06
Num natural reuse	0.14	0.13	0.20	0.23	0.37	0.25	0.22	0.26	0.10	0.16
Avg dist between oldest and youngest instrs (base, *future*)	71,136	25,115	51,114	63,131	67,123	31,183	75,128	47,75	19,19	39,49
Num loads issued by *primary* thread that take more than 40 cycles (base, *future*)	0.12, 0.05	0.02, 0.02	0.11, 0.05	0.02, 0.01	0.04, 0.04	0.05, 0	0.03, 0.02	0.05, 0.04	0, 0	0, 0
Num *future* instrs issued	0.7	0.2	1.4	0.8	0.8	0.6	0.6	0.9	0.2	0.4
Branch prediction rate (rounded off)	95%	97%	94%	98%	99%	98%	89%	98%	80%	93%
% of mispreds detected by *future* instrs	88%	0%	59%	42%	74%	99%	73%	68%	4%	3%
IRB hit rate for *primary* thread	20%	5%	10%	35%	8%	0%	5%	14%	22%	16%

Table 5.3: Various statistics pertaining to the *future* thread (with a dynamic interval-based allocation of registers) and the base case with no *future* thread (most numbers are normalized to the number of committed instructions, for example, Num timeouts is the number of timeouts per committed instruction).

Benchmark	Number of phase changes	Most commonly selected allocations
em3d	1	28:12
mst	11	24:16
perimeter	1	16:24, 28:12
art	17	28:12
swim	0	28:12
lucas	0	8:32
sp	9	28:12, 36:4
bt	4	20:20, 36:4
go	27	40:0, 32:8
compress	11	36:4, 32:8

Table 5.4: Number of phase changes encountered for each program.

this distant ILP. The highest speedups for *lucas* and *mst* are seen by reserving only eight registers for the *primary* thread, but this is the worst allocation for a number of programs that also have nearby ILP. This motivates the need for a dynamic scheme that picks the right allocation on the fly, depending on program requirements. The second last bar in Figure 5.4 shows that the overall speedup of 1.17 with the interval-based dynamic scheme far exceeds the speedup of 1.11 possible with the best static organization. The most impressive speedups are seen for *em3d*, *perimeter*, *lucas*, and *bt*, all of which have a high degree of distant ILP. We see almost no improvements for non-memory-intensive programs like *go* and *compress*[4] as they rarely run out of registers, thereby not triggering the *future* thread. Table 5.4 details the number of phase changes seen for each program in the benchmark set. The last bar shows performance results when attempting positional adaptation with exploration. We find that significant speedups can be had in many cases, but this is often less than that possible with the interval-based mechanism. The in-flight windows considered by the *future* thread are on the order of a few hundred instructions. Hence, attempting reconfiguration at finer granularities can lead to noisy measurements

[4]*Compress* has a high L1 miss rate, but a small L2 miss rate, and the instruction window in the base processor is large enough to hide L2 latencies.

and incorrect decisions. We observed best performance while attempting reconfigurations at every 100th branch and while recording five samples for each event. Because of this, the initial overhead of computing the predictions is fairly high. *Swim* was the only program where positional adaptation outperformed the interval-based scheme. This is caused by the unpredictable performance observed because of lost register values following a branch mispredict. This phenomenon is further discussed in the next sections.

All subsequent discussions and results assume the use of the dynamic interval-based allocation of registers between the *primary* and *future* threads.

Effects of Prefetch, Natural Reuse, Branch Resolution, and Instruction Reuse

Table 5.3 shows various statistics that help us explain the behavior of the *future* thread. Figure 5.5 helps isolate the contributions of the various components to the performance of the *future* thread. In Figure 5.5, the first bar (only prefetch) shows a *future* thread implementation that runs ahead along predicted paths to warm up the data and instruction caches, while ignoring the outcome of all branch instructions. In this scenario, branch mispredicts are discovered only when the *primary* thread re-executes the branch instruction. Register values produced by the *future* thread are thrown away and never used by the *primary* thread. There is no IRB in the processor. The second bar includes the effect of *natural reuse*. Register values produced by the *future* thread can be integrated into the *primary* thread if the *future* thread has not recycled that register. The third bar shows an implementation where the *future* thread also resolves branch mispredicts early and initiates recovery. The fourth bar represents a model that adds an IRB.

We see that the prefetch effect results in big improvements in some of the programs (*perimeter, lucas, bt*), but because this prefetch effect comes at the expense of throwing away nearby results, noticeable slowdowns are seen for many of the programs. As a result, the overall benefit from prefetching is reduced to a marginal 3%. The use of *natural reuse* helps alleviate this problem to some extent. A smaller number of instructions are now executed twice and the *primary* thread does not suffer as much from the use of a smaller window. The overall performance improvement from prefetch and *natural reuse* is 12%. *Natural reuse* serves as a kind of prefetch for register values. The advantage of initiating long latency ALU operations well before the

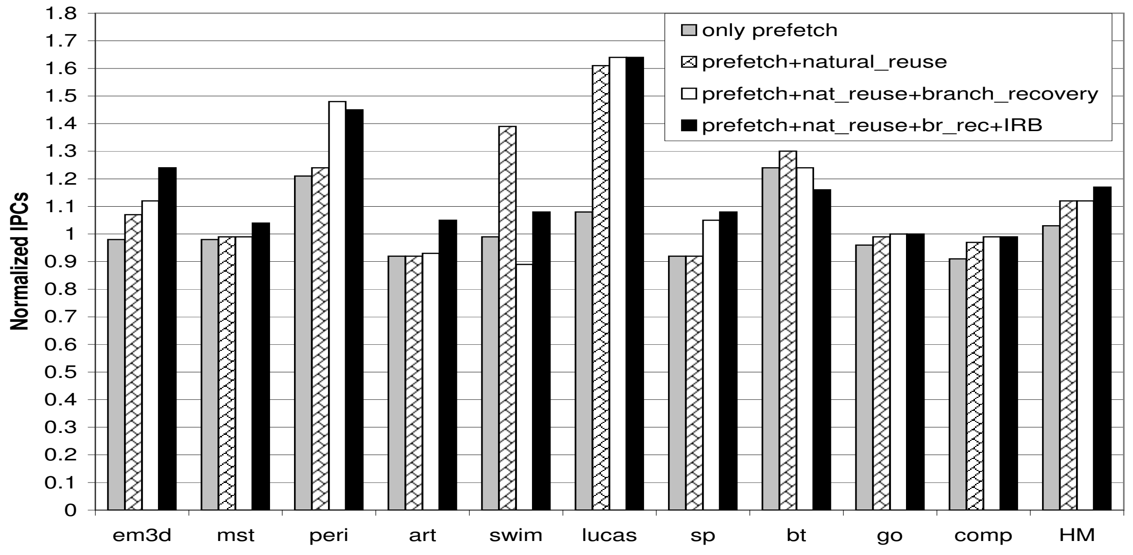


Figure 5.5: *Future* thread performance broken down as prefetch, natural reuse, early branch recovery, and instruction reuse.

primary thread encounters them is quite significant.

Table 5.3 shows that there is a sharp drop in the number of long latency loads seen by the *primary* thread. The number of loads per committed instruction that see a latency of more than 40 cycles falls by almost a factor of two and is even reduced to zero in the case of *lucas*. For *lucas*, the dynamic scheme allocates most rename registers to the *future* thread and this enables it to advance as far as the next loop iteration, thereby fetching the data and initiating the operations long before the *primary* thread starts that iteration.

When the *future* thread is allowed to initiate early branch recovery, we see significant improvements for the programs with low branch prediction accuracies. This results in an additional improvement of 5%, 24%, and 13% in *em3d, perimeter*, and *sp*, respectively. On the other hand, we see a big drop in performance for *swim*. When the *future* thread initiates early branch recovery, it tries to restore a valid register state. Because of the eager release of registers, some values remain lost, disallowing progress along those dependence chains. This sets off a chain reaction, where the *future* thread runs much further ahead but is unable to execute any of the instructions. It can be productive again only when the *primary* thread catches up, which occurs

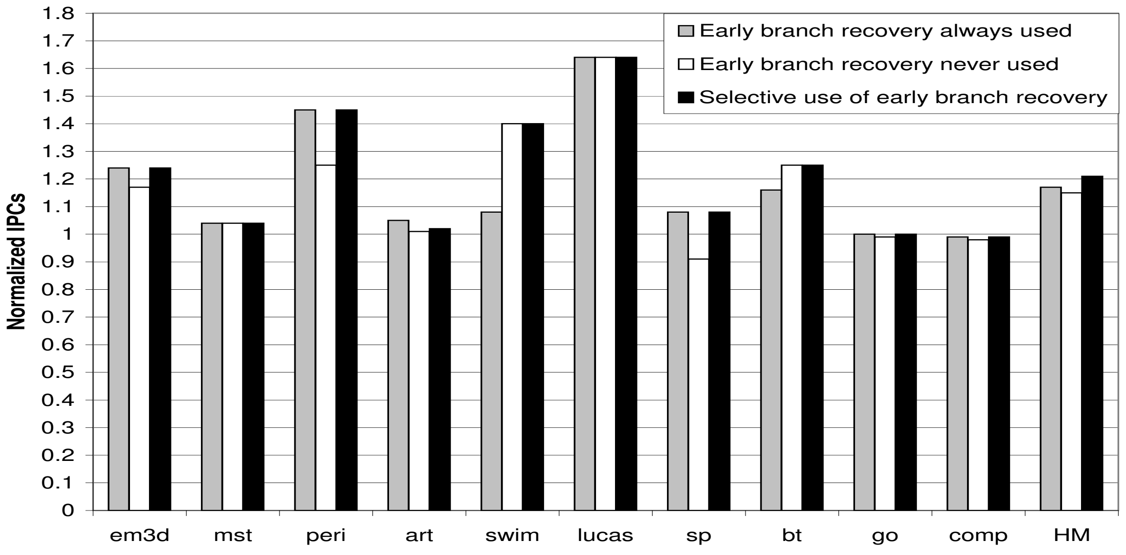


Figure 5.6: Selective use of early branch recovery.

when the *primary* discovers a branch mispredict (for a branch not executed by the *future*) and squashes all subsequent instructions. *Swim* is a loop-based floating-point code and has a low branch mispredict rate. As a result, the *future* thread may have to wait a very long time before it has valid register mappings. This effect is also somewhat seen for *bt*. This negative effect of early branch recovery can be easily eliminated by not attempting it for programs with high branch prediction accuracies.

Finally, by adding the IRB we see an additional overall improvement of 5%. A number of instructions that have been dispatched by the *future* thread need not be re-executed when seen by the *primary* thread. The last row in Table 5.3 shows that up to 35% of these instructions can obtain their result from the IRB. This IRB hit rate improves slightly when we use larger IRBs. Using a 128-entry IRB, we see additional improvements of 8% and 7% in *mst* and *bt*, resulting in an additional 1% overall improvement.

The above breakdown indicates that prefetch and *natural reuse* account for most of the benefit, with the IRB also contributing to some extent. The early recovery from branch mispredicts did not result in any overall speedup because of its negative effect on programs with high branch prediction accuracies. To exploit this feature, we also incorporated the use of selective

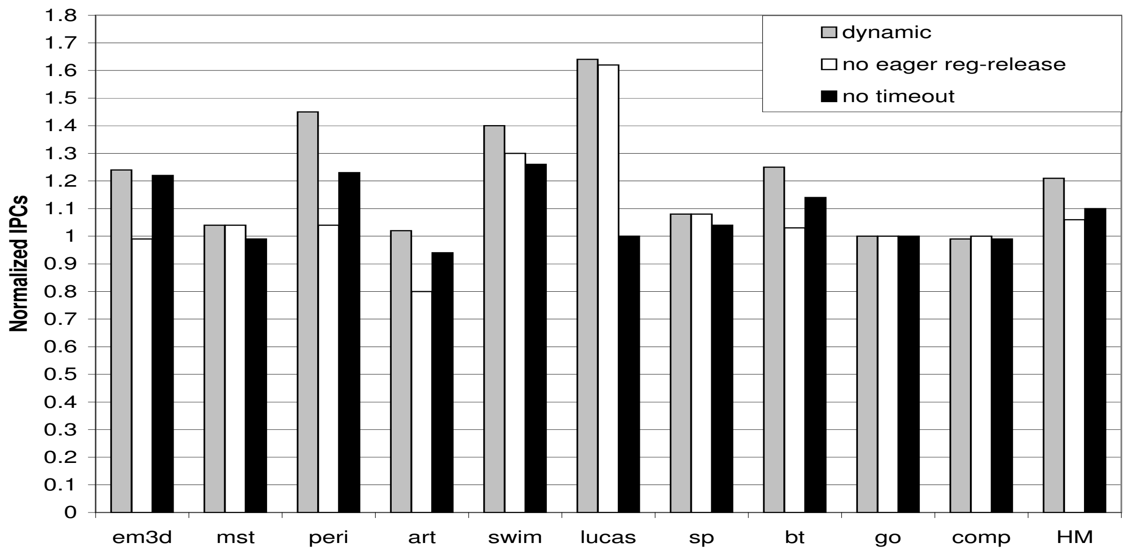


Figure 5.7: The contributions of the various features of the *future* thread. The leftmost bar represents the dynamic scheme with all features turned on. The next two bars show the effect of not using the eager release of registers and the effect of not using the timeout mechanism.

early branch recovery. If the branch prediction accuracy exceeded 98% (*art, swim, lucas, bt*), we did not employ early recovery. In Figure 5.6, we show the effect when early branch recovery is always used and when it is never used. We then show the effect of selectively using one of these two options based on the program. As a result, the overall speedup increases from 1.17 to 1.21. The maximum speedup observed was 1.64 and the maximum slowdown was 0.99.

All our subsequent results assume the use of selective early branch recovery.

Breakdown of Contributions

Two major design components enable the *future* thread to advance ahead of the *primary*. From Table 5.3, it can be seen that the average distance between the oldest and youngest instruction within the processor increases greatly because of the *future* thread. This number represents the size of the in-flight instruction window. The largest window seen by the base architecture is only 75 instructions (in the case of *sp*), but the *future* thread can look in a much larger window (as large as 183 in the case of *lucas*) because of the eager release of registers and the timeout

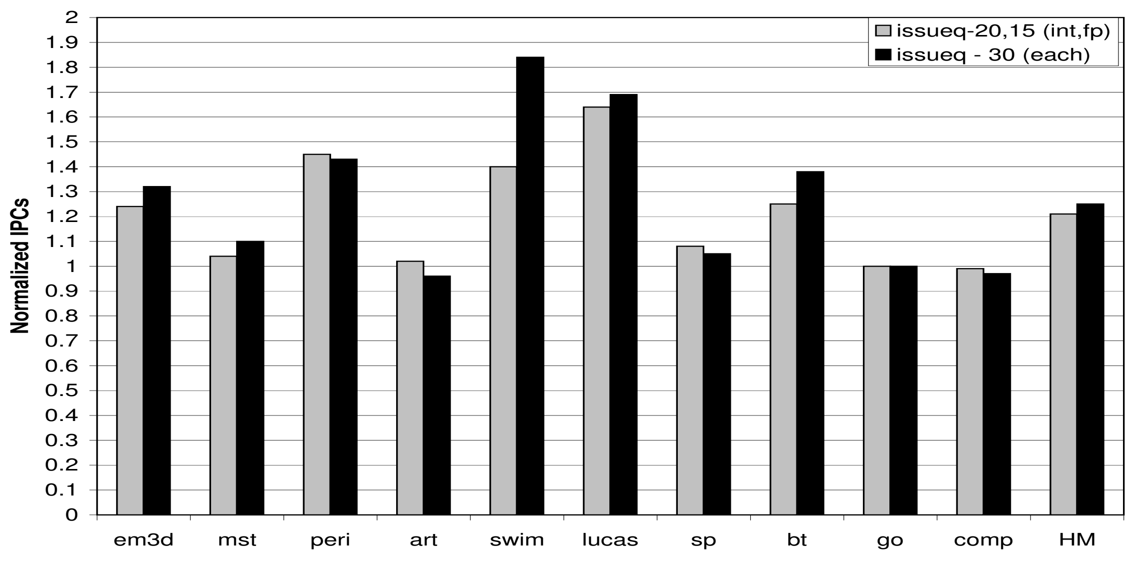


Figure 5.8: The effect of a larger issue queue. The left bar shows speedups with the *future* thread for the Alpha-like processor, while the right bar shows speedups for a processor model that has the same parameters except for a larger issue queue.

mechanism. Both of these often come into play as evidenced by the statistics in the first two rows of Table 5.3.

Figure 5.7 quantifies the contributions of each of these components by disabling them one at a time. It can be seen that eager register release accounts for most of the speedup in *em3d*, *perimeter*, and *bt*, while timeout helps greatly in *perimeter*, *swim*, and *lucas*. For *lucas*, the primary bottleneck is the issue queue. The use of the timeout mechanism not only helps free up registers, it also reduces contention for the issue queue, thereby not stalling dispatch. This allows the *future* thread to advance far enough to do an effective job prefetching. Eliminating either feature results in overall speedups of only 1.06 and 1.10. Note that including both features results in more than additive speedups. As we examine a larger in-flight window, the opportunities for timeout and register reuse also increase, allowing the combination of the two to be synergistic.

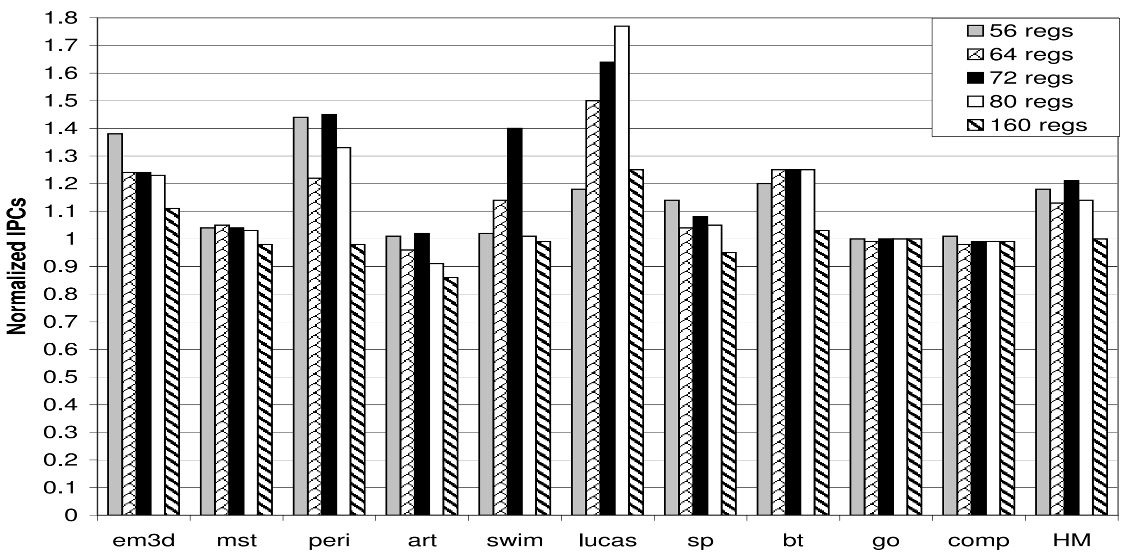


Figure 5.9: Speedups with the *future* thread for processor models that have different register file sizes.

Effect of various processor parameters

Mst is a memory-intensive program that does not show much improvement as it has little nearby ILP, causing instructions to wait in the issue queue, thus stalling dispatch. For the other programs, by using the *future* thread, the register file is removed as the bottleneck to dispatch. Hence, stalls are usually caused by the small size of the issue queue. We next evaluate the *future* thread for a processor model that has larger int and fp issue queues of 30 entries each. The larger issue queues resulted in very little improvements for the base case, but they enabled the *future* thread to advance even further, resulting in an overall speedup of 1.25 (see Figure 5.8). The greatest improvement was seen for *swim*, where the size of the in-flight instruction window with the *future* thread went from 119 for the smaller issue queues to 169 for the larger queues.

We next study the effect of different register file sizes. Figure 5.9 shows speedups with the *future* thread for processor models that have physical register file sizes ranging from 56 to 80 registers (int and fp, each). We also show a processor model that is much more aggressive, having 160 registers and 64 issue queue entries (int and fp, each). Each bar uses the corresponding base case to compute speedups. Two effects come into play here. Using a smaller register file makes it more of a bottleneck, increasing the potential benefit of the *future* thread. However, with a smaller register file, the *future* thread will also be limited in its ability to look ahead, reducing the prefetch effect. Depending on which effect dominates, we see different behaviors for the different programs. Hence, a clear trend is not seen in the overall speedup numbers. It must be pointed out that the raw IPC for a 56-register base case augmented with the *future* thread (0.72 IPC) is better than the raw IPC for a 72-register base case without the *future* thread (0.71 IPC). While the IPCs are comparable, the former processor model can support a faster clock speed if the register file is on the cycle time critical path. However, if the processor can support large register files and issue queues (as shown by the last bar), the benefit of the *future* thread is very marginal – only *em3d* and *lucas* show appreciable speedups. Hence, the future thread is only useful in a scenario where implementation constraints limit the sizes of the register file and issue queue, thereby rendering them as bottlenecks.

The processor model that has been described and simulated assumes that both the *primary* and *future* threads can rename and dispatch up to four instructions each in every cycle. This would require as many as eight write ports in the issue queue. Further, we have assumed that additional paths exist in the issue queue so that tags of timed out instructions can be broadcast. We also simulated a processor model that takes into account these constraints. Only up to a total of four instructions from either the *primary* or *future* thread were allowed to dispatch. Only up to a total of four instructions were allowed to either issue or timeout every cycle. These two constraints ensure that we do not introduce additional complexity, as compared to the base case. Only *swim* and *sp* showed modest performance loss because of this, but they still outperformed the base case.

Finally, we evaluate two other projected future processor models. We use the approaches outlined by Agarwal et al [Agarwal *et al.*, 2000] to define a pipeline-scaled model that keeps structure sizes the same but uses deeper pipelines and a capacity-scaled model that shrinks structure sizes so as to not have deep pipelines. The pipeline-scaled model increases the L1, L2, memory, and mispredict latencies to 3, 20, 105, and 15 cycles, respectively. The capacity-scaled model uses 64 physical registers, issue queue sizes of 15 (int) and 12 (fp), 32KB 2-cycle L1s, 1MB 15-cycle L2, memory latency of 105 cycles, and a mispredict penalty of 10 cycles. If

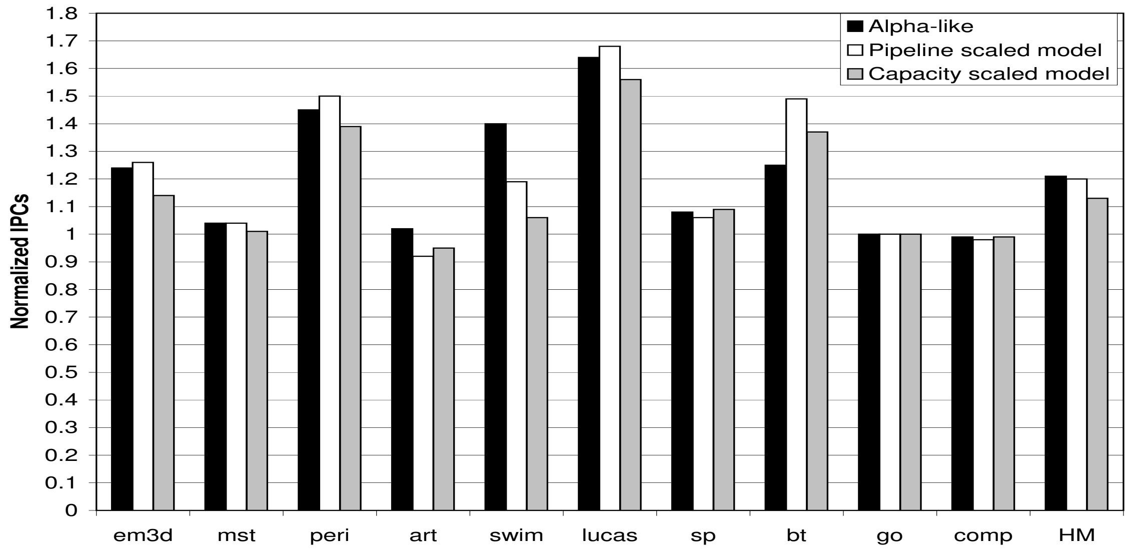


Figure 5.10: Speedups for two future processor models.

the *future* thread is able to hide the longer memory access times completely, its relative impact from the prefetch effect is also greater. Figure 5.10 shows that the speedups go up slightly for most programs with the pipeline-scaled model. No overall benefit is seen because there is a performance degradation for *swim*. For the alpha-like model, the use of the *future* thread causes the percentage of long latency loads (more than 40 cycles) in *swim* to go down from 19% to 2%, while causing the average load latency to go from 22 to 6. With the longer load latencies in the pipeline-scaled model, however, the *future* thread is unable to completely hide the memory latency. While the *future* thread does a similar prefetch job, reducing the average load latency from 31 to 19, the percentage of long latency loads is reduced only marginally from 20% to 18%. As a result, the relative impact of the *future* thread is a lot less for the pipeline-scaled model in the case of *swim*.

The speedups for the capacity-scaled model are less impressive. This is partly because the issue queue becomes a greater bottleneck and partly because the smaller register file does not allow the *future* thread to run much further ahead.

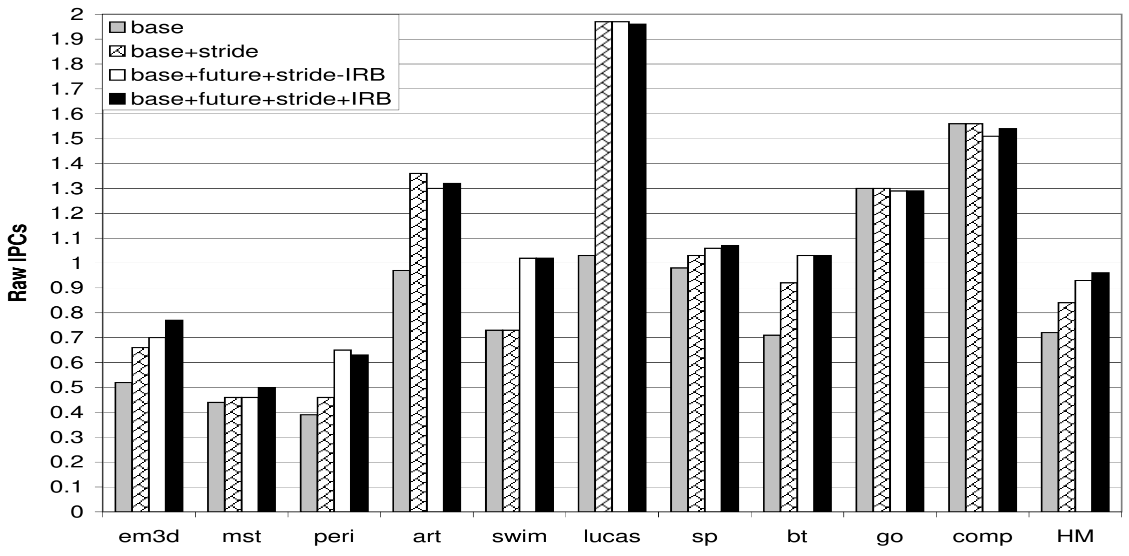


Figure 5.11: Raw IPCs for the base case, for the base case with a stride prefetcher, and for the combination of the *future* thread and the stride prefetcher (without and with an IRB).

Combination with Hardware Prefetch

The *future* thread has the ability to prefetch data in codes with unpredictable control and data flow. A number of the programs in this study also have regular accesses and can be effectively handled by the use of a hardware stride predictor [Chen and Baer, 1995]. We next evaluate the effect of such a hardware prefetch mechanism on the behavior of the base processor model and the *future* thread.

Figure 5.11 shows raw IPC numbers for four models. The first bar shows the IPC of the base case and the second bar shows the IPC for a base case augmented with a stride prefetcher. In our simulations, we use a 512-entry direct-mapped stride prefetcher (indexed by program counter) that maintains a finite state machine for each of the most recently issued load instructions. The finite state machine determines if the data accesses are regular. When a load is encountered, it issues a prefetch for the next load if the last two or more loads had the same stride. Prefetches are issued only if the stride exceeds 16 bytes in order to avoid unnecessary prefetches due to the spatial locality within a cache line. Prefetches contend for cache ports just like any other load operation and bring data into the various levels of cache, *i.e.*, no separate prefetch buffer is used.

We see big improvements for a number of programs, resulting in an overall speedup of 1.17. We also see significant improvements for pointer-chasing applications like *em3d* and *perimeter*. The data access patterns in these applications do not change over time, and hence, following the pointers results in accesses to regularly strided addresses. Such speedups would probably not be seen for other pointer-intensive codes with dynamic data structures. We then employ the *future* thread in addition to having the stride prefetcher[5] (shown by the third bar in the figure). We do not use the IRB in this case so as to isolate only the effects of prefetch, *natural reuse*, and early branch recovery. We notice reasonable improvements in a number of cases. Because the *future* thread jumps ahead and issues loads early, it also ends up issuing the prefetches early. This helps to hide the memory latency even beyond that which is possible with just the stride prefetcher. In the case of *lucas*, the stride prefetcher does a nearly perfect job, resulting in no additional speedups from the *future* thread. The overall speedup compared to the base case is 1.29. The last bar also includes an IRB, thus showing the full benefit of the *future* thread. The overall speedup is now 1.33.

These results indicate that the *future* thread can be combined with other forms of cache prefetching to yield even greater speedups. Because of *natural reuse* and the IRB, it also ‘prefetches’ ALU computations. It also initiates early branch recovery if the program is limited by a poor branch predictor.

Summary

Our results show that for an Alpha-like processor model, an impressive overall speedup of 1.21 is seen when using the *future* thread. The most important contributions to this improvement come from (i) a dynamic interval and exploration-based scheme that allocates registers between the *primary* and *future* threads, (ii) data prefetch and *natural reuse*, (iii) selectively employing early recovery from branch mispredicts, and (iv) instruction reuse because of the IRB. The eager

[5]When triggered, only the *future* thread can update the finite state machine of the stride predictor. Both threads issue prefetches.

release of registers and the timeout mechanism, both contribute synergistically to the ability of the *future* thread to jump ahead. We have explored a variety of processor models to study the sensitivity of our results to various parameters. Good performance was seen for different issue queue and register file sizes, and for future processor models. The combination with hardware stride prefetching resulted in further improved performance.

5.4 Additional Hardware Requirements

In this section, we qualitatively discuss the extra hardware required to implement the *future* thread.

Future Rename Table

The rename table for the *future* thread constitutes the biggest overhead in terms of transistors. In addition to the register mapping, it also stores the sequence number. While a conventional rename table checkpoints the mapping in the branch stack, the *future* rename table checkpoints the sequence number. Each cell in a rename table contains a shift register to checkpoint values. Since this constitutes most of the cell, the size of the value being checkpointed plays a large part in determining the table's access time [Palacharla *et al.*, 1997]. A conventional rename table checkpoints 7-bit values (the physical register tag), while the *future* rename table checkpoints the sequence number (a 9-10 bit value). While this implies a longer access time for the rename table, the results in Palacharla et al [Palacharla *et al.*, 1997] indicate that the rename table is not on the critical path for the technology parameters examined.

In addition, while looking up the *primary* rename table, an access has to be made to the Preg Status Table to detect if a mapping for that instruction exists in the *future* thread. This lookup can be done in parallel with the lookup of the *primary* rename table and should not affect the critical path.

The operations to be performed by the *future* rename table on a branch mispredict are another source of complexity. The checkpointed sequence number has to be examined and based on this, the mapping has to be copied from either the Preg Status Table or from the *primary* rename table. Since all these structures have a limited number of read and write ports, copying as many as 64 mappings could take a number of cycles. One possible optimization would be to checkpoint the actual mapping instead of the sequence number when it is known that the mapping cannot change. For example, if the sequence number indicates that the instruction that set this mapping has been dispatched in the *primary* thread, then it is known that this mapping will still be true when the branch mispredict is discovered. Hence, in this case, by checkpointing the mapping, a copy need not be made at the time of mispredict recovery. Even with these optimizations, it is still possible that the recovery could add a few cycles to the mispredict penalty for the *future* thread. We simulated the effect of an extra four cycle mispredict penalty and noticed only a 1% performance degradation in two of the programs with high misprediction rates. Given the opportunistic nature of the *future* thread, its mispredict penalty does not play a very major role in affecting performance.

Preg Status Table

The Preg Status Table stores the sequence number, the physical register tag, the users counter, the overwrite bit, the written bit, and the timeout counter. The overwrite bit will have to be checkpointed at every branch, and would hence include a shift register in its cell. This amounts to a total overhead of not more than 25 bits per entry. While it has been logically described as one structure, it can be broken up into a number of small CAM structures, as each field can be independently accessed. The most complex of these would be the users field which would need as many as 16 ports (corresponding to two operands for each of four instructions being renamed and four instructions being issued). This structure would still be a lot smaller than a rename table that has as many ports, much larger fields per entry, and many more entries. Also, the counters could be approximated by a simpler structure that used fewer ports as it is unlikely that all possible 16 ports would be used in a cycle.

In addition to these two structures, some negligible overhead is imposed by the branch FIFO (a simple array of bits, typically not exceeding 50 bits), and the *future* IFQ that buffers instructions before they are dispatched into the issue queue. A counter and associated logic would be required for the sequence number. The dynamic scheme requires a few additional counters to monitor the frequency of branches and cache accesses (hardware profiling counters are already present in most modern processors). The IRB, as implemented in our simulations, is a fairly simple structure. The S_n implementation scheme does not affect other processor critical paths. An IRB with only 16 entries can eliminate much of the redundancy that results from the execution of the *future* thread.

5.5 Other Approaches to Improving Register File Complexity

The primary focus of this chapter has been to improve performance by increasing the efficiency of the register file. By cleverly managing the register resources, a few registers can be used to support a large window of in-flight instructions. In our approach, register values are discarded to allow the mapping of distant instructions, thereby introducing the trade-off between nearby and distant ILP and necessitating its management. We have also explored other alternatives to improving the efficiency of the register file and describe that next.

The key observation in our study is that many register values have no active consumers and need not occupy valuable storage space in the register file that feeds the functional units. Hence, one approach to to improving register file efficiency is to move such register values into a backup storage [Balasubramonian *et al.*, 2001a; Balasubramonian *et al.*, 2001d]. In the event that we encounter a branch mispredict or an exception, these values are reinstated into the register file. This makes it possible to support an in-flight window as large as the combined size of the L1 and L2. The L2 is not frequently accessed and has small porting requirements, allowing it to have many entries with very low power, area, and cycle-time overheads. Such a mechanism introduces overhead to maintain sufficient state to allow recovery after a branch mispredict, but many of these introduced structures do not lie on cycle-time critical paths. Our results show that the use of a two-level structure helps reduce the access time of the first-level register file in comparison to a single-level register file for roughly the same IPC. When using the instructions per second metric, the two-level organization performs 17% better than the best single-level organization.

In [Balasubramonian *et al.*, 2001d], we explore a second mechanism to reduce register file complexity. Conventional register files are multi-ported structures, with each cell having as many ports as the maximum allowed bandwidth in any cycle. In [Balasubramonian *et al.*, 2001d], we propose a design where the register file is broken up into multiple banks, each with a single port. Thus, as many accesses can be made in a cycle as the number of banks, so long as no two accesses are made to the same bank. By enforcing this constraint and reducing the number of ports per cell in the register file, its area, access time, and power consumption are dramatically reduced. We show that such an organization has a minimal impact on IPC and entails a modest amount of complexity in the scheduling of instructions.

5.6 Related Work

Dundas and Mudge [Dundas and Mudge, 1997] introduced a scheme for halting the *main* instruction stream on a cache miss, and running ahead to prefetch data. However, this scheme was only applicable to an in-order machine with no instruction-level parallelism support. In our method, because the base processor is a dynamic superscalar architecture, the *main* instruction stream is never halted - once it runs out of registers, we use spare registers as a scratch-pad to make forward opportunistic progress. Thus, the implementation is vastly different, the performance benefits are far greater, and we are targeting cases where we run out of physical registers.

The idea of forming multiple threads that execute distant instructions has been exploited in a number of approaches, such as Multiscalar [Sohi *et al.*, 1995], Trace processors [Rotenberg *et al.*, 1997], DMT [Akkary and Driscoll, 1998], and TLDS [Steffan and Mowry, 1998], that use hardware or compiler generated threads to fork off computation that might be far ahead of the current commit stage of the processor. These are all hardware intensive solutions as they assume that there would effectively be a separate processing unit or a Simultaneous Multithreaded (SMT [Tullsen *et al.*, 1995]) base to execute these threads. They require significant hardware to store results and to transfer register values between threads to free up dependences. They are also highly speculative in nature, as these threads might lie much further ahead in the program control flow. The *future* thread is a lot less speculative as it starts off from where the *main* thread left off. Dependences are resolved just as in the baseline processor, without any added hardware (although we do complicate the dispatch stage). The *future* thread makes forward progress by relaxing the constraints on register release and by having the timeout mechanism, which are both unique to this approach.

Zilles and Sohi [Zilles and Sohi, 2000] characterize problem instructions (cache misses, branches) and the instructions that lead to them. They point out that a smaller subset of the program code can be pre-executed so that the *main* instruction stream does not encounter cache misses or branch mispredicts. They assume an underlying implementation that can pre-execute these slices. Roth and Sohi [Roth and Sohi, 2001] talk about such an implementation that can pre-execute certain dependence chains (hence, referred to as data-driven multithreading). They use profiling to generate these slices and annotate the code to trigger them at appropriate points. These threads use physical registers to store their results and they are integrated into the main program thread when it catches up. Our approach is less concerned with the ability to fork off a previously created thread early. It is a brute-force method that tries to make forward progress even when the registers run out, and does not rely on compiler or profiling help. We also show how an IRB can be used to integrate results back into the *main* thread instead of keeping results around in registers.

Other recent approaches to pre-execution include Speculative Precomputation [Collins *et al.*, 2001], where idle thread contexts in an SMT processor are used to spawn speculative threads that do data prefetch. These threads are identified with profiling help and are included as part of the binary. Another recent proposal by Annavaram et al [Annavaram *et al.*, 2001] also targets problem instructions when they are unable to execute because of a stalled dispatch stage. These instructions are flagged when they enter the instruction fetch queue (IFQ) and the dependence graph within the IFQ is computed in hardware to identify instructions leading up to the problem instruction. These instructions are executed in a separate engine so that data is appropriately brought in to the data caches.

There have also been a couple of attempts at improving branch resolution by pre-execution [Farcy *et al.*, 1998; Roth *et al.*, 1999], where the slice determining the branch is duplicated and made to run in a separate window (sharing the same physical registers). Farcy et al [Farcy *et al.*, 1998] notice regularity in the branch condition computations and use value prediction to accelerate the second thread.

Simultaneous Subordinate Microthreading (SSMT) [Chappell *et al.*, 1999] and Assisted Execution [Dubois and Song, 1998] are similar schemes where custom-generated threads are invoked within the hardware by certain events. These threads do very simple specific things and cannot be automatically generated. There has been recent work in the area of pre-execution. Most notably, Mutlu et al. [Mutlu *et al.*, 2003] propose a mechanism where the architectural registers are check-pointed before spawning the pre-execution instructions.

A related concept is AR-SMT [Rotenberg, 1999] and SRT [Reinhardt and Mukherjee, 2000], which run two copies of the same program on an SMT processor and compare results from both threads. Their goal is to detect transient faults in a chip, rather than to enhance performance. An extension of this is the Slipstream processor [Sundaramoorthy *et al.*, 2000], where the thread running ahead is a shortened version of the original program (dynamically created by detecting and eliminating ineffectual pieces of the program), and the trailing thread is the full program that verifies the correct working of the leading thread. The two programs together can run faster than the single original program because the leading thread communicates values and branch outcomes to the trailing thread as (often correct) predictions.

The primary advantage of the *future* thread is its prefetching effect. A number of hardware [Chen and Baer, 1995; Jouppi, 1990; Roth *et al.*, 1998] and software prefetching [Luk and Mowry, 1996; Callahan and Porterfield, 1990; Mowry *et al.*, 1992] schemes have been proposed. Most of these schemes can do a better job of prefetching as they exploit some higher-level program information (regularity of accesses). This regularity can be determined at compile time or as strides or load-value dependences in hardware. This lack of high-level information prevents us from doing a very effective job of prefetching. We, however, do a more exact job as we respect dependences and actually compute load addresses (rather than use heuristics like most hardware prefetch schemes). We also use dynamic branch prediction to follow the probable control-flow path, instead of greedily prefetching [Luk and Mowry, 1996] along all possible paths. This prevents us from fetching useless lines into the cache (unless we are on the wrong branch path). Hence, our techniques are also applicable to irregular codes with unpredictable control flow and unpredictable data accesses. Luk [Luk, 2001] addresses a similar problem in the context of an SMT processor by using the compiler to help pre-execute these codes. We do not add instruction overhead, unlike software prefetching schemes. Some of the prefetch schemes can also be combined with the *future* thread to yield greater speedups, as demonstrated in Section 5.3.

There has also been a software approach to tackling the problem of a single cache miss holding up the ROB. Pai and Adve [Pai and Adve, 1999] describe a compiler algorithm that restructures code so that cache misses are clustered, thereby increasing the memory parallelism while the ROB is stalled. Compiler based methods can only be targeted at regular codes. The *future* thread can jump ahead and issue *future* memory operations while the *main* instruction window is stalled, thus achieving the same effect of improved memory parallelism.

To our knowledge, an Instruction Reuse Buffer [Sodani and Sohi, 1997] has only been applied to exploit value locality and squash reuse (reuse of instructions that get squashed because of a mispredict, but lie on both paths out of the branch). Our proposal is a novel application of the IRB.

5.7 Summary

In this chapter, we have designed and evaluated a microarchitecture that dynamically allocates a portion of the processor's physical resources to a *future* thread in order to exploit distant ILP in addition to nearby ILP. Long latency instructions tend to stall the commit phase of a traditional superscalar architecture on reaching the head of the re-order buffer. Subsequent instructions use up the available physical registers, after which the dispatch stage stalls. In our proposed microarchitecture, part of the physical registers are allocated for the *main* program and once they are consumed, the *future* thread gets triggered and makes forward progress. It eagerly releases registers and times out instructions that wait too long in order to opportunistically advance far beyond what the *primary* thread is capable of. It thus improves performance by resolving branch mispredicts early, by warming up the data cache, the instruction cache, and the instruction reuse buffer, and by reusing register mappings and values. Since resources are taken away from the *primary* thread to allow the *future* thread to advance, we are trading off nearby ILP in order to mine distant ILP. Depending on the ILP characteristics of the program, there exists an optimal allocation of resources between the two threads. We found that an interval and exploration-based mechanism is very effective at computing this optimal allocation at run-time. A mechanism based on fine-grained positional adaptation did not work as well because of the difficulty in estimating the effect of the *future* thread over short intervals of time. We did not evaluate metrics to predict the behavior of the *future* thread because of the sheer complexity involved in reliably computing such a metric.

Our evaluation on some of the more memory-intensive benchmarks shows very promising speedups of up to 1.64. The overall improvement on our benchmark suite is 21%. The contributions come mainly from prefetching and computation reuse, with significant contributions from early branch recovery in the programs limited by poor branch prediction accuracies. The use of a small 16-entry IRB accounts for 5% of this improvement. The dynamic allocation of registers plays a major role in tuning the hardware to the ILP requirements of each program phase. The dynamic choice of whether to use early branch resolution based on the branch predictor accuracy also improves performance. The use of a larger issue queue allows the *future* thread to achieve an overall speedup of 1.25. We also observed impressive speedups for a variety of processor models and even in combination with a hardware stride prefetcher.

6 Conclusions

This dissertation studies technology trends and their impact on the design of modern microprocessors. The increasing dominance of wire delays presents an important challenge for architects – the management of long on and off-chip latencies. These long latencies introduce trade-offs in the design of different microprocessor structures. Chapter 3 highlights the trade-off between capacity and access times within the L1 and L2 data caches. In Chapter 4, we observe the trade-off between communication and parallelism in a highly clustered processor. Chapter 5 demonstrates the importance of supporting a large window of in-flight instructions with few resources, making it necessary to allocate the limited processor resources for either nearby or distant parallelism. Thus, we have observed a number of trade-offs emerging in modern processors, caused either by long wire delays in storage structures or by the need to tolerate long DRAM latencies.

Adaptation Algorithms

In order to manage trade-offs at run-time without programmer or compiler intervention, we propose a number of algorithms that detect the program's characteristics and accordingly match the hardware to the program's needs. These mechanisms attempt reconfiguration at regular time intervals or at specific positions in the code, i.e., branches or subroutine call/returns. They select the best organization either by profiling the candidate hardware organizations or by relying on program metrics that can predict the optimal organization. The interval-based algorithms require only a few hardware counters for bookkeeping and entail negligible execution-time overhead. We found that for most programs, we were able to determine an interval length such that program behavior was captured and this behavior was very consistent over a large number of intervals. The algorithms based on positional adaptation require a hardware table that stores state for each encountered branch or subroutine. Since positional adaptation allows a quick reaction to a phase change, it is very effective at targeting small phases. The algorithms that predict the optimal organization need additional logic to compute various program metrics.

The Reconfigurable Cache

In Chapter 3, we proposed an adaptive cache layout that varied the boundary between the L1 and L2. The layout exploits the presence of repeaters in cache wordlines to allow for reconfiguration options in a low-intrusive manner. Each program phase was allocated exactly the amount of L1 cache space that balanced the L1 miss rate and the L1 access time. This amount was usually equal to the working set size of the program phase. This not only improves performance, it also reduces the number of transfers between the different levels of the cache, thereby reducing total power consumption. The dynamic control algorithms are very effective at handling different program behaviors: we observed overall performance improvements of as much as 15% and energy savings of as much as 42% across different processor organizations.

Clustered Processors

In order to exploit available chip real-estate, processors are increasingly supporting a large number of threads, resulting in large resource sizes. A clustered organization helps reduce the complexity in designing such a large processor. In Chapter 4, we examine the scalability of single-thread performance on a highly clustered and communication-bound processor. A large number of clusters imply high communication costs, but most programs do not have sufficient parallelism in them to exploit all the clusters. A subset of clusters yields optimal performance for each program phase. Note that the clustered organization is necessitated by current technology trends; our approach facilitates the easy management of cluster resources. Once the optimal number of clusters has been determined, instructions and data are restricted to this subset. This not only improves overall single-thread performance by 15%, it also frees up more than half the clusters so they can be used for other purposes (for use by other threads or to reduce leakage energy).

Pre-Execution

The proposal in Chapter 5 also attempts to mine a high degree of parallelism, but with a limited number of resources. We employ a pre-execution thread to use resources efficiently and jump further ahead to warm up various processor structures. The allocation of resources between the main program thread and the pre-execution thread determines the priority between nearby and distant parallelism. For example, a program with limited distant parallelism performs better when all the resources are allocated for the main thread. On the other hand, a program that is limited by memory accesses performs better when resources are allocated to the pre-execution thread, allowing it to make forward progress and prefetch data into the caches. The overall improvement over a base case without a pre-execution thread was as much as 21%, with half the improvement coming from algorithms to dynamically allocate resources and tune processor policies. We have also evaluated a two-level register file organization that allows the dispatch of a large in-flight instruction window without increasing the complexity of the first-level register file [Balasubramonian *et al.*, 2001d].

Comparing the Adaptation Algorithms

We found that the interval-based mechanism with exploration worked well in every context. It reliably captures program behavior in most cases and accurately selects the optimal organization. The exploration process was especially effective because the number of candidate organizations was limited in each of our studies. The hardware overhead of this mechanism is minimal and it achieves most of the potential speed-ups from dynamic adaptation. Replacing the exploration process with predictive techniques was effective for the reconfigurable cache and for the clustered organization. In each of these settings, there were metrics (the working set size and the degree of distant parallelism) that were relatively easy to compute and that were excellent indicators of the optimal organization. It was harder to devise metrics to estimate the optimal allocation of resources while employing the pre-execution thread. The interval-based mechanism can entail heavy overheads if the program exhibits a high degree of variability and if the number of candidate organizations is very large. We also found that positional adaptation was very effective while determining the optimal number of clusters. The primary advantage of this mechanism is its ability to target small phases and this is beneficial only if the hardware can react equally quickly to any reconfiguration. With the cache and with the pre-execution thread, we found that the hardware was generally slower to adapt, causing an interval-based mechanism to be more effective. Positional adaptation can result in inaccurate configuration selections if the behavior of a code section changes over time.

Future Work

As future work, we intend to verify the effective behavior of our control algorithms over a wider range of trade-off settings. With the growing emphasis on power consumption, architects are being forced to make a number of choices that trade off some performance for better power efficiency. In order to control the performance loss, adaptive hardware can be employed that goes into low-power mode only when the performance penalty is guaranteed to be negligible. Such an approach can also help control thermal emergencies. There exist other performance-centric trade-offs that might also benefit from adaptation, for example, choice of branch predictors, use of control speculation, use of value speculation, etc.

If adaptation algorithms are employed for a number of structures on the chip, the interaction of these mechanisms with each other is yet unclear. It is unlikely that the different adaptive hardware units can monitor their behavior independently of each other, necessitating that the algorithms be modified. As the number of hardware units increases, the number of candidate hardware organizations increases dramatically. The overhead from exploring all of these organizations might be significant enough that other metrics might be required to predict the optimal hardware structures.

We would like to extend our cluster allocation algorithms to multi-threaded workloads to study how total processor throughput could be maximized. Since a subset of all clusters can yield optimal performance for each thread, partitioning the clusters among the threads can help maximize single-thread response time as well as total throughput. Such algorithms will have to consider the allocation of cache banks and coherence among them when using a decentralized cache.

The study of the clustered processors has exposed us to many challenging problems in their design. The comparison between a centralized and a decentralized cache deserves more attention. A decentralized cache can dramatically reduce inter-cluster communication if accurate bank predictors can be built. The inter-cluster interconnect has a significant effect on performance. A grid interconnect allows much higher ILP than a ring interconnect because of its higher connectivity, but its implementation complexity is also higher. We plan a more detailed evaluation of the effect of these choices on performance, latencies, layouts, wiring complexity, power, etc. Further, a multi-threaded workload might benefit from an interconnect that has high connectivity within certain subsets of clusters. These research efforts would continue to reduce the negative impact of long wire delays and memory latencies in future generations of processors.

Bibliography

[Agarwal *et al.*, 2002] A. Agarwal, H. Li, and K.Roy, "DRG-Cache: A Data Retention Gated-Ground Cache for Low Power," In *Proceedings of the 39th Conference on Design Automation*, June 2002.

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

[Aggarwal and Franklin, 2001] A. Aggarwal and M. Franklin, "An Empirical Study of the Scalability Aspects of Instruction Distribution Algorithms for Clustered Processors," In *Proceedings of ISPASS*, 2001.

[Akkary and Driscoll, 1998] H. Akkary and M. Driscoll, "A Dynamic Multithreading Processor," In *Proceedings of MICRO-31*, pages 226–236, November 1998.

[Albonesi, 1998] D.H. Albonesi, "Dynamic IPC/Clock Rate Optimization," *Proceedings of ISCA-25*, pages 282–292, June 1998.

[Albonesi, 1999] D.H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation," *Proceedings of MICRO-32*, pages 248–259, November 1999.

[Annavaram *et al.*, 2001] M. Annavaram, J. Patel, and E. Davidson, "Data Prefetching by Dependence Graph Precomputation," In *Proceedings of ISCA-28*, pages 52–61, July 2001.

[Association, 1999] Semiconductor Industry Association, "The National Technology Roadmap for Engineers," Technical report, 1999.

[Bahar and Manne, 2001] R. I. Bahar and S. Manne, "Power and Energy Reduction Via Pipeline Balancing," In *Proceedings of ISCA-28*, pages 218–229, July 2001.

[Bailey *et al.*, 1991] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, D. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," *International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[Bakoglu and Meindl, 1985] H.B. Bakoglu and J.D. Meindl, "Optimal Interconnect Circuits for VLSI," *IEEE Transactions on Computers*, 32(5):903–909, May 1985.

[Balasubramonian *et al.*, 2000a] R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Dynamic Memory Hierarchy Performance Optimization," *Workshop on Solving the Memory Wall Problem*, June 2000.

[Balasubramonian *et al.*, 2000b] R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures," In *Proceedings of MICRO-33*, pages 245–257, December 2000.

[Balasubramonian *et al.*, 2003a] R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "A Dynamically Tunable Memory Hierarchy," *IEEE Transactions on Computers (to appear)*, 2003.

[Balasubramonian *et al.*, 2001a] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi, "A High-Performance Two-Level Register File Organization," Technical Report 745, University of Rochester, April 2001.

[Balasubramonian *et al.*, 2001b] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi, "Dynamically Allocating Processor Resources between Nearby and Distant ILP," Technical Report 743, University of Rochester, April 2001.

[Balasubramonian *et al.*, 2001c] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi, "Dynamically Allocating Processor Resources between Nearby and Distant ILP," In *Proceedings of ISCA-28*, pages 26–37, July 2001.

[Balasubramonian *et al.*, 2001d] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi, "Reducing the Complexity of the Register File in Dynamic Superscalar Processors," In *Proceedings of MICRO-34*, pages 237–248, December 2001.

[Balasubramonian *et al.*, 2002] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi, "Microarchitectural Trade-Offs in the Design of a Scalable Clustered Microprocessor," Technical Report 771, University of Rochester, January 2002.

[Balasubramonian *et al.*, 2003b] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi, "Dynamically Managing the Communication-Parallelism Trade-Off in Future Clustered Processors," In *Proceedings of ISCA-30*, pages 275–286, June 2003.

[Baniasadi and Moshovos, 2000] A. Baniasadi and A. Moshovos, "Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors," In *Proceedings of MICRO-33*, pages 337–347, December 2000.

[Bannon, 1998] P. Bannon, "Alpha 21364: A Scalable Single-Chip SMP," *Microprocessor Forum*, October 1998.

[Burger and Austin, 1997] D. Burger and T. Austin, "The Simplescalar Toolset, Version 2.0," Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.

[Buyuktosunoglu *et al.*, 2000] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D.H. Albonesi, "An Adaptive Issue Queue for Reduced Power at High Performance," In *Workshop on Power-Aware Computer Systems (PACS2000, held in conjunction with ASPLOS-IX)*, November 2000.

[Buyuktosunoglu *et al.*, 2001] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D.H. Albonesi, "A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors," In *Proceedings of the 11th Great Lakes Symposium on VLSI*, pages 73–78, March 2001.

[Callahan and Porterfield, 1990] D. Callahan and A. Porterfield, "Data Cache Performance of Supercomputer Applications," In *Proceedings of ICS*, pages 564–572, January 1990.

[Canal *et al.*, 2000] R. Canal, J. M. Parcerisa, and A. Gonzalez, “Dynamic Cluster Assignment Mechanisms,” In *Proceedings of HPCA-6*, pages 132–142, January 2000.

[Canal *et al.*, 2001] R. Canal, J. M. Parcerisa, and A. Gonzalez, “Dynamic Code Partitioning for Clustered Architectures,” *International Journal of Parallel Programming*, 29(1):59–79, 2001.

[Capitanio *et al.*, 1992] A. Capitanio, N. Dutt, and A. Nicolau, “Partitioned Register Files for VLIWs: A Preliminary Analysis of Trade-offs,” In *Proceedings of MICRO-25*, pages 292–300, December 1992.

[Chappell *et al.*, 1999] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt, “Simultaneous Subordinate Microthreading (SSMT),” In *Proceedings of ISCA-26*, pages 186–195, May 1999.

[Chen and Baer, 1995] T. Chen and J. Baer, “Effective Hardware Based Data Prefetching for High Performance Processors,” *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[Cho *et al.*, 1999a] S. Cho, P-C. Yew, and G. Lee, “Access Region Locality for High-Bandwidth Processor Memory System Design,” In *Proceedings of MICRO-32*, pages 136–146, November 1999.

[Cho *et al.*, 1999b] S. Cho, P-C. Yew, and G. Lee, “Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor,” In *Proceedings of ISCA-26*, pages 100–110, May 1999.

[Collins *et al.*, 2001] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y-F. Lee, D. Lavery, and J. Shen, “Speculative Precomputation: Long-Range Prefetching of Delinquent Loads,” In *Proceedings of ISCA-28*, pages 14–25, July 2001.

[Cruz *et al.*, 2000] J-L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham, “Multiple-Banked Register File Architectures,” In *Proceedings of ISCA-27*, pages 316–325, June 2000.

[Dahlgren and Stenstrom, 1991] F. Dahlgren and P. Stenstrom, “On Reconfigurable On-Chip Data Caches,” In *Proceedings of MICRO-24*, pages 189–198, 1991.

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

[Dhodapkar and Smith, 2002] A. Dhodapkar and J. E. Smith, "Managing Multi-Configurable Hardware via Dynamic Working Set Analysis," In *Proceedings of ISCA-29*, pages 233–244, May 2002.

[Dropsho *et al.*, 2002] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. L. Scott, "Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power," In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 141–152, September 2002.

[Dubois and Song, 1998] M. Dubois and Y. H. Song, "Assisted Execution," Technical Report CENG 98-25, EE-Systems, University of Southern California, October 1998.

[Dundas and Mudge, 1997] J. Dundas and T. Mudge, "Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss," In *Proceedings of ICS*, pages 68–75, 1997.

[Farcy *et al.*, 1998] A. Farcy, O. Temam, R. Espasa, and T. Juan, "Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes," In *Proceedings of MICRO-31*, pages 59–68, November 1998.

[Farkas *et al.*, 1997] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic, "The Multicluster Architecture: Reducing Cycle Time through Partitioning," In *Proceedings of MICRO-30*, pages 149–159, December 1997.

[Farkas *et al.*, 1996] K. Farkas, N. Jouppi, and P. Chow, "Register File Considerations in Dynamically Scheduled Processors," In *Proceedings of HPCA-2*, pages 40–51, February 1996.

[Farkas and Jouppi, 1994] K.I. Farkas and N.P. Jouppi, "Complexity/Performance Tradeoffs with Non-Blocking Loads," *Proceedings of ISCA-21*, pages 211–222, April 1994.

[Fields *et al.*, 2001] B. Fields, S. Rubin, and R. Bodik, "Focusing Processor Policies via Critical-Path Prediction," In *Proceedings of ISCA-28*, pages 74–85, July 2001.

[Fisk and Bahar, 1999] B. Fisk and I. Bahar, “The Non-Critical Buffer: Using Load Latency Tolerance to Improve Data Cache Efficiency,” In *IEEE International Conference on Computer Design*, pages 538–545, October 1999.

[Flautner *et al.*, 2002] K. Flautner, N.S. Kim, S. Martin, D. Blaauw, and T. Mudge, “Drowsy Caches: Simple Techniques for Reducing Leakage Power,” In *Proceedings of ISCA-29*, May 2002.

[Fleischman, 1999] J. Fleischman, “Private Communication,” October 1999.

[Folegnani and Gonzalez, 2000] D. Folegnani and A. Gonzalez, “Reducing Power Consumption of the Issue Logic,” In *Workshop on Complexity-Effective Design (WCED2000, held in conjunction with ISCA-27)*, June 2000.

[Ghiasi *et al.*, 2000] S. Ghiasi, J. Casmira, and D. Grunwald, “Using IPC Variations in Workloads with Externally Specified Rates to Reduce Power Consumption,” In *Workshop on Complexity Effective Design (WCED2000, held in conjunction with ISCA-27)*, June 2000.

[Gowan *et al.*, 1998] M. Gowan, L. Biro, and D. Jackson, “Power Considerations in the Design of the Alpha 21264 Microprocessor,” In *Proceedings of the 35th Design Automation Conference*, 1998.

[Gwennap, 1997] L. Gwennap, “PA-8500’s 1.5M Cache Aids Performance,” *Microprocessor Report*, 11(15), November 17, 1997.

[Hennessy and Patterson, 1996] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 2nd edition, 1996.

[Heo *et al.*, 2002] S. Heo, K. Barr, M. Hampton, and K. Asanovic, “Dynamic Fine-Grain Leakage Reduction Using Leakage-Biased Bitlines,” In *Proceedings of ISCA-29*, May 2002.

[Huang *et al.*, 2003] M. Huang, J. Renau, and J. Torrellas, “Positional Adaptation of Processors: Applications to Energy Reduction,” In *Proceedings of ISCA-30*, pages 157–168, June 2003.

[Huang *et al.*, 2000] M. Huang, J. Reneau, S.M. Yoo, and J. Torrellas, "A Framework for Dynamic Energy Efficiency and Temperature Management," In *Proceedings of MICRO-33*, pages 202–213, December 2000.

[Jouppi, 1990] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," In *Proceedings of ISCA-17*, pages 364–373, May 1990.

[Kamble and Ghose, 1997] M.B. Kamble and K. Ghose, "Analytical Energy Dissipation Models for Low Power Caches," *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 143–148, August 1997.

[Kaxiras *et al.*, 2001] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power," In *Proceedings of ISCA-28*, July 2001.

[Keckler and Dally, 1992] S.W. Keckler and W.J. Dally, "Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism," In *Proceedings of ISCA-19*, pages 202–213, May 1992.

[Kessler, 1999] R. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro*, 19(2):24–36, March/April 1999.

[Kessler *et al.*, 1998] R.E. Kessler, E.J McLellan, and D.A. Webb, "The Alpha 21264 Microprocessor Architecture," In *Proceedings of ICCD*, 1998.

[Kumar, 1997] A. Kumar, "The HP PA-8000 RISC CPU," *IEEE Computer*, 17(2):27–32, March 1997.

[Lee *et al.*, 1997] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," In *Proceedings of MICRO-30*, pages 330–335, December 1997.

[Lesartre and Hunt, 1997] G. Lesartre and D. Hunt, "PA-8500: The Continuing Evolution of the PA-8000 Family," *Proceedings of Compcon*, 1997.

[Lowney *et al.*, 1993] P.G. Lowney, S. Freudenberger, T. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnell, and J.C. Ruttenberg, "The Multiflow Trace Scheduling Compiler," *Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[Luk, 2001] C-K. Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," In *Proceedings of ISCA-28*, pages 40–51, July 2001.

[Luk and Mowry, 1996] C-K. Luk and T. Mowry, "Compiler-based Prefetching for Recursive Data Structures," In *Proceedings of ASPLOS VII*, pages 222–233, 1996.

[Magklis *et al.*, 2003] G. Magklis, M.L. Scott, G. Semeraro, D.H. Albonesi, and S. Dropsho, "Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor," In *Proceedings of ISCA-30*, June 2003.

[Matzke, 1997] D. Matzke, "Will Physical Scalability Sabotage Performance Gains?," *IEEE Computer*, 30(9):37–39, September 1997.

[McFarland, 1997] G.W. McFarland, *CMOS Technology Scaling and Its Impact on Cache Delay*, PhD thesis, Stanford University, June 1997.

[McFarland and Flynn, 1995] G.W. McFarland and M. Flynn, "Limits of Scaling MOSFETS," Technical Report CSL-TR-95-62, Stanford University, November 1995.

[Mowry *et al.*, 1992] T. Mowry, M. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," In *Proceedings of ASPLOS-V*, pages 62–73, 1992.

[Mutlu *et al.*, 2003] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," In *Proceedings of HPCA-9*, February 2003.

[Nagarajan *et al.*, 2001] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler, "A Design Space Evaluation of Grid Processor Architectures," In *Proceedings of MICRO-34*, pages 40–51, December 2001.

[Nii *et al.*, 1998] K. Nii, H. Makino, Y. Tujihashi, C. Morishima, Y. Hayakawa, H. Nunogami, T. Arakawa, and H. Hamano, "A Low Power SRAM Using Auto-Backgate-Controlled MT-CMOS," In *Proceedings of ISLPED*, 1998.

[Pai and Adve, 1999] V. Pai and S. Adve, "Code Transformations to Improve Memory Parallelism," In *Proceedings of MICRO-32*, pages 147–155, November 1999.

[Palacharla *et al.*, 1997] S. Palacharla, N. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors," In *Proceedings of ISCA-24*, pages 206–218, June 1997.

[Parcerisa *et al.*, 2002] J-M. Parcerisa, J. Sahuquillo, A. Gonzalez, and J. Duato, "Efficient Interconnects for Clustered Microarchitectures," In *Proceedings of PACT*, September 2002.

[Ponomarev *et al.*, 2001] D.V. Ponomarev, G. Kucuk, and K. Ghose, "Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources," In *Proceedings of MICRO-34*, pages 90–101, December 2001.

[Powell *et al.*, 2001] M. Powell, A. Agrawal, T.N. Vijaykumar, B. Falsafi, and K. Roy, "Reducing Set-Associative Cache Energy via Selective Direct-Mapping and Way Prediction," In *Proceedings of MICRO-34*, December 2001.

[Ranganathan and Franklin, 1998] N. Ranganathan and M. Franklin, "An Empirical Study of Decentralized ILP Execution Models," In *Proceedings of ASPLOS-VIII*, pages 272–281, October 1998.

[Ranganathan *et al.*, 2000] P. Ranganathan, S. Adve, and N.P. Jouppi, "Reconfigurable Caches and Their Application to Media Processing," *Proceedings of ISCA-27*, pages 214–224, June 2000.

[Reinhardt and Mukherjee, 2000] S. Reinhardt and S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," In *Proceedings of ISCA-27*, pages 25–36, June 2000.

[Rivers *et al.*, 1997] J. Rivers, G. Tyson, E. Davidson, and T. Austin, "On High-Bandwidth Data Cache Design for Multi-Issue Processors," In *Proceedings of MICRO-30*, pages 46–56, December 1997.

[Rogers *et al.*, 1995] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren, "Supporting Dynamic Data Structures on Distributed Memory Machines," *ACM TOPLAS*, March 1995.

[Rotenberg, 1999] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," In *Proceedings of 29th International Symposium on Fault-Tolerant Computing*, June 1999.

[Rotenberg *et al.*, 1997] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J.E. Smith, "Trace Processors," In *Proceedings of MICRO-30*, pages 138–148, December 1997.

[Roth *et al.*, 1998] A. Roth, A. Moshovos, and G. Sohi, "Dependence Based Prefetching for Linked Data Structures," In *Proceedings of ASPLOS VIII*, pages 115–126, October 1998.

[Roth *et al.*, 1999] A. Roth, A. Moshovos, and G. Sohi, "Improving Virtual Function Call Target Prediction via Dependence-based Pre-computation," In *Proceedings of ICS*, pages 356–364, June 1999.

[Roth and Sohi, 2001] A. Roth and G. Sohi, "Speculative Data-Driven Multithreading," In *Proceedings of HPCA-7*, January 2001.

[Sherwood *et al.*, 2003] T. Sherwood, S. Sair, and B. Calder, "Phase Tracking and Prediction," In *Proceedings of ISCA-30*, June 2003.

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

[Smith and Sohi, 1995] J.E. Smith and G.S. Sohi, "The Microarchitecture of Superscalar Processors," *Proceedings of the IEEE*, 83:1609–1624, December 1995.

[Sodani and Sohi, 1997] A. Sodani and G. Sohi, "Dynamic Instruction Reuse," In *Proceedings of ISCA-24*, pages 194–205, June 1997.

[Sohi *et al.*, 1995] G. Sohi, S. Breach, and T.N. Vijaykumar, "Multiscalar Processors," In *Proceedings of ISCA-22*, pages 414–425, June 1995.

[Srinivasan *et al.*, 2001] S. Srinivasan, R. Ju, A. Lebeck, and C. Wilkerson, "Locality vs. Criticality," In *Proceedings of ISCA-28*, pages 132–143, July 2001.

[Srinivasan and Lebeck, 1999] S. T. Srinivasan and A. R. Lebeck, "Load Latency Tolerance in Dynamically Scheduled Processors," *Journal of Instruction-Level Parallelism*, 1, October 1999.

[Steffan and Mowry, 1998] J. Steffan and T. Mowry, "The Potential for Using Thread Level Data-Speculation to Facilitate Automatic Parallelization," In *Proceedings of HPCA-4*, pages 2–13, February 1998.

[Sundaramoorthy *et al.*, 2000] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream Processors: Improving both Performance and Fault Tolerance," In *Proceedings of ASPLOS-IX*, pages 257–268, November 2000.

[Tiwari *et al.*, 1998] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez, "Reducing Power in High-Performance Microprocessors," In *Proceedings of the 35th Design Automation Conference*, 1998.

[Tullsen *et al.*, 1995] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," In *Proceedings of ISCA-22*, pages 392–403, June 1995.

[Tune *et al.*, 2001] E. Tune, D. Liang, D. Tullsen, and B. Calder, "Dynamic Prediction of Critical Path Instructions," In *Proceedings of HPCA-7*, pages 185–196, January 2001.

[Veidenbaum *et al.*, 1999] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji, "Adapting Cache Line Size to Application Behavior," In *Proceedings of ICS*, pages 145–154, June 1999.

[Wulf and McKee, 1995] Wm. A. Wulf and S.A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News*, 23(1):20–24, March 1995.

[Yang *et al.*, 2001] S.H. Yang, M. Powell, B. Falsafi, K. Roy, and T.N. Vijaykumar, "An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep Submicron High-Performance I-Caches," In *Proceedings of HPCA-7*, pages 147–158, January 2001.

[Yeager, 1996] K. Yeager, “The MIPS R10000 Superscalar Microprocessor,” *IEEE Micro*, 16(2):28–41, April 1996.

[Yoaz *et al.*, 1999] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, “Speculation Techniques for Improving Load Related Instruction Scheduling,” In *Proceedings of ISCA-26*, pages 42–53, May 1999.

[Zilles and Sohi, 2000] C. Zilles and G. Sohi, “Understanding the Backward Slices of Performance Degrading Instructions,” In *Proceedings of ISCA-27*, pages 172–181, June 2000.

[Zyuban and Kogge, 2001] V. Zyuban and P. Kogge, “Inherently Lower-Power High-Performance Superscalar Architectures,” *IEEE Transactions on Computers*, March 2001.