THE TIMEKEEPING METHODOLOGY: EXPLOITING GENERATIONAL LIFETIME BEHAVIOR TO IMPROVE PROCESSOR POWER AND PERFORMANCE

ZHIGANG HU

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
ELECTRICAL ENGINEERING

NOVEMBER 2002

© Copyright by Zhigang Hu, 2002.

All Rights Reserved

Abstract

Today's CPU designers face increasingly aggressive CPU performance goals while also dealing with challenging limits on power dissipation. The conflict of performance and power requirements increases the importance of simple but effective techniques. This thesis demonstrates how processors can be optimized by exploiting knowledge about time durations between key processor and memory events. These "timekeeping" techniques can give performance or power improvements with simple hardware.

We use the memory hierarchy as the main example to illustrate the timekeeping methodology. First we introduce some basic concepts, including the generational nature of cache behavior. By exploiting characteristics of key timekeeping metrics, we show how they form the basis for a rich set of policies to classify and predict future program behavior. Hardware mechanisms are then proposed to harness these predictions.

Three techniques are presented as applications of the timekeeping methodology to the memory system. The first mechanism, *cache decay*, can reduce cache leakage energy by 4X by identifying long-idle cache lines using simple 2-bit counters and turning them off. The second mechanism, a *timekeeping victim cache filter*, uses similar 2-bit counters to identify cache lines with short dead times and chooses them as candidates for the victim cache. This filters out 87% of victim cache traffic while improving performance. In the third mechanism, *timekeeping prefetch*, we exploit regularity across consecutive generations of the same cache line, using information from the previous generation as predictions for the current generation. The resulting prefetcher is highly effective and also hardware-efficient. With an 8KB history table, an average performance improvement of 11% is achieved for SPEC2000.

Outside the memory system, we also show how to apply the timekeeping methodology to other subsystems such as branch predictors. A key characteristic of predictor data is that they are execution hints that do not affect program correctness. To exploit this charac-

teristic, we propose to use naturally decaying 4-transistor cells to build branch predictors, instead of traditional 6-transistor cells. This reduces branch predictor leakage by 60-80% with cell area savings up to 33%.

The techniques presented clearly demonstrate the power of the timekeeping methodology. We expect that in our future work, as well as in work by others, more timekeeping techniques will be proposed to help to meet the many challenges in future processors.

Acknowledgments

First and foremost, I would like to gratefully thank my advisor, Professor Margaret Martonosi, for leading my way into the wonderful field of computer architecture, and for guiding my numerous random thoughts to solid, reportable work. I would also like to acknowledge Dr. Stefanos Kaxiras, for providing a precious opportunity of summer internship at Bell Labs, and for taking the role of a co-advisor during the key years of my graduate study. Thanks also go to my other co-authors, Profs Kevin Skadron and Doug Clark, Philo Juang, Dr. Girijia Narlikar, Dr. Phil Diodato, Dr. Alan Berenbaum, and Dr. Rae McLellan.

I am very grateful to the other Computer Engineering, Computer Science, and Information and System Sciences faculty at Princeton, for teaching me a broad range of knowledge, from which I will benefit for my whole life. Special thanks go to my academia advisor, Professor Niraj Jha, for directing me in the first year of my graduate study and for teaching me a key course for my research.

Many researchers outside Princeton also helped my research, either through insightful discussions, or by providing tools or advice for my research. Among them are Dr. Anchow Lai, Dr. Pradip Bose, Dr. George Cai and many others.

Last and most importantly, my deepest thanks to my family, especially my parents, for their support and encouragement over the years. Finally, to my wife Liya Wan, for her love, support and always taking care of me.

Contents

Abstract iii

1 Introduction **1**

1.1 Introduction 1

1.2 Contributions 5

1.2.1 Timekeeping in the Memory System 6

1.2.2 Timekeeping in Other Systems 8

1.3 Organization 9

2 The Timekeeping Methodology **10**

2.1 The Timekeeping Methodology: Introduction 10

2.2 Timekeeping in the Memory System 17

2.2.1 Introduction to Timekeeping Metrics in the Memory System 17

2.2.2 Simulation Model Parameters 19

2.2.3 Statistical Distributions of Timekeeping Metrics 22

2.2.4 Using Timekeeping Metrics to Predict Conflict Misses 23

2.2.5 Summary 29

2.3 Timekeeping Victim Cache Filter 30

2.3.1 Introduction 30

2.3.2 Simulation Results 32

2.3.3 Adaptive Scheme 33
2.3.4 Timekeeping Victim Cache Filter: Summary 34
2.4 Related Work 35
2.5 Chapter Summary 36

3 Cache Decay **38**
3.1 Potential Benefits 38
3.2 Timekeeping for Leakage Control: Cache Decay 41
3.3 Cache Decay: Implementation 44
3.4 Power Evaluation Methodology 47
3.5 Simulation Model Parameters 51
3.6 Cache Decay: Simulation Results 51
3.7 Cache Decay: Adaptive Variants 55
3.8 Changes in the Generational Behavior and Decay 60
3.8.1 Sensitivity to Cache Size, Associativity and Block Size 60
3.8.2 Instruction Cache 61
3.8.3 Multiple Levels of Cache Hierarchy 61
3.8.4 Multiprogramming 64
3.9 Related Work 65
3.10 Chapter Summary 66

4 Timekeeping Prefetch **69**
4.1 Prefetching: Problem Overview 70
4.2 Tag-History-Based Address Predictor 71
4.3 Live-Time-Based Dead Block Predictor 74
4.4 Timekeeping Prefetch: Implementation 76
4.5 Simulation Results 80

4.6 Related Work 84
4.7 Chapter Summary 85

5 Timekeeping in Branch Predictors 87
5.1 Introduction 87
5.2 Branch Predictors Studied 89
5.3 Simulation Model Parameters 91
5.4 Spatial and Temporal Locality in Branch Predictors 92
5.5 Decay with Basic Branch Predictors 94
5.6 Decay with Hybrid Predictors 96
5.7 Branch Predictor Decay with Quasi-Static 4T Cells 98
5.7.1 The Quasi-Static 4T Cell 100
5.7.2 Retention Times In 4T Cells 101
5.7.3 Locality Considerations 102
5.7.4 Results for Decay Based on 4T Cells 103
5.8 Chapter Summary 106

6 Conclusions 108
6.1 Contributions 108
6.2 Future Directions 111
6.3 Chapter Summary 113

Chapter 1

Introduction

1.1 Introduction

For several decades now, the advance of semiconductor technologies has enabled a doubling of transistor density on a manufactured die every year [53]. This trend, widely known as "Moore's law", provided computer architects with solid physical basis for building high performance computers.

The task of a computer architect is a complex one: to "determine what attributes are important for a new machine, then design a machine to maximize performance while staying within cost constraints" [25]. In other words, a computer architect must understand the functional requirements of software applications, organize transistors into a complete machine to satisfy these functional requirements, and then optimize the design for performance, cost and other factors.

With accumulated experience over the years, as well as the help from computers themselves (through computer aided design), computer architectures have evolved from simple sequential designs in early years to complex super-pipelined, superscalar, out-of-order designs such as the Intel Pentium 4 processors [28]. When designing such powerful and

complicated processors, computer architects face enormous challenges. Among these challenges, the "memory wall" and the increasing power consumption and density problems stand out as crucial bottlenecks awaiting further breakthroughs:

- **The "memory wall"**: The rate of improvement in microprocessor speed far exceeds that in DRAM memory speed [25]. This is illustrated in Figure 1.1: while CPU speeds double approximately every eighteen months, main memory speeds double only about every ten years. Because of these diverging rates, a memory request in current microprocessors could take hundreds of cycles to complete. This often leaves the CPU staying idle in waiting of the data, and thus greatly limits the system performance. To solve this problem, on-chip caches have long been used as high speed buffers for the main memory [68]. However, current cache designs suffer from cache misses and their performance is far from optimal. Overall, the "memory wall" is still a unsolved problem for computer architects [49, 79].

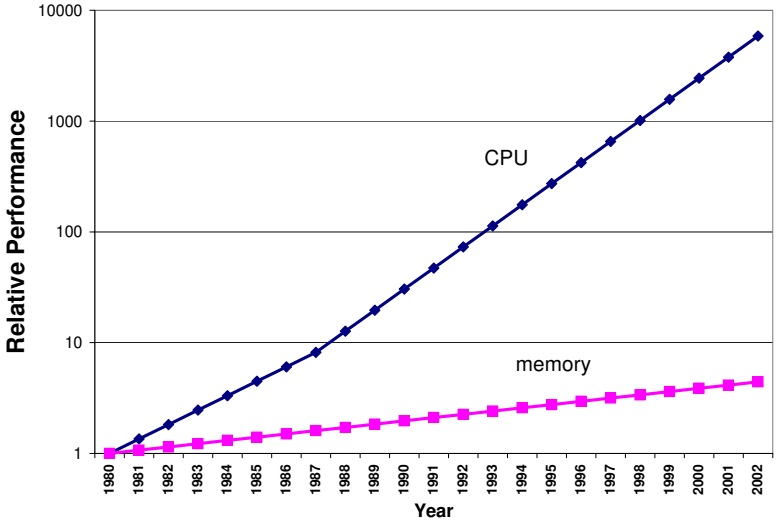

Figure 1.1: The widening speed gap between microprocessor and main memory, starting with 1980 performance as a baseline. The memory baseline is 64-KB DRAM. The CPU refers to the whole microprocessor including on-chip caches. Data are based on [25].

- **Power consumption**: As shown in Figure 1.2, as CPU chips are more densely packed with transistors, and as clock frequencies increase, power dissipation on modern

CPUs will keep increasing in the following generations. Although power has traditionally been a worry mainly for mobile and portable devices, it is now becoming a concern in even the desktop and server domains.

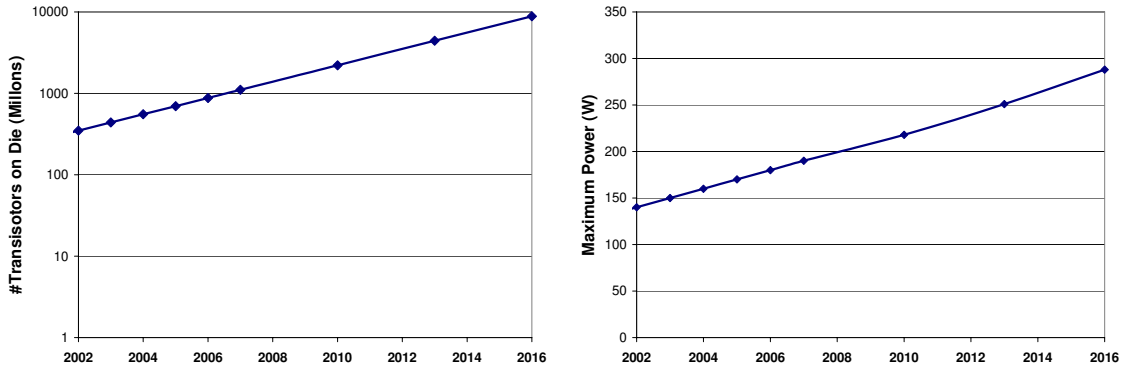

Figure 1.2: Increasing transistors on die (left) and corresponding increases in power dissipation (right). Data are based on the International Technology Roadmap for Semiconductors, 2001 Edition [66].

In current CMOS circuits, the dominant form of power dissipation is "dynamic" or "switching" power, which arises from the repeated charging and discharging of transistors and wires due to computing activity. Dynamic power is proportional to the square of the supply voltage; for that reason, it has been common to reduce supply voltage to improve both performance and power. The left graph in Figure 1.3 demonstrates this trend. While effective, this optimization often has the side effect of increasing the amount of "static" or "leakage" power that a CMOS circuit dissipates, which is mainly due to the sub-threshold current that flows through transistors even when they are not switching. Static power is so-named because it is dissipated constantly, not simply on wire transitions. Static power is a function of the circuit area, the fabrication technology, and the circuit design style. In current chips, static power represents about 2-5% of power dissipation (or even higher [35]), but it is expected to grow exponentially in upcoming generations [5, 66], as shown by the right graph in Figure 1.3. With a trend of growing impact, leakage power has become a major

issue for power-aware computer architects.

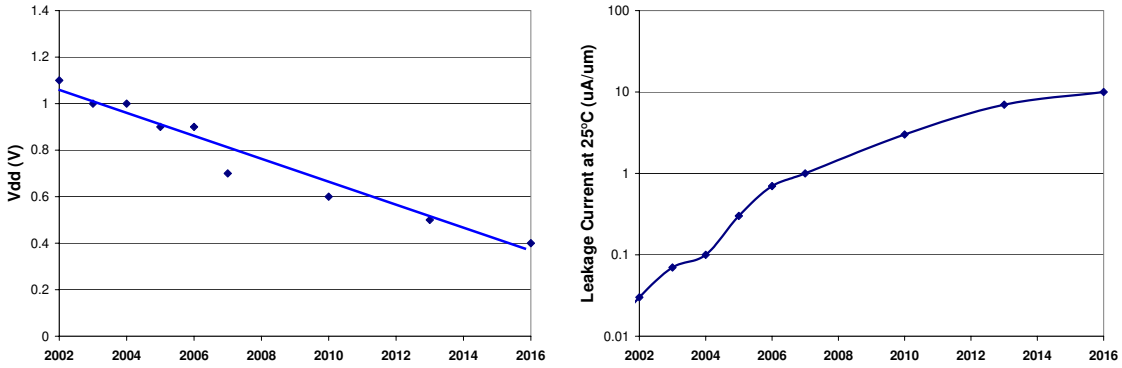

Figure 1.3: Decreasing supply voltage (Vdd) (left) and exponential increases in leakage current (right). Data are based on the International Technology Roadmap for Semiconductors, 2001 Edition [66]. The left graph is in linear scale while the right graph is in log scale.

In attacking these challenges, many computer architects adopt a quantitative approach as described below [25]:

1. First, identify certain aspects of program behavior that are crucial to performance, power or other targets under attack.

2. Second, profile and observe the chosen aspects of program behavior to locate the "common cases" that most frequently occur.

3. Third, design specific hardware structures to cater to the common cases and make them the most effective in terms of the set target.

In this process, the first step is a starting point, limiting the scope of the following steps. Since there are numerous aspects of program behavior, attempt to cover all aspects of program behavior is not practical. On the other hand, ignoring important aspects of program behavior could miss great opportunities for improving processor performance. Therefore, it is important to pinpoint appropriate aspects of program behavior to investigate.

Most prior work has focused on time-independent aspects of program activity. In these approaches, event ordering and interleaving are of prime importance. An example of such work is the trace-based analysis adopted by most researchers [74, 78, 29]. In trace-based analysis, program memory behavior is typically represented as collections of memory address traces. Memory addresses and their access ordering are captured in these traces, but the time intervals between accesses are missing, therefore can not be exploited.

1.2 Contributions

In contrast to traditional trace-based analysis, in this thesis we propose to keep track of the time intervals dynamically at run time, use them to make predictions about future program behavior, and build hardware mechanisms to exploit these predictions for improving processor performance and power consumption. This methodology, dubbed the "timekeeping methodology", exploits statistical characteristics in the time-dependent aspects of program behavior, as well as those that are time-independent. As an example, one characteristic of a program's time-dependent behavior is that data tend to become useless after they have been idle for a long time. To exploit this characteristic, we can keep track of the idle time (the time interval between current time and the time of the previous access) of a cache line, predict a cache line useless if the idle time exceeds a threshold, and then "turn off" the cache line to cut off its leakage power consumption. This mechanism, called "cache decay", provides substantial power savings while maintaining processor performance. Using this example, as well as several other sample applications, this thesis shows that time-based tracking of program behavior can be a powerful way of understanding and improving future program behavior.

The timekeeping methodology can be widely applied to various components in digital processors. We will first use the memory system as the main example to illustrate the power

of the timekeeping methodology. Later we will give examples of how the timekeeping methodology can be applied to other structures, such as branch predictors.

1.2.1 Timekeeping in the Memory System

We follow a three-step process to apply the timekeeping methodology to the memory system:

1. First, we provide a complete quantitative characterization of the time-dependent memory reference behavior. We construct an expanded set of useful metrics regarding generational behavior of cache lines.

2. Second, using these metrics, we introduce a fundamentally different approach for on-the-fly categorization of application reference patterns. We give reliable predictors of conflict misses, dead blocks and other key aspects of reference behavior.

3. Third, based on our ability to discover these reference patterns on-the-fly, we propose hardware structures that exploit this knowledge to improve performance and power consumption.

We propose three novel hardware mechanisms as applications of timekeeping in the memory system to improve processor power and performance.

- Cache Decay [33, 42, 43]: Cache decay targets cache lines with long idle times for reducing cache leakage energy. With long idle times, these cache lines consume the largest portion of cache leakage energy. If a long idle time can be identified at run time, then the associated cache line can be "turned off" in time to cut off leakage consumption. A key program characteristic that supports cache decay is that if a cache line has been idle for a long time, it is likely no longer useful, so turning it off will not incur any extra cache miss. Cache decay can be implemented using simple

hierarchical counters, with a two-bit counter for each cache line, and a single global counter for the whole cache. Simulation results show that cache decay can reduce cache leakage energy by $4X$, with minimal impact on miss rate or performance.

- Timekeeping Victim Cache Filter [34]: In contrast to cache decay, the timekeeping victim cache filter targets cache lines with short idle times. Short idle times indicate cache lines that are likely victims of conflict misses, which are prematurely evicted due to mapping conflicts. Consequently, these cache lines make perfect candidates for conflict-oriented structures, such as victim caches. Simulation results for the SPEC2000 benchmarks prove the effectiveness of such filters: the victim cache traffic is reduced by about 87%, while at the same time better performance is achieved compared to victim caches without filters. Compared to another filter [1] built upon time-independent analysis, the timekeeping filter achieves 22% more performance improvement with a similar effect on victim cache traffic.

- Timekeeping Prefetch [34]: The timekeeping prefetch mechanism demonstrates that a history-based predictor can predict both what should be prefetched and when to initiate the prefetch. The intuition is as follows: in a cache set, if in the past cache line A was followed by B, then C, and the live time (the time between the first and the last access) of B is lt(B), then the next time we see A followed by B, we can predict C as the next address and lt(B) as the live time of B. Thus a prefetch to C can be scheduled at roughly lt(B) after the appearance of B. In other words, the history of previous occurrences can predict the behavior of the current occurrence. Our simulation results show that the resulting prefetcher can improve processor performance by an average of 11% for the whole SPEC2000 benchmark suite, with an 8 KB correlation table. For 6 out of the 26 benchmarks, it improves performance by more than 20%. With a storage requirement that is 2-3 orders of magnitude smaller,

this timekeeping prefetcher outperforms previously proposed prefetchers exploiting time-independent memory behavior.

While cache decay targets cache lines with long idle times, timekeeping victim cache filter mainly concerns those with short idle times. Both cache decay and timekeeping victim cache filter exploit lifetime characteristics within single occurrences. In contrast to these two mechanisms, timekeeping prefetch demonstrate how regularity across consecutive occurrences of the same cache line can be exploited.

Different timekeeping techniques require specific hardware support. A common requirement in all timekeeping techniques is to track time intervals at run time. Our proposal is to gauge the intervals by hierarchical counters. Each cache line is augmented with a local counter that is 2-5 bits wide. This only adds 1% - 2% extra hardware to a typical cache line. All local counters are triggered by a global cycle counter, which is shared by the whole processor. The local counters gauge time intervals dynamically, at the granularity determined by the global counter. Hierarchical counters are more hardware-efficient than a flat design that uses full counters in each cache line. Moreover, the shared global counter provides a central knob that can be dynamically turned to adapt to varying run time program behavior.

1.2.2 Timekeeping in Other Systems

Aside from the memory system, we also show how to apply the timekeeping methodology to other structures, such as branch predictors [30, 31, 32]. With large array structures similar to caches, branch predictors are natural candidates for applying cache decay strategies. A direct application of decay to branch predictors, in the granularity of a row in the predictor array, can reduce the branch predictor leakage energy by 40-60% with minimal impact on either the predictor accuracy or the program performance. More interestingly, because branch predictor contents are often short-lived, and not critical to execution correctness,

we propose to store them with quasi-static 4-transistor (4T) cells, instead of traditional 6-transistor (6T) RAM cells. 4T cells are generally smaller than 6T cells, and naturally decay over time so no extra control is required to activate decay. Overall, 4T-based branch predictors offer up to 33% in cell area savings and 60-80% leakage savings with minimal performance impact. More importantly, the match between data characteristics of branch predictor contents and 4T cells suggests a new thinking of how transient data should be supported in power-aware processors.

The techniques presented in this thesis are highly effective and interesting by themselves. As applications of the timekeeping methodology, they further demonstrate the effectiveness of this novel methodology. We expect that in our future work, as well as in work by other researchers, more timekeeping techniques can be proposed to help future processors to meet the many challenges in power and performance.

1.3 Organization

Chapter 2 gives an overview of the timekeeping methodology, introduces the basic concepts for timekeeping in the memory system, and presents a timekeeping mechanism to improve the management of victim caches. Chapter 3 describes another timekeeping mechanism that provides cache leakage energy savings while maintaining processor performance. Chapter 4 proposes a novel hardware prefetecher which exploits regularity between the past and present cache line lifetime behavior. Chapter 5 extends the timekeeping methodology to structures other than caches, such as branch predictors. Finally, Chapter 6 offers our conclusion.

Chapter 2

The Timekeeping Methodology

In this chapter we first give an introduction to the timekeeping methodology in Section 2.1. Later in Section 2.2 we use the memory system as the main example to further illustrate the timekeeping methodology, Section 2.3 then presents a sample application of timekeeping in the memory system to improve victim cache efficiency. Section 2.4 discusses some related work. Finally, Section 2.5 concludes this chapter.

2.1 The Timekeeping Methodology: Introduction

Frequently described as the fourth dimension, time is a key element in real life. From early sundials to mechanical clocks to atomic clocks, humans have taken a great effort to measure time accurately. The reason for this is simple: all events are time-dependent, *i.e.*, all events occur within a time frame. Keeping track of time enables humans to know at what stage they currently are and to anticipate or predict what will happen in the future and when.

Very often events exhibit a recurring pattern within the time frame. Each occurrence is somewhat similar to the previous occurrence, but with possible changes. Each occurrence

is defined as a "generation". Within each generation, objects go through a period of active life, and then followed by a period of inactive life. It is very important to identify these two periods. For example, the lifetime of milk can be divided into two periods, the valid period when it is safe to drink, and the expired period when it becomes stale and unsafe to drink.

In digital processors events also occur within their time frames, and are usually associated with some information. Data and instructions dominate architectural information in a typical processor. However, with increasingly speculative techniques adopted in current processors, the amount of transient, predictive data in processors has grown greatly. The term "transient, predictive data" refers to the information stored in predictive structures such as branch predictors. Since such data mainly serve as execution hints for improving processor performance, losing them does not affect the correctness of the execution. In Chapter 5 we exploit this characteristics to propose an energy-efficient storage scheme for transient data.

Figure 2.1 shows the instruction flow. Instructions are fetched from the main memory, and pass sequentially through the L2 instruction cache, the L1 instruction cache, the instruction fetch buffer, the reorder buffer, the reservation station, the functional units and then ended in the reorder buffer. In each component, an instruction stays active for a period after initial activation, and then remains inactive for some time until finally getting evicted. After eviction, often the same instruction may re-enter a component sometime later and thus start a new occurrence. Borrowing terminology from the real world, we call each recurrence a "generation". Within each generation, we call the active period the "live time", and the inactive period the "dead time". For example, consider an instruction in the reorder buffer. The lifetime of the instruction starts when it is dispatched into the reorder buffer in program order. It ends when the instruction is committed and removed from the reorder buffer. During the time between dispatch and issue, instructions re-check the availability of their operands whenever a new instruction finishes execution. When they become ready,

i.e., all their operands are available, they are issued to the reservation station. There, they wait to get free function units, then start execution, and finally write back their results and wakeup dependent instructions in the reorder buffer. Even though instructions complete their execution after writeback, they usually remain in the reorder buffer for a while waiting for previous instructions (in program order) to commit. This ensures the handling of precise interrupts. Overall, we can divide the lifetime of a typical instruction in the reorder buffer into two distinct periods: the active period, between dispatch and writeback, and the inactive period, between writeback and commit. An interesting exception happens when a misprediction is discovered for an earlier branch, or when a fault is detected in an earlier instruction. In this case, the lifetime of the instruction is terminated immediately.

Figure 2.2 depicts the lifetime of an instruction in the reorder buffer.

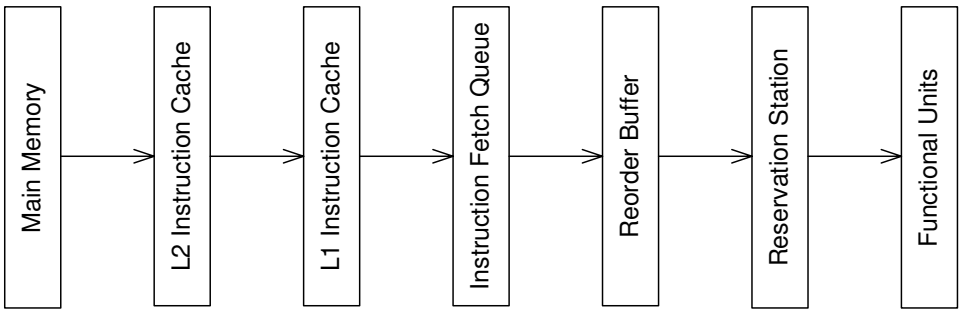

Figure 2.1: The instruction flow in a typical processor.

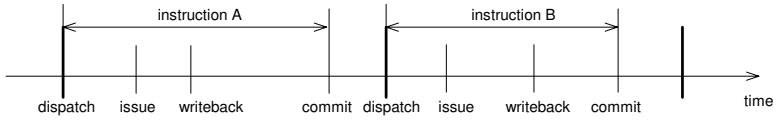

Figure 2.2: Timeline depicting the lifetime of an instruction in the reorder buffer. The instruction is active between dispatch and writeback, and becomes inactive between writeback and commit. After commit, the entry is freed, remains idle for a while, and later is reclaimed by another instruction. Note that "dispatch" refers to the transfer from the instruction fetch queue to the reorder buffer, while "issue" refers to the transfer from the reorder buffer to the reservation station.

Figure 2.3 shows the flow of data, in the time order, through main memory, the L2 data cache, the L1 data cache, the functional units and the register file. Data in these components

exhibit similar lifetime behavior as described above for instructions. For example, consider a cache line of data in the L1 data cache, as shown in Figure 2.4. The lifetime of the cache line begins with a cache miss to the particular cache line, which initiates a load of the cache line into the cache. While staying in the cache, the cache line is first actively accessed for several times, followed by a period with no accesses, and eventually evicted out of the cache to make room for another cache line B.

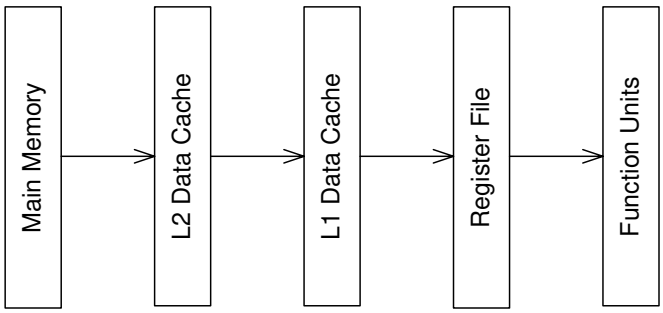

Figure 2.3: The data flow in a typical processor.

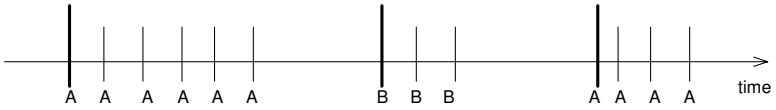

Figure 2.4: Timeline depicting a reference stream in a cache frame. Long heavy ticks represent cache misses, while short light ticks represent cache hits.

Much prior computer architecture research has been based on "orderkeeping" methodologies. In an "orderkeeping" methodology, the ordering of events matters but the exact time intervals between events are ignored. For example, the reorder buffer keeps all instructions in their original program order, but the time intervals, such as the length of active/inactive period for each instruction, and the time between two consecutive instructions, are not tracked. As another example of an "orderkeeping methodology", the memory system behavior is usually represented as "traces" of memory addresses. The reference stream in Figure 2.4 can be represented as $(..., A, A, A, A, A, A, B, B, B, A, A, A, A, ...)$ or simply $(..., A, B, A, ...)$. These traces preserve the appearance order of the addresses, but the time

intervals, such as the time between consecutive accesses to A, or between last A and first B, are missing from the traces, and are not tracked at run time either.

In this thesis, we propose a "timekeeping" methodology, in which the time intervals are tracked and used to classify, predict, and optimize future memory behavior. The key characteristic of the timekeeping methodology, compared to traditional orderkeeping or trace-based methodology, is that time intervals are tracked and exploited. The ultimate goal of timekeeping is the following. Imagine that execution is at some arbitrary point along the timeline depicted in Figure 2.4. We can know something about past history along that timeline, but we wish to predict what is likely to happen soon in the future. First, we wish to deduce *where we currently are*. That is, is the current cache line still active, or will it be evicted next? This cannot be perfectly known until the next event, but if we can predict it accurately, we can build power and performance optimizations based on this prediction. Second, we wish to deduce *what will happen next*: a re-reference of the cached data? Or a reference to new data that begins a new generation? Accurately deducing what will be referenced next *and when* is a crux issue for improving processor performance.

Let us look at some examples to illustrate why the time intervals are useful. If we consider the data in an L1 data cache line, we can deduce the following rules:

- If the common case is that a cache line becomes useless after a period of X cycles, then instead of always waiting until next access, a cache line can be proactively evicted after X cycles. This is similar to how we handle milk in the real life: if we know milk will typically expire 2 weeks after buying, we would stop drinking it after that time, and discard it to make room in the refrigerator for new milk.

- If in the previous occurrence, a cache line became useless after Y cycles, then in the current occurrence we also expect it to become useless after Y cycles. Compared to the previous rule, this rule emphasizes learning from individual history. In real life,

the milk expiration time depends heavily on the brand of the milk and the particular refrigerator, so knowing the expiration time of the milk bought last time could lead to a more accurate estimate of the expiration time.

From this example we can see why the timekeeping methodology works: it is because the time intervals can help us to deduce what will happen next. The key characteristic exploited by the timekeeping methodology is the correlation between the time intervals and future program behavior, which can be observed either by investigating the common cases, or by resorting to past history. Hardware structures can be built to exploit the correlation to optimize processor behavior. Figure 2.5 depicts the stages one goes through in using the time intervals to build mechanisms, This is the general work flow of applying the timekeeping methodology.

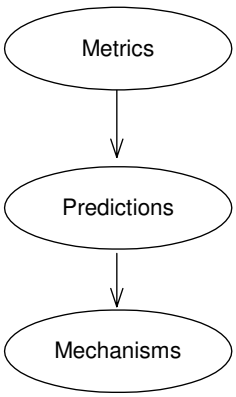

Figure 2.5: The general work flow for timekeeping techniques.

- **Metrics**: The first step in applying the timekeeping methodology is to investigate the metrics, which are the time intervals between events. For instructions in the reorder buffer (See Figure 2.2), the metrics include the time intervals between dispatch and issue, between issue and writeback, between writeback and commit, etc. For data in caches (See Figure 2.4), the metrics include the intervals between consecutive access to the same cache line, between the first access and the last access, between the last

access and the eviction, etc. These time intervals characterize the time-dependent aspects of processor behavior; therefore they are of prime interest to the timekeeping methodology.

- **Predictions**: The key premise of the timekeeping methodology is that the timekeeping metrics can be used to identify *predictions* or *indications* about future program behavior. For instructions, the predictions might include whether an instruction is performance-critical, whether it is speculative, etc. For data, the predictions include identifying conflict misses or deducing dead cache blocks, etc. Interestingly, as shown later in this chapter, sometimes more than one metric can be used as a predictor of the same behavior, each with a different trade-off between cost and effectiveness.

- **Mechanisms**: To exploit these predictions, hardware structures can be added to track the timekeeping metrics at run time, and use them to activate performance-enhancing or power-reducing schemes. For example, one characteristic of processor behavior is that often when data stay idle for a long time, they become useless and thus can be "turned off" to cut off leakage consumption. To exploit this characteristic, counters can be added to gauge the data idle time. When the idle time exceed a threshold, the counters can signal to "turn off" the data, thus reducing leakage power consumption.

The timekeeping methodology can be widely applied to the whole processor. We will use the memory system as the major example to illustrate the power of the timekeeping methodology. In the next section, we first introduce some basic metrics in the memory system, then we present statistical distributions of the metrics, and finally we demonstrate how the metrics can be used to make predictions about processor memory behavior. To illustrate the effectiveness of such predictions, Section 2.3 describes a "timekeeping victim cache filter", which tracks a timekeeping metric at run time to predict whether a cache line

is a victim of conflict misses, and improves the efficiency of the victim cache based on this prediction. Chapters 3 and 4 present two more examples of timekeeping in the memory system: cache decay and timekeeping prefetch. Timekeeping in other subsystems will be discussed in Chapter 5.

2.2 Timekeeping in the Memory System

This section gives an overview of how the timekeeping methodology can be applied to the memory system. We start by introducing and profiling the basic timekeeping metrics in the memory system, and then show how they can be used to predict memory system behavior.

2.2.1 Introduction to Timekeeping Metrics in the Memory System

Figure 2.6 depicts a stream of references to a particular cache line. One can break this reference stream into *generations*. Each generation is comprised of a series of references to the cache line. Using the terminology from [77], the i-th generation begins immediately after the i-th miss to that cache line, when a new memory line is brought into the cache frame. This generation ends when the line is replaced and a new one is brought into the cache frame. Generations begin with one or more cache references. Following the last reference before eviction, the generation is said to have entered its dead time. At this point, this generation has no further successful use of the items in the cache line, so the line is said to be dead. Each cache line generation is divided into two parts: the *live time* of the cache line, where the line is actively accessed by the processor and the *dead time*, awaiting eviction.

The live time of a cache line starts with the miss that brings the data into the cache and ends with the last successful access before the item is evicted. The dead time is defined as the time duration where the cached data will not be used again successfully *during this*

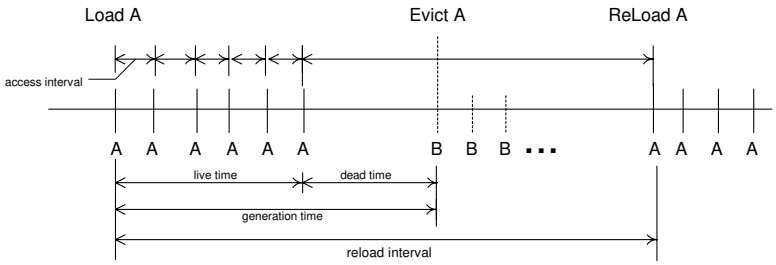

Figure 2.6: Timeline depicting the reference stream to a cache frame. First, A is resident, followed by A's eviction to begin a generation with B resident, and eventually, A is re-referenced to begin yet another generation.

generation. That is, the dead time is the time between the last access for a generation and when the data are actually evicted. Many times we see only a single miss and then eviction. We consider these cases to have zero live time; here the generation length is equal to the dead time.

There are further metrics that also turn out to be of practical interest. These include the access interval and reload interval. Access interval refers to the time interval between successive accesses to the same cache line *within the live time of a generation*. In contrast, reload interval denote the time duration between the beginnings of two generations that involve the same data in the same memory line. The reload interval in one level of the hierarchy (eg, L1) is actually the access interval for the cache in the next lower level of the hierarchy (eg, L2) assuming the data are resident there.

To further understand the basic metrics, we can relate them to program memory behavior. The correlation between the metrics and program behavior is the key characteristic that will be exploited by the timekeeping methodology.

1. Live Times vs. Dead Times: Cache lines are in two distinct states depending on whether they are currently in live time or dead time. In live time, cache lines are active and expecting more reuse before eviction, so they must be kept valid. On the other hand, when in dead time, cache lines will not be used before eviction, therefore their contents can be safely discarded (although they need to be written back if dirty.)

When dead, a cache line can be either turned off to cut off leakage, as in the cache decay mechanism (Chapter 3,) or be reclaimed to store prefetched data predicted to be useful in the future, as in timekeeping prefetch (Chapter 4).

2. Access Intervals vs. Dead Times: Both access intervals and dead times are inter-access intervals between consecutive accesses to the same cache frame. The difference is, for access interval, the accesses are targeting the same cache line, while for dead time, the accesses are targeting different cache lines. Imagine there is a timer which constantly gauges inter-access intervals in a cache frame. At any point of time, the timer is either counting an access interval, or a dead time, but never both. It is only when the next access appears that we exactly know what the timer has been counting. However, in the interim, the duration elapsed hints the end result: If the duration is long enough, it is very likely that a dead time is being counted, but not an access interval. This observation will be exploited in the cache decay mechanism, detailed in Chapter 3.

3. Access Intervals vs. Reload Intervals: Access interval represents the time between consecutive *accesses* to the same cache line. Reload interval is the time between consecutive *misses* to the same cache line. In a first glimpse, they are very different metrics. However, with hierarchical cache design, misses to one cache level usually turn out to be hits in the next cache level. Consequently, for data resident in the L2 cache, a reload interval in the L1 cache corresponds to the access interval of the same data in the L2 cache, down one level in the hierarchy.

2.2.2 Simulation Model Parameters

In the next section, we will show the statistical distributions of the basic metrics introduced in the previous section. Unless stated otherwise, our results in this chapter, as well as for

the rest of this thesis, use SimpleScalar [8] to model an aggressive 8-issue out-of-order processor with the configuration parameters shown in Table 2.1.

Processor Core	
Clock rate	2GHZ
Instruction Window	128-RUU, 128-LSQ
Issue width	8 instructions per cycle
Functional Units	8 IntALU,3 IntMult/Div, 6 FPALU,2 FPMult/Div, 4 Load/Store Units
Memory Hierarchy	
L1 Dcache Size	32KB, 1-way, 32B blocks, 64 MSHRs
L1 Icache Size	32KB, 4-way, 32B blocks
L1/L2 bus	32-bit wide, 2GHZ
L2 I/D	each 1MB, 4-way LRU, 64B blocks,12-cycle latency
L2/Memory bus	64-bit wide, 400MHZ
Memory Latency	70 cycles
Prefetcher	
Prefetch MSHRs	32
Prefetch Request Queue	128 entries

Table 2.1: Configuration of simulated processor.

We evaluate our results using the SPEC CPU2000 benchmark suite [73]. The benchmarks are compiled for the Alpha instruction set using the Compaq Alpha compiler with SPEC *peak* settings. For each program, we skip the first 1 billion instructions to avoid unrepresentative behavior at the beginning of the program's execution. We then simulate 2 billion instructions using the reference input set. We include some overview statistics here for background. Figure 2.7 shows how much the performance (IPC) of each benchmark would improve if *all* conflict and capacity misses in the L1 data cache could be eliminated. This is the target we aim for in our memory optimizations. The programs are sorted from left to right according to the amount they would speed up if conflict and capacity misses could be removed.

Figure 2.8 breaks down the misses of these programs (with an L1 cache configured as in Table 2.1) into three stacked bar segments denoting cold, conflict and capacity misses.

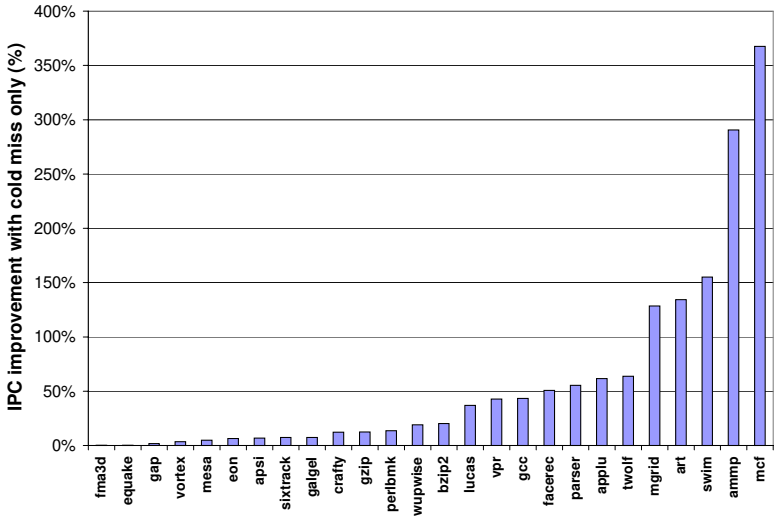

Figure 2.7: Potential IPC improvement if all conflict and capacity misses in the L1 data cache could be eliminated for SPEC2000 benchmarks.

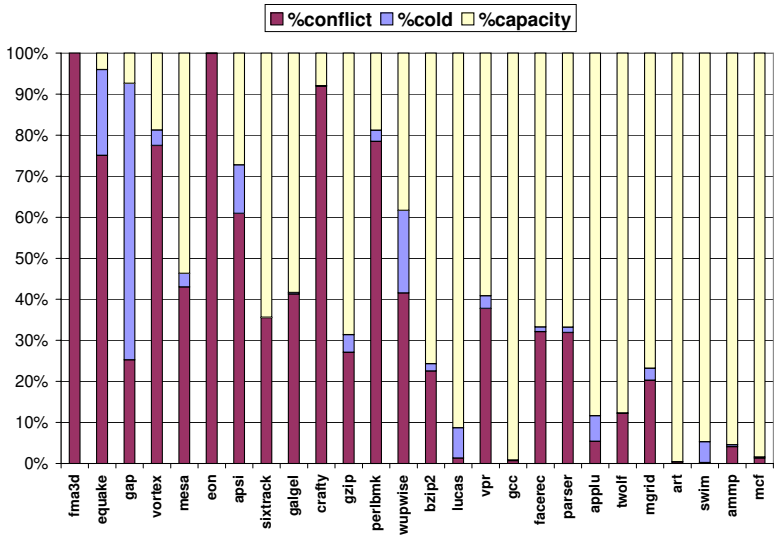

Figure 2.8: Breakdown of program L1 data cache misses into three categories: conflict, cold and capacity. Data are obtained with a 32KB direct-mapped cache, as shown in Table 2.1.

An interesting observation here is that the programs that exhibit the biggest potential for improvement (RHS of Figure 2.8) also tend to have comparatively more capacity misses than conflict misses. Thus, we expect that eliminating capacity misses will result in larger benefit than eliminating conflict misses. This is confirmed in Chapter 4.

2.2.3 Statistical Distributions of Timekeeping Metrics

The upper two graphs in Figure 2.9 illustrates a distribution for SPEC2000 benchmarks of *live times* and *dead times*. Recall that live time is defined as the time duration for a cached item between when it arrives in cache, and when it experiences its last successful use before its eviction. Dead time is defined as the time duration between when an item in the cache is last used successfully, and when it is evicted from the cache.

The lower two graphs in Figure 2.9 illustrates access interval and reload interval distributions. Reload intervals are plotted with the x-axis times 1000 cycles rather than 100X as in previous graphs. Access interval is the time duration between successive references within a cache line live time. In contrast, reload interval is the time between the beginnings of two successive generations involving the same memory line.

Dead times are in general much longer than average access intervals. For example, over all of the SPEC suite, 61% of access intervals are 100 cycles or less. In contrast, only 31% of dead times are less than 100 cycles. On the other hand, 21% of dead times are more than 10,000 cycles, while only 2% access intervals are 10,000 cycles or more. This is a useful observation because it hints that we can succeed in discerning dead times versus access intervals *a priori* based on the durations observed. Chapter 3 will exploit this observation.

Another observation from the distributions of the basic metrics is that for dead times and reload intervals, the distributions are "bi-modal", *i.e.*, the distributions are heavy at heads and tails, while light in the middle. This naturally hints two distinct categories of cache lines: those with long dead times (or reload intervals), versus those with short dead

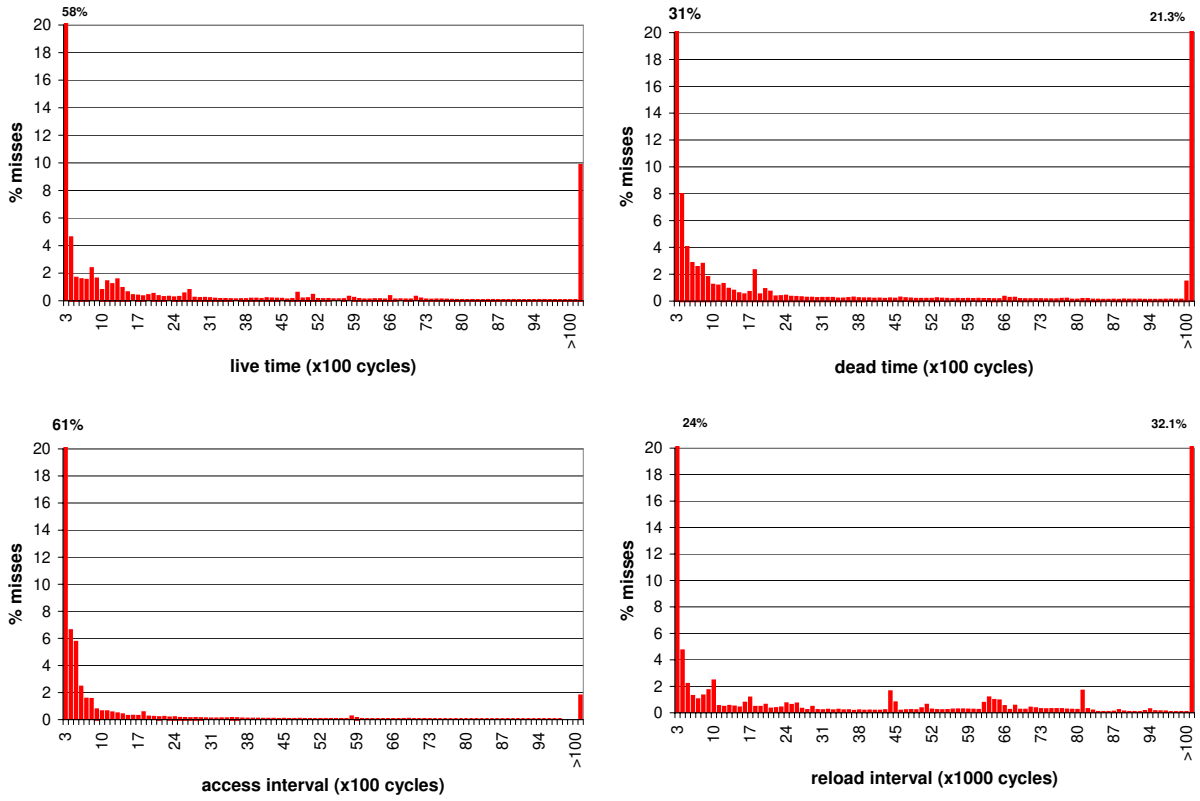

Figure 2.9: Distribution of live times (upper left), dead times (upper right), access intervals (lower left), and reload intervals (lower right) for all generations of cache lines in the SPEC2000 simulations. Numbers marked at the head and the tail of each distribution represent the percentage of metrics that falls into that range.

times (or reload intervals). We discuss this categorization in detail in the next section.

2.2.4 Using Timekeeping Metrics to Predict Conflict Misses

When a cache line A is evicted, there are two possible reasons: (1) there is a mapping conflict between A and another cache line B, or (2) there is not enough space so A must be evicted to make room for B. Based on Hill's definitions in [27], the exact reason will be known by investigating the reload of A. This reload is a cache miss in a direct-mapped cache. However, if we substitute the direct-mapped cache with a fully-associative cache of the same capacity, two situations could occur:

1. If the reload does not miss in the fully-associative cache, then the reason it misses

in the original direct-mapped cache is the mapping conflict between A and B, which is not tolerated by the limited associativity of a direct-mapped cache, but helped by the fully-associative cache. In this case, the reload of A is a "conflict miss" in the original direct-mapped cache.

2. If the reload still misses in the fully-associative cache, then the reason it misses in the direct-mapped cache is a lack of space. In this case, the reload of A is classified as a "capacity miss" in the original cache.

Interpreting Hill's definitions with generational behavior, a conflict miss occurs because its last generation was unexpectedly interrupted—something that would have not happened in a fully associative cache. Similarly, a capacity miss occurs because its last generation was ended because of lack of space—again, something that would not have happened in a larger cache. In this section we evaluate prediction of the miss types with timekeeping metrics. When we correlate metrics to a miss type we always refer to the timekeeping metrics of the last generation of the cache line that suffers the miss. In other words, we use what happens to the current generation of a cache line to predict the miss type of the next miss to the same cache line.

By Reload Interval While Figure 2.9 showed reload intervals over all generations, Figure 2.10 splits the reload interval distribution into two graphs for different miss types. These statistics show vividly different behavior for conflict and capacity misses. In particular, reload intervals for capacity misses are overwhelmingly in the tail of the distribution. In contrast, reload intervals for conflict misses tend to be fairly small: an average of roughly 8000 cycles. The average reload interval for a capacity miss is one to two orders of magnitude larger than that for a conflict miss! Large reload intervals for capacity misses make sense: for an access to an item to be classified a capacity miss, there must be at least 1024 (total number of blocks in the simulated cache) unique accesses to drive the item out of a

fully-associative cache after its last access. In a processor that typically issues 1-2 memory accesses per cycle, the time for 1024 accesses is on the order of a thousand cycles, and it may take even longer before 1024 *unique* lines have been accessed. On the contrary, a conflict miss can have less than 1024 unique cache accesses after their last access; this leads to their small reload intervals.

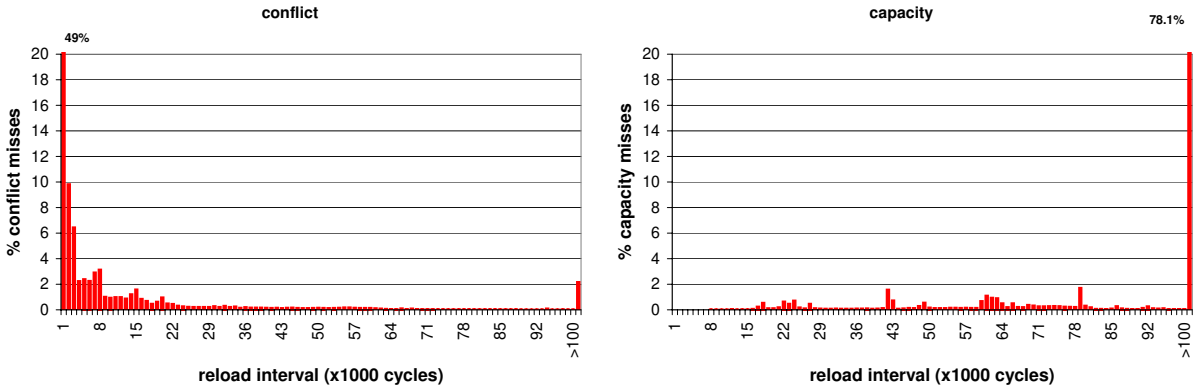

Figure 2.10: Distribution of reload interval for conflict (left) and capacity (right) misses

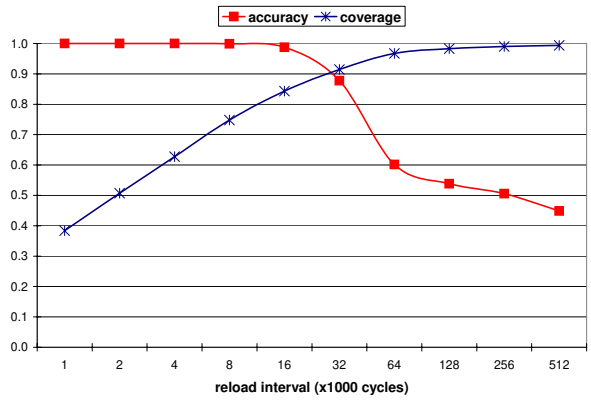

Figure 2.11: Accuracy and coverage for conflict miss predictions based on *reload interval*. Each data point indicates what the accuracy or coverage would be for predicting conflict misses to be all instances where the *reload interval* is less than the quantity on the x-axis.

Reload intervals make excellent predictors of conflict misses. Figure 2.11 shows accuracy and coverage when reload interval is used as predictor. For each point on the x-axis, one curve gives the accuracy of predicting that reload intervals less than that x-axis value denote conflict misses. The other curve gives the coverage of that predictor: i.e., how

often it makes a prediction. For example, a prediction that "if the reload interval is less than 32Kcycles, then the next miss of the same cache line will be a conflict miss" has an accuracy of about 91%, and covers about 88% of all conflict misses.

When conflict misses are defined as small reload intervals (about 1000 cycles or less) prediction accuracy is close to perfect. Coverage, the percent of conflict misses captured by the prediction, is low at that point, however, about 40%. The importance of reload interval, though, shows in the behavior of this predictor as we increase the threshold: up to 16K cycles, accuracy is stable and nearly perfect, while coverage increases to about 85%. This is appealing for selecting an operating point because it means we can walk out along the accuracy curve to 16K cycles before accuracy sees any substantive drop. The clear drop there makes that a natural breakpoint for setting up a conflict predictor based on reload intervals smaller than 16K.

By Dead Time Figure 2.12 shows the distribution of dead time divided by miss types. Again, we see trends similar to reload interval distribution, though not as clear cut. That is, dead times are typically small for conflict misses, while much larger for capacity misses. These observations about dead times hint at a phenomenon one could exploit. Namely, one can deduce that an item has been "prematurely" evicted from the cache due to a conflict miss, if its dead time is quite short. Where dead times are quite large, it hints at the fact that the item probably left the cache at the end of its "natural lifetime"; that is, it was probably evicted as a capacity miss at the end of its usage.

Figure 2.13 shows accuracy and coverage of a predictor that predicts an upcoming conflict miss based on the length of the dead time of the current generation. Namely, for a point on the x-axis, accuracy and coverage data indicate what the prediction outcome would be if one considered dead times less than that value as indicators of conflict misses. Coverage is essentially the fraction of conflict misses for which we make a prediction.

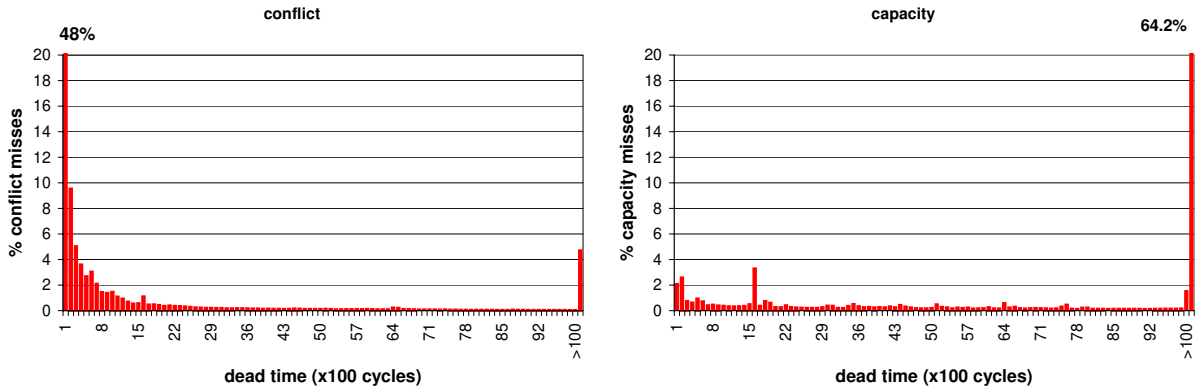

Figure 2.12: Distribution of dead time

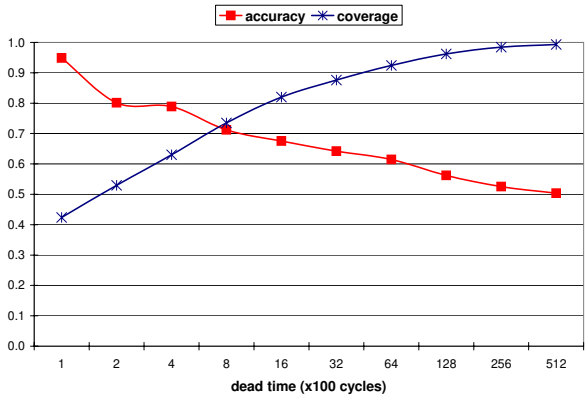

Figure 2.13: Accuracy and coverage for conflict miss predictions based on *dead time*. Each data point indicates what the accuracy or coverage would be for predicting conflict misses to be all instances where the *dead time* is less than the quantity on the x-axis.

Accuracy is the likelihood that our prediction is correct, for the instances where we do make a prediction.

Predicting a conflict miss if the dead time of its last generation is smaller than a given threshold is very accurate (over 90%) for small thresholds (100 cycles or less). But coverage is only about 40% (attesting to the fact that most dead times are large). Increasing the dead-time threshold degrades accuracy but increases coverage. A likely method for choosing an appropriate operating point would be to walk down the accuracy curve (i.e., walk out towards larger dead times) until just before accuracy values drop to a point of insufficient accuracy. One can then check that the coverage at this operating point is sufficient for

the predictor's purpose. In the next section, we describe a hardware mechanism that uses dead-time predictions of conflict misses to filter victim cache entries.

By Live Time Live time is also highly biased between conflict (very small live times) and capacity misses (larger live times). A very important special case here is when we have a live time equal to zero. This special case makes for a simple and fairly accurate predictor of conflict misses. In fact, a single ("re-reference") bit in each L1 cache line is all that is needed to distinguish between zero and non-zero live times.

Figure 2.14 shows the accuracy and coverage of such a prediction. Accuracy is very high: for many programs, accuracy is close to one. The geometric mean for all SPEC2000 is 68% accuracy, but coverage is low. Coverage varies from benchmark to benchmark with a geometric mean of roughly 30%. In contrast to the previous approaches, this prediction has no knobs to turn to trade accuracy for coverage. Because of the low coverage and its specialized nature, live-time conflict prediction is likely to be useful in few situations. We include it here mainly to demonstrate how different metrics can classify or predict the same program behavior.

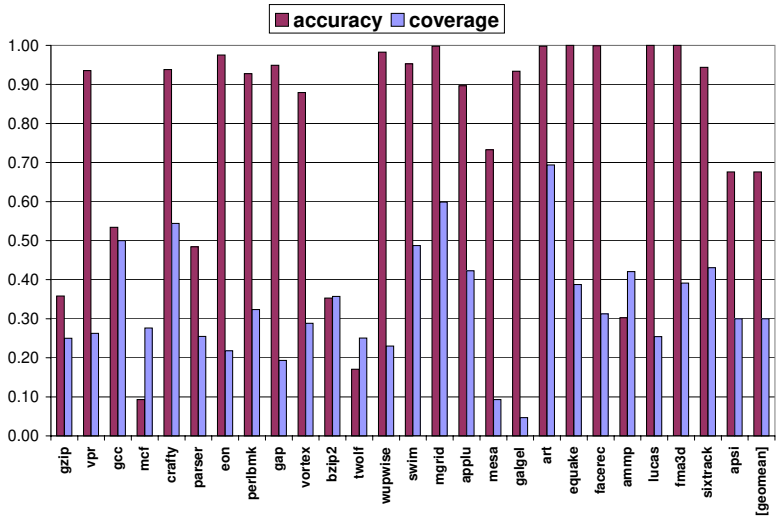

Figure 2.14: Accuracy and coverage using "live time = 0" as a predictor of conflict misses.

Prediction Location Conflict predictors based on dead times (or live times) rely only on L1-centric information. In contrast, conflict predictors based on reload intervals would most likely be implemented by monitoring access intervals in the L2 cache. As a result, one's choice of how to predict conflict misses might depend on whether the structure using the predictor is more conveniently implemented near the L1 or L2 cache.

2.2.5 Summary

In this section, we first introduced the basic metrics for timekeeping in the memory system, including live time, dead time, reload interval and access interval. We profiled the metrics and then showed their statistical distributions. We found that access intervals are overwhelmingly short, while dead times and reload intervals exhibit "bimodal" distributions. In other words, generations with short dead times (and short reload intervals), and those with long dead times (and thus, long reload intervals), are both very common. Our investigation indicates that they represent two distinct categories of cache line generations:

1. Conflict-oriented generations: These cache line generations are ended due to a conflict between the current cache line and its successor. Because they are often ended prematurely, these cache line generations tend to have short dead times and reload intervals, and many of them have a zero live time.

2. Capacity-oriented generations: These cache line generations are ended due to a lack of cache space. These cache lines tend to have long dead times and reload intervals.

Because of the close correlation between the timekeeping metrics and program behavior, these metrics can be dynamically tracked and used to optimize processor behavior. In the next section, we present a timekeeping mechanism that identifies generations with *short* dead times and filters the victim cache traffic by only allowing such cache lines to enter the

victim cache. Later in Chapter 3, another timekeeping mechanism that targets *long* dead times will be discussed.

2.3 Timekeeping Victim Cache Filter

In the previous section we discussed timekeeping metrics that can be used as reliable indicators of conflict misses, particularly dead time and reload interval [1]. Such predictions will naturally facilitate conflict-miss-oriented structures, such as victim caches. In this section, we demonstrate such an example.

2.3.1 Introduction

A victim cache is a small fully-associative cache that reduces L1 miss penalties by holding items evicted due to recent conflicts. Victim caches help with conflict misses, so we propose using conflict indicators such as dead times and reload intervals to manage the victim cache. In particular, we want to avoid entering items into the victim cache that are unlikely to be reused soon.

Small reload intervals are highly correlated to conflict misses and they are an effective filter for a victim cache. The intuition that ties reload intervals to victim caches is the following: Since the size of victim cache is small, a victim block will stay only for a limited time before it is evicted out of the victim cache. In terms of generational behavior, this means that only victim blocks with small reload intervals are likely to hit in the victim cache. Blocks with large reload intervals will probably get evicted before their next access so it is wasteful to put them into the victim cache. Unfortunately, reload intervals are only available for counting in L2, and are not known at the time of eviction. This inhibits their

[1] We do not further examine the zero-live-time predictor because of its relatively low coverage and significant overlap with the technique based on dead time.

use as a means to manage an L1 victim cache.

Besides short reload intervals, short dead times are also very good indicators of conflict misses. Dead times are readily available in L1 at the point of eviction and as such are a natural choice for managing a victim cache associated with the L1. We use a policy in which the victim cache only captures those evicted blocks that have dead times of less than a *threshold* of 1K cycles. Figure 2.12 shows that these blocks are likely to result in conflict misses.

The hardware structure of the dead-time victim filter is shown in Figure 2.15. A single, coarse-grained counter per cache line measures dead time. The counter is reset with every access and advances with global ticks that occur every 512 cycles. Upon a miss the counter contains the time since the last access, i.e., the dead time. An evicted cache line is allowed into the victim cache if its counter value is less than or equal to 1 (giving a range for the dead time from 0 to 1023 cycles).

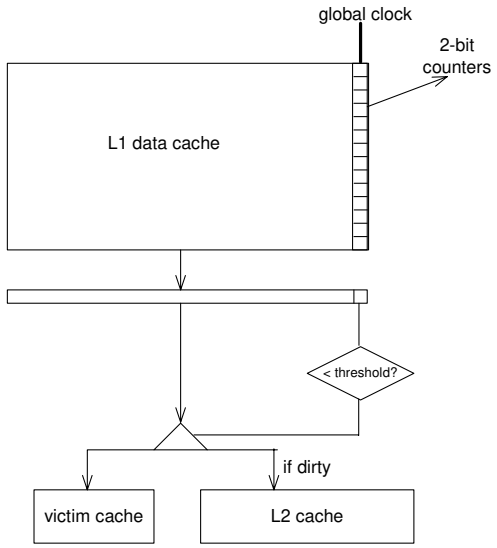

Figure 2.15: Implementation of timekeeping victim cache filter. The global clock ticks every 512 cycles. Upon each global tick, the local counters associated with each cache line increment by one. When a cache line is evicted, if its local counter is less than or equal to 1, it is allowed to enter the victim cache. Otherwise, it will be discarded, after written back to L2 if dirty.

2.3.2 Simulation Results

Our experiments, as shown in Figure 2.16, show that for a 32-entry victim cache managed by the timekeeping filter, the traffic to the victim cache is reduced by 87%. Such a reduction can greatly relieve the traffic pressure on the victim cache, and save power by reducing the number of victim cache fills. Moreover, this reduction is achieved without sacrificing performance, as seen in the top graph of Figure 2.16.

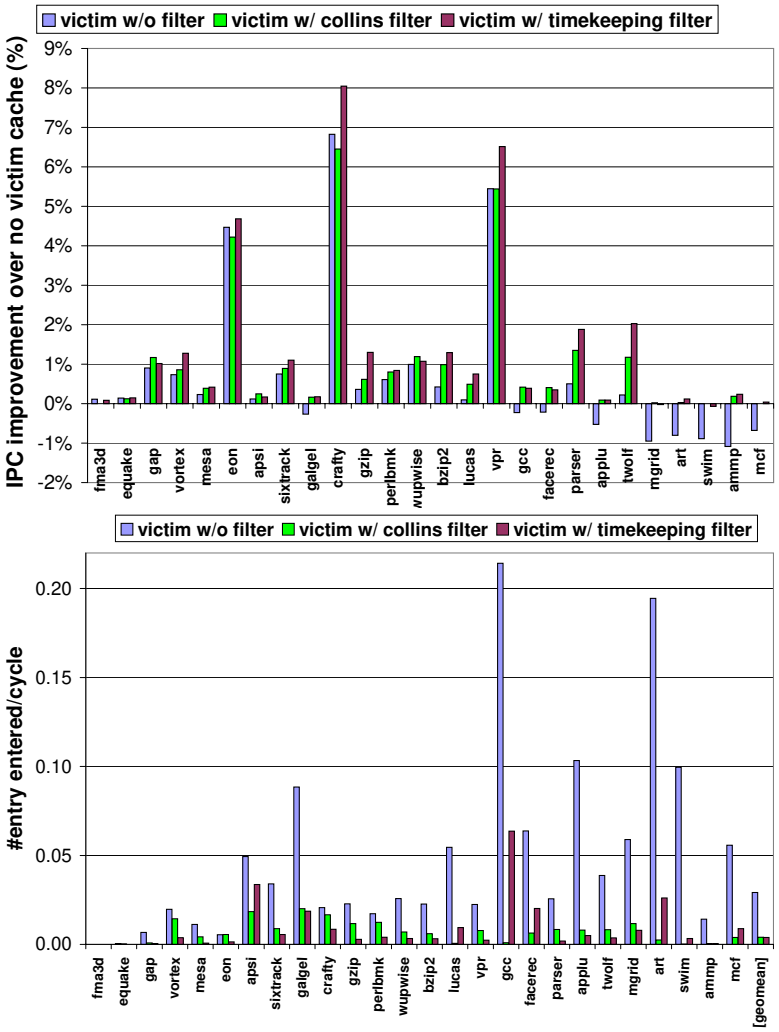

Figure 2.16: Top: IPC improvement offered by timekeeping victim cache filter and a comparison to prior work. Bottom: Fill traffic to victim cache for our method, prior filtering method, and no filtering.

Collins et al. suggest filtering the victim cache traffic by selecting only victims of

possible conflict misses [13]. Their solution requires storing an extra tag for each cache line (remembering what was there before) to distinguish conflict misses from capacity misses. Comparing our approach with a Collins-style filter, we see similar traffic reduction, but our timekeeping victim filter leads to higher IPC for all the benchmarks. The order of the programs in this figure is the same as in Figure 2.7. Recall from Figure 2.7 that the potential for speedup increases to the right, but the ratio of conflict misses to total misses increases to the left. In Figure 2.16, the programs that experience the largest speedups for our timekeeping victim filter are clustered in the middle of the graph. Programs to the far left have little room for improvement. Programs to the far right whose misses are overwhelmingly capacity misses are *negatively* affected with an *unfiltered* victim cache, but they retain their performance if a conflict filter (either Collins-style or timekeeping) is employed.

2.3.3 Adaptive Scheme

The performance of our timekeeping victim filter indicates that the parameters (dead-time threshold, cache sizes, etc.) are well matched. This is not coincidental. What makes our timekeeping techniques invaluable is that they provide a sound framework to reason about such parameters rather than to revert to trial-and-error. We will informally "prove" that the optimal dead-time threshold actually stems from the relative size of the victim cache versus the L1 data cache, and the reuse characteristics of the program. It is essentially a form of Little's Law, from queueing theory [21]. The reasoning is as follows:

1. We can think of a victim cache as providing associativity for some frames of the direct-mapped cache. Without any filtering, associativity is provided on a first-come, first-served basis: every block that is evicted gets an entry in the victim cache.

2. Our timekeeping filtering based on dead time results in a careful selection of the

frames for which the victim cache is allowed to provide associativity. Timekeeping filtering culls out blocks with dead times greater than the threshold. In turn, the dead-time threshold controls the *number* of frames for which associativity is provided for.

3. Our filtering ensures that the victim cache will provide associativity *only* for the "active" blocks that are fairly-recently used at the time of their eviction.

4. Since the victim cache cannot provide associativity to more frames than its entries, the best size of the victim cache relates to the amount of cache in active use. A larger set of "active" blocks dilute the effectiveness of the victim cache associativity. In the data here, with a 1K cycle dead time threshold, only about 3% of cache blocks resident at any moment meet the threshold. Since 3% of 1024 total cache blocks is 30.72, a 32-entry victim cache is a good match.

The relation of the dead-time threshold and the size of the victim cache not only gives us a good policy to statically select an appropriate threshold, but also points to adaptive filtering techniques. An adaptive filtering can adjust the dead time threshold at run-time so the number of candidate blocks remains approximately equal to the number of the entries in the victim cache. With a modest amount of additional hardware an adaptive filter would perform even better than static filter shown above, which already outperforms previous proposals.

2.3.4 Timekeeping Victim Cache Filter: Summary

A typical cache line generation is expected to have a short live time followed by a long dead time. However, conflict misses are "catastrophic" to the typical generation of a block in that they cut its live time or dead time short. A generation resulting from a conflict exhibits either a zero live time or an inordinately short dead time. Furthermore, since the block is thrown out of the cache despite being alive, its reload interval is also very

short, indicating they are likely re-referenced soon. Such blocks make perfect candidates for conflict-oriented structures, such as the victim cache. In this section, we proposed the timekeeping victim cache filter that filters the fill traffic to the victim cache so as to feed it only with blocks evicted with short dead times, which are likely evicted as a result of conflict. Our victim cache filtering results in both performance improvements and significant reductions in victim cache traffic. Our filter outperforms a previous proposal that predicts conflict misses remembering previous tags in the cache.

2.4 Related Work

Puzak introduced the first notion of live and dead cache lines in [60]. He demonstrated the modeling and measurement of the amount of dead lines in the cache and the significance of this parameter to the performance of the system. Mendelson et al. proposed an analytical model for predicting the fraction of live and dead cache lines of a process, as a function of the process' execution time and the behavior of other processes in a multitasking environment [52]. Wood et al. presented the first work to describe and investigate the generational behavior of cache lines [77], and exploit it for improving the accuracy of cache sampling techniques in simulations. They showed that one can deduce the miss rates of unprimed references at the beginning of reference trace samples by considering the proportion of cycles a cache line spends *dead* or waiting to be evicted.

Compared to prior work, this thesis is the first to propose dynamically tracking generational lifetime behavior at run time, and to use them to make predictions of future memory behavior. We give a complete quantitative characterization of generational lifetime behavior, and propose a group of novel hardware mechanisms to dynamically optimize processor performance and power consumption.

2.5 Chapter Summary

In this chapter we first gave an introduction to the timekeeping methodology. While most prior work for improving processor power and performance only exploits time-independent aspects of program behavior, such as event ordering and interleaving, the timekeeping methodology proposes to keep track of the time intervals between events, to use them to make predictions of future program behavior, and then to build hardware mechanisms to harness these predictions. This three-step process, from metrics to predictions to mechanisms, is the general work flow for all timekeeping techniques.

The timekeeping methodology can be applied to all types of information, including data, instructions and transient predictive data, in all components in digital processors. In this thesis, the memory system is used as the main example to illustrate the timekeeping methodology. Timekeeping in the memory system exploits the generational lifetime behavior of cache lines. The basic metrics for timekeeping in the memory system include live time, dead time, access interval, and reload interval. Live time and dead time refer to, respectively, the active and inactive period in a cache line generation. Access interval refers to the interval between consecutive successful accesses to the same cache line. Reload interval is the time between consecutive misses to the same cache line. Statistical distributions of the timekeeping metrics show that: most access intervals are very short, while dead times and reload intervals are "bi-modal", *i.e.*, some of them are short while many others are long. This characteristic reveals the correlation between the program memory behavior and the metrics. More specifically, short dead times and reload intervals are mainly caused by conflict misses, while long dead times and reload intervals are more likely associated with capacity misses. In other words, a short dead time or reload interval indicates that the current cache line generation is ended due to a mapping conflict. Such cache lines makes good candidates for conflict-oriented structures, such as a victim cache. A "timekeeping

victim cache filter", which keeps tracks of dead times using hierarchical counters, and allows a victim to enter the victim cache only when its dead time is small, can reduce the victim cache traffic by 87%, while improving overall system performance.

The timekeeping victim cache filter clearly demonstrates the power of the timekeeping methodology: with simple extra hardware to track the time intervals, the processor performance can be improved along with a power benefit. In the next chapters, we will present several other applications of the timekeeping methodology.

Chapter 3

Cache Decay

In the previous chapter, we gave an overview of the timekeeping methodology, then discussed how it can be applied to the memory system, and finally presented a sample application, the timekeeping victim cache filter, that can reduce victim cache traffic with simple counter hardware. The timekeeping victim cache filter targets cache line generations with short dead times, because they are likely victims of conflict misses and thus fit well to the purpose of the victim cache. In this chapter we demonstrate how the timekeeping methodology can help to control cache leakage energy consumption. We start by evaluating the potential benefits of such a technique.

3.1 Potential Benefits

As described in Chapter 1, leakage energy is increasing exponentially and is becoming a major challenge in designing power-efficient processors. Because caches comprise much of the area in current and future microprocessors, it makes sense to target them when developing leakage-reducing strategies. Recent work by Powell et al. has shown that transistor structures can be devised to limit leakage power by banking the cache and providing "sleep"

transistors which dramatically reduce leakage current by gating off the V_{dd} [59, 80]. Note that this circuit technique is state-destroying, meaning that when "sleep" transistors are enabled the data stored in memory cells will get lost. Therefore, if the data are dirty, they should be written back before entering into sleep state.

Our work exploits these sleep transistors at a finer granularity: individual cache lines. In the previous chapter we described the generational lifetime behavior of cache lines and we introduced the live time and the dead time period within a cache line generation. Live time is the time between the first and last access within a generation, while dead time is the time between the last access and the eviction. The key idea of timekeeping for cache leakage control is that if a dead time can be identified at run time, then the associated cache line can be turned off to cut off leakage power. Because the access after the dead time is to a different cache line, turning off the cache line will not lead to extra cache misses. Thus turning off cache lines during dead times can reduce cache leakage energy without sacrificing performance.

To motivate the potential of this approach, we start by presenting an idealized study of its advantages. Here, we have run simulations using an "oracle" predictor of when dead time starts in each cache line. That is, we note when a cache line has had its last successful access, before the cache miss that begins the next generation. We imagine, in this section only, that we can identify these dead times with 100% accuracy and eliminate cache leakage during the dead times.

Figure 3.1 illustrates the fraction of dead time we measured for a 32KB L1 data cache on our benchmark collection. This is the total fraction of time cache lines spend in their dead times [1]. We only count complete generations that end with a miss in the cache frame. The average across the benchmark suite is quite high: around 65% for integer benchmarks

[1] We sum the dead times and the live times of all the generations we encounter and we compute the ratio $dead/(dead+live)$. We do not compute individual dead ratios per generation and then average them, as this would skew the results towards short generations.

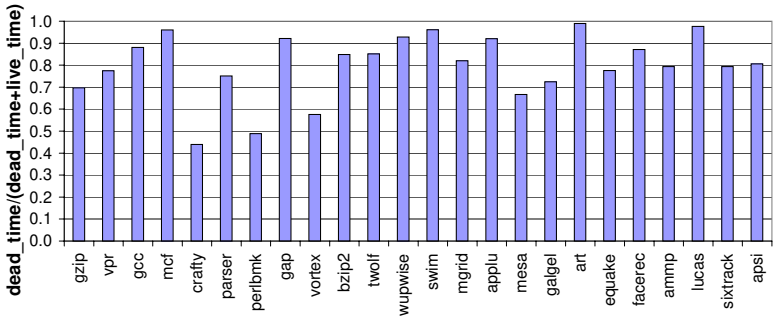

Figure 3.1: Fraction of time cached data are "dead."

and even higher (80%) for FP benchmarks. From the dead time distribution in Figure 2.9, we can see that it is mainly the long dead times that dominates the aggregate length of all dead times. Consider next an oracle predictor which knows precisely when a cache line becomes dead. With it, we could turn the cache line off with zero impact on cache miss rate or program performance. Such an oracle predictor would allow us to save power directly proportional to the shutoff ratio. If on average, 65% of the cache is shut off, and if we can implement this shutoff with negligible overhead power, then we can cut cache leakage power by one half or more.

Note that this oracle prediction is not necessarily an upper bound on the leakage power improvements to be offered by putting cache lines to sleep. Rather, the oracle predictor offers the best possible leakage power improvements *subject to the constraint that cache misses do not increase*. There may be cases where even though a line is live (i.e., it will be referenced again before eviction) the reuse will be far into the future. In such cases, it may be power-optimal to shut off the cache line early, mark it as invalid, and accept a moderate increase in the number of cache misses. Later sections will offer more realistic policies for managing these tradeoffs. On the other hand, real world attempts to put cache lines to sleep will also incur some small amounts of overhead power as we also discuss in the following sections.

3.2 Timekeeping for Leakage Control: Cache Decay

We now examine possible timekeeping policies for guiding how to use a mechanism that can reduce cache leakage by turning off individual cache lines. A key aspect of these policies is the desire to balance the potential for saving leakage energy (by turning lines off) against the potential for incurring extra L2 cache accesses (if we introduce extra misses by turning lines off prematurely). We wish to either deduce immediately at a reference point that the cache line is now worth turning off, or else infer this fact by watching its behavior over time, deducing when no further accesses are likely to arise, and therefore turning the line off. This section focuses on the latter case, which we refer to as cache decay.

With oracle knowledge of reference patterns, Figure 3.1 demonstrated that the leakage energy to be saved would be significant. The question is: can we develop policies that come acceptably close to this oracle? In fact, this question can be approached by relating it to the theoretical area of competitive algorithms [44]. Competitive algorithms make cost/benefit decisions online (i.e., without oracle knowledge of the future) that offer benefits within a constant factor of an optimal offline (i.e., oracle-based) algorithm. A body of computer systems work has previously successfully applied such strategies to problems including superpage promotion for TLB performance, prefetching and multiprocessor synchronization [41, 61].

A generic policy for competitive algorithms is to take action at a point in time where the extra cost we have incurred so far by waiting is precisely equal to the extra cost we might incur if we act but guess wrong. Such a policy, it has been shown, leads to worst case cost that is within a factor of two of the offline optimal algorithm.[2]

For example, in the case of our cache decay policy we are trying to determine when

[2] [61] includes a helpful example: the ski rent-vs.-buy problem. For example, if ski rental charges are $40 per day, and skis cost $400 to buy, then online approaches suggest that a beginning skier (who doesn't know whether they will enjoy skiing or not) would be wise to rent skis 10 times before buying. This equalizes the rental cost to the purchase cost, bounding total cost at two times the optimal offline approach.

to turn a cache line off. The longer we wait, the higher the leakage energy dissipated. On the other hand, if we prematurely turn off a line that may still have hits, then we inject extra misses which incur dynamic power for L2 cache accesses. Competitive algorithms point us towards a solution: we could leave each cache line turned on until the static energy it has dissipated since its last access is precisely equal to the dynamic energy that would be dissipated if turning the line off induces an extra miss. With such a policy, we could guarantee that the energy used would be within a factor of two of that used by the optimal offline policy shown in Figure 3.1.

As will be detailed in Section 3.4, we estimate the dynamic energy required for a single L2 access to be roughly 9 times as large as the static leakage energy dissipated by whole L1 data cache. If we consider just one line from the L1 cache, then that ratio gets multiplied by 1024, since the cache we are studying has 1024 lines. This analysis suggests that to come within a factor of two of the oracle-policy we should leave cache lines turned on until they have gone roughly 10,000 cycles without an access. At that point, we should turn them off. Since the dynamic vs. static energy ratio varies so heavily with design and fabrication factors, we consider a wider range of decay intervals, from 1K to 512K cycles, to explore the design space thoroughly.

The optimality of this oracle-based policy applies to the case where no additional cache misses are allowed to be added. In cases of very glacial reuse, however, it may be energy-beneficial to turn off a cache line, mark its contents invalid, and incur an L2 cache miss later, rather than to hold contents in L1 and incur leakage power for a long time.

For the online approach (and its bound) to be of practical interest, the wait times before turning a cache line off must be short enough to be seen in real-life. That is, the average dead times seen in real programs must be long enough to allow the lines to be turned off a useful amount of the time. Therefore, we wish to characterize the cache dead times typically seen in applications, in order to gauge what sorts of decay intervals may be practical.

Figure 3.2 shows cumulative distributions of access intervals and dead times for gzip (dotted lines) and applu (solid lines). The last point on the horizontal axis graph represents the tail of the distributions beyond that point. Recall that the term *access interval* refers to the time between any two accesses during the live-time of a cache generation. Dead time refers to the time between the last access to an item in cache, and when it is actually evicted. Our experiments show that across the benchmark suite, there are a sizable fraction of dead times greater than 10,000 cycles. Thus, the time range suggested by the online policy turns out to be one of significant practical promise.

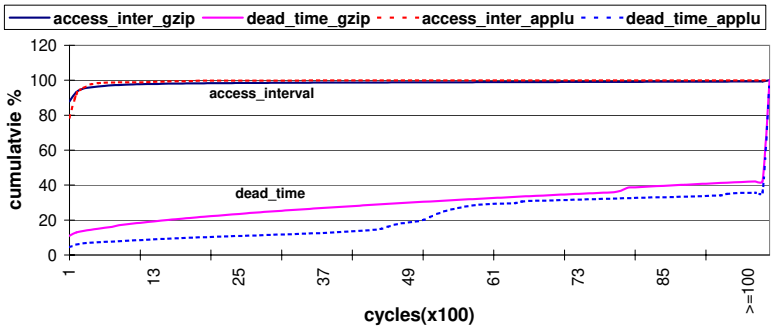

Figure 3.2: Cumulative distribution of Access Interval and Dead Time for gzip and applu.

Figure 3.2 also highlights the fact that there is a huge difference between average access interval and average dead time. For gzip, the average access interval is 458 cycles while the average dead time is nearly 38,243 cycles. For applu, the results are similar: 181 cycles per access interval and 14,984 cycles per dead time. This suggests that many dead times are not only long, but also can be moderately easy to identify, since we will be able to notice when the flurry of short access interval references is over.

Based on these observations, we focus on timekeeping techniques in which cache decay intervals (the wait times to turn lines off) are set between 1K and 512K cycles for the L1 cache. These intervals span broadly over the range suggested by both competitive algorithms and the dead time distributions. The following section details a particular way of implementing a cache decay policy with a single fixed decay interval. Section 3.7 refines

this approach to consider an adaptive policy whose decay interval automatically adjusts to application behavior.

3.3 Cache Decay: Implementation

To switch off a cache line we use the *gated* V_{dd} technique developed by Yang et al. [59]. The idea in this technique is to insert a "sleep" transistor between the ground (or supply) and the SRAM cells of the cache line. The stacking effect [12] of this transistor when it is off reduces by orders of magnitude the leakage current of the SRAM cell transistors to the point that leakage power of the cache line can be considered negligible. According to [59] a specific implementation of the gated V_{dd} transistor (NMOS gated V_{dd}, dual V_t, wide, with charge pump) results in minimal impact in access latency but with a 5% area penalty. We assume this implementation of the gated V_{dd} technique for cache decay.

One way to represent recency of a cache line's access is via a binary counter associated with the cache line. Each time the cache line is accessed the counter is reset to its initial value. The counter is incremented periodically at fixed time intervals. If no accesses to the cache line occur and the counter saturates to its maximum count (signifying that the decay interval has elapsed) it switches off power to the corresponding cache line.

Our competitive algorithm bound and the dead time distributions both indicate that decay intervals should be in the range of tens of thousands of cycles. Such large decay intervals make it impractical for the counters to count cycles—too many bits would be required. Instead, it is necessary for the counters to tick at a much coarser level. Our solution is to utilize a hierarchical counter mechanism where a single global cycle counter is set up to provide the ticks for smaller cache-line counters (as shown in Figure 3.3).

Our simulations show that an infrequently-ticking two-bit counter per cache line provides sufficient resolution and produces the same results as a larger counter with the same

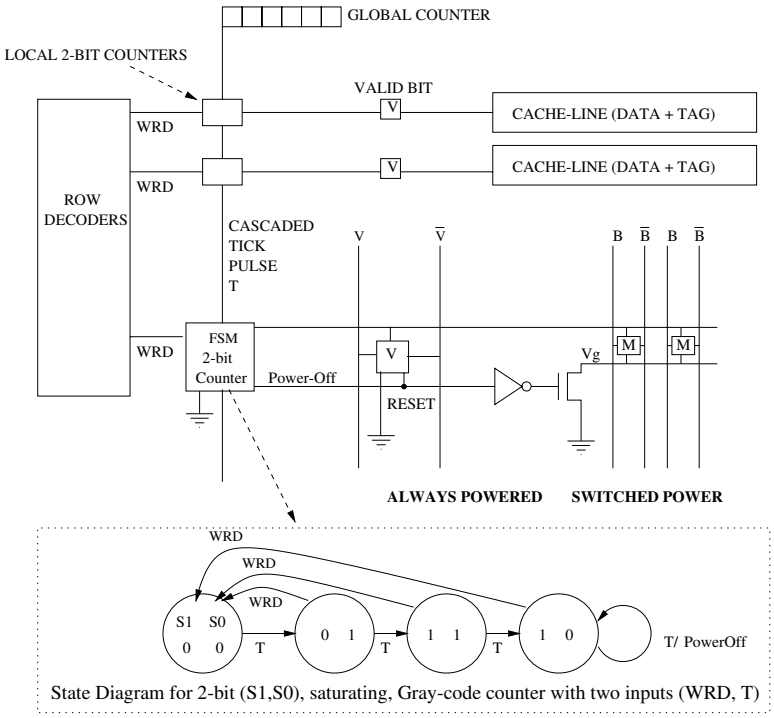

Figure 3.3: Implementing cache decay with hierarchical counters.

effective decay interval. If it takes four ticks of the 2-bit counter to decay a cache line (Figure 3.3), the resulting decay interval is—on average—3.5 times the period of the global counter.

In our power evaluations, we assume that the global counter will come for free, since many processors already contain various cycle counters for the operating system or for performance counting [14, 50, 81]. If such counters are not available, a simple N-bit binary ripple counter could be built with 40N + 20 transistors, of which few would transition each cycle.

To minimize state transitions in the local 2-bit cache-line counters and thus minimize dynamic power consumption we use Gray coding so only one bit changes state at any time. Furthermore, to simplify the counters and minimize transistor count we choose to implement them asynchronously. Each cache line contains circuitry to implement the state machine depicted in Figure 3.3. The two inputs to the local counters, the global tick signal

T generated by the global counter and the cache-line access signal WRD, are well behaved so there are no meta-stability problems. The output signal Power-Off, controls the *gated* V_{dd} transistor and turns off power when asserted. To avoid the possibility of a burst of writebacks with every global tick signal (if multiple dirty lines decay simultaneously) the tick signal is *cascaded* from one local counter to the next with a one-cycle latency. This does not affect results but it spreads writebacks in time.

Local counters change value with every global counter (T) pulse. However, this happens at very coarse intervals (equal to the period of the global counter). Resetting a local counter with an access to a cache line is not a cause of concern either. If the cache line is heavily accessed the counter has no opportunity to change from its initial value so resetting it does not expend any dynamic power (none of the counter's transistors switch). The cases where power is consumed are accesses to cache lines that have been idle for at least one period of the global counter. Our simulation results indicate that over all 1024 2-bit counters used in our scheme, there are 0.2 bit transitions per cycle on average. Modeling each counter as a 2-bit register in Wattch [7], we estimate roughly .1pJ per access. Therefore, at an average of 0.02pJ per cycle, the power expended by all 1024 of these infrequently-ticking counters is roughly 4 orders of magnitude lower than the cache leakage energy which we estimate at 0.45nJ per cycle in the next section. For this reason, our power analysis will consider this counter overhead to be negligible from this point forward.

Switching off power to a cache line has important implications for the rest of the cache circuitry. In particular, the first access to a powered-off cache line should: (i) result in a miss (since data and tag might be corrupted without power) (ii) reset the counter and restore power to the cache line and (iii) delay an appropriate amount of time until the cache-line circuits stabilize after power is restored. To satisfy these requirements we use the Valid bit of the cache line as part of the decay mechanism (Figure 3.3). First, the valid bit is always powered. Second, we add a reset capability to the valid bit so the Power-Off signal can

clear it. Thus, the first access to a power-off cache line always results in a miss regardless of the contents of the tag. Since satisfying this miss from the lower memory hierarchy is the only way to restore the valid bit, a newly-powered cache line will have enough time to stabilize. In addition, no other access (to this cache line) can read the possibly corrupted data in the interim.

Analog implementation: Another way to represent the recency of a cache line's access is via charge stored on a capacitor. Each time the cache line is accessed, the capacitor is grounded. In the common case of a frequently-accessed cache line, the capacitor will be discharged. Over time, the capacitor is charged through a resistor connected to the supply voltage (V_{dd}). Once the charge reaches a sufficiently high level, a voltage comparator detects it, asserts the Power-Off signal and switches off power to the corresponding cache line. Although the RC time constant cannot be changed (it is determined by the fabricated size of the capacitor and resistor) the bias of the voltage comparator can be adjusted to different temporal access patterns. An analog implementation is inherently noise sensitive and can change state asynchronously with the remainder of the digital circuitry. Some method of synchronously sampling the voltage comparator must be used to avoid meta-stability. Since an analog implementation can be fabricated to mimic the digital implementation, the rest of this paper focuses on the latter.

3.4 Power Evaluation Methodology

A basic premise of our evaluations is to measure the static power saved by turning off portions of the cache, and then compare it to the extra dynamic power dissipated in our method. Our method dissipates extra dynamic power in two main ways. First, we introduce counter hardware to support our decay policy decisions, so we need to account for the dynamic power of these counters in our evaluations. Second, our method can dissipate

extra dynamic power in cases where our decay policy introduces additional L1 cache misses not present in the original reference stream. These L1 misses translate to extra L2 reads and sometimes also extra writebacks. Turning off a dirty line results in an early writeback which is extraneous only if paired with an extra miss. For the rest of this paper, when we discuss extra misses we implicitly include associated extra writebacks.

Since both leakage and dynamic power values vary heavily with different designs and fabrication processes, it is difficult to nail down specific values for evaluation purposes. Rather, in this paper we focus on ratios of values. In this section, we describe our rationale for the range of ratio values we focus on. Later sections present our results for different ratios within this range.

A key energy metric in our study is "normalized cache leakage energy". This refers to a ratio of the energy of the L1 with cache decay policies, versus the original L1 cache leakage energy. The numerator in this relationship sums three terms. The first term is the improved leakage energy resulting from our policies. The second term is energy from counter maintenance or other overhead hardware for cache decay policies. The third term is extra dynamic energy incurred if cache decay introduces extra L1 misses that result in extra L2 cache accesses (reads and writebacks).

Dividing through by original cache leakage energy, we can use weighting factors that relate the dynamic energy of extra L2 accesses and extra counters, to the original cache leakage energy per cycle. Thus, the normalized cache leakage energy after versus before our improvements can be represented as the sum of three terms: $ActiveRatio + (Ovhd : leak)(OvhdActivity) + (L2Access : leak)(extraL2Accesses)$. ActiveRatio is the average fraction of the cache bits, tag or data, that are powered on. *Ovhd:leak* is the ratio of the cost of counter accesses in our cache decay method relative to the leakage energy. This multiplied by overhead activity (*OvhdActivity*) gives a relative sense of overhead energy in the system. The *L2Access:leak* ratio relates dynamic energy due to an additional miss (or

writeback) to a single clock cycle of static leakage energy in the L1 cache. Multiplying this by the number of extra L2 accesses induced by cache delay gives the dynamic cost induced. By exploring different plausible values for the two key ratios, we present the benefits of cache decay somewhat independently of fabrication details.

Considering appropriate ratios is fundamental in evaluating our policies. We focus here on the *L2Access:leak* ratio. As discussed in the Section 3.3, the dynamic energy of decay counters are negligible therefore are ignored from now on.

We wish to turn off cache lines as often as possible in order to save leakage power. We balance this, however, against a desire to avoid increasing the miss rate of the L1 cache. Increasing the miss rate of the L1 cache has several power implications. First and most directly, it causes dynamic power dissipation due to an access to the L2 cache, and possible additional accesses down the memory hierarchy. Second, a L1 cache miss may force dependent instructions to stall, interfering with smooth pipeline operation and dissipating extra power. Third and finally, the additional L1 cache miss may cause the program to run for extra cycles, and these extra cycles will also lead to extra power being dissipated.

We encapsulate the energy dissipated due to an extra miss into a single ratio called *L2Access:leak*. The predominant effect to model is the amount of dynamic power dissipated in the L2 cache and beyond, due to the L1 cache miss. Additional power due to stalls and extra program cycles is minimal. Benchmarks see very few cycles of increased runtime ($< 0.7\%$) due to the increased misses for the decay policies we consider. In fact, in some situations, some benchmarks actually run slightly faster with cache decay techniques. This is because writebacks occur eagerly on cache decays, and so are less likely to stall the processor later on [23].

To model the ratio of dynamic L2 access energy compared to static L1 leakage per cycle, we first refer to recent work which estimates dynamic energy per L2 cache access

in the range of 3-5nJ per access for L2 caches of the size we consider (1MB) [39]. We then compared this data to industry data by back-calculating energy per cache access for Alpha 21164's 96KB S-cache; it is roughly 10nJ per access for a 300MHz fabricated in a 0.5μ process [6]. Although the S-cache is about one-tenth the capacity of the L2 caches we consider, our back-calculation led to a higher energy estimate. First, we note that banking strategies typically employed in large caches lessen the degree by which energy-per-access scales with size. Second, the higher 0.5μ feature size used in this older design would lead to larger capacitance and higher energy per access. Our main validation goal was to check that data given by the analytic models are plausible; our results in later sections are plotted for ratios varying widely enough to absorb significant error in these calculations.

The denominator of the *L2Access:leak* relates to the leakage energy dissipated by the L1 data cache. Again, we collected this data from several methods and compared. From the low-V_t data given in Table 2 of [80], one can calculate that the leakage energy per cycle for a 32KB cache will be roughly 0.45nJ. A simple aggregate calculation from industry data helps us validate this. Namely, using leakage power of roughly 2-5% of current CPU power dissipation, L1 cache is roughly 10-20% of that leakage [5], and CPU power dissipations are around 75W. This places L1 leakage energy at roughly 0.3nJ per cycle. Again, both methods of calculating this data give results within the same order-of-magnitude.

Dividing the 4nJ dynamic energy per access estimate by the .45nJ static leakage per cycle estimate, we get a ratio of 8.9 relating extra miss power to static leakage per cycle. Clearly, these estimates will vary widely with design style and fabrication technology though. In the future, leakage energy is expected to increase dramatically, which will also impact this relationship. To account for all these factors, our energy results are plotted for several *L2Access:leak* ratios varying over a wide range (5 to 100). Our results are conservative in the sense that high leakage in future technologies will tend to decrease this ratio. If that happens, it will only improve on the results we present in this chapter.

Processor Core	
Instruction Window	80-RUU, 40-LSQ
Issue width	4 instructions per cycle
Functional Units	4 IntALU,1 IntMult/Div, 4 FPALU,1 FPMult/Div, 2 MemPorts
Memory Hierarchy	
L1 Dcache Size	32KB, 1-way, 32B blocks, WB
L1 Icache Size	32KB, 1-way, 32B blocks, WB
L2	Unified, 1MB, 8-way LRU, 64B blocks,6-cycle latency, WB
Memory	100 cycles
TLB Size	128-entry, 30-cycle miss penalty

Table 3.1: Configuration of simulated processor for cache decay.

3.5 Simulation Model Parameters

Simulations in this chapter are based on the SimpleScalar framework [8]. Our model processor has sizing parameters that closely resemble Alpha 21264 [22], but without a clustered organization. The main processor and memory hierarchy parameters are shown in Table 3.1.

We evaluate our results using benchmarks from the SPEC CPU2000 [73] and MediaBench suites [46]. The MediaBench applications help us demonstrate the utility of cache decay for applications with significant streaming data. The benchmarks are compiled for the Alpha instruction set using the Compaq Alpha compiler with SPEC *peak* settings. For each program, we follow the recommendation in [63], but skip a minimum of 1 billion instructions. We then simulate 500M instructions using the reference input set.

3.6 Cache Decay: Simulation Results

We now present experimental results for the timekeeping cache decay policy based on binary counters described in section 3.3. First Figures 3.4 and 3.5 plot the active ratio and

miss rate as a function of cache decay interval for a collection of integer and floating point programs. In each graph, each application has five bars. In the active ratio graph, the first bar is the active ratio for a traditional 32KB L1 data cache. Since all the cache is turned on all the time, the active ratio is 100%. Furthermore, we have determined that our benchmark programs touch the entirety of the standard caches for the duration of execution (active ratio over 99%). The other bars show the active ratio (average number of cache bits turned on) for decay intervals ranging from 512K cycles down to 1K cycles. Clearly, shorter decay intervals dramatically reduce the active ratio, and thus reduce leakage energy in the L1 data cache, but that is only part of the story. The miss rate graphs show how increasingly aggressive decay intervals affect the programs' miss rates.

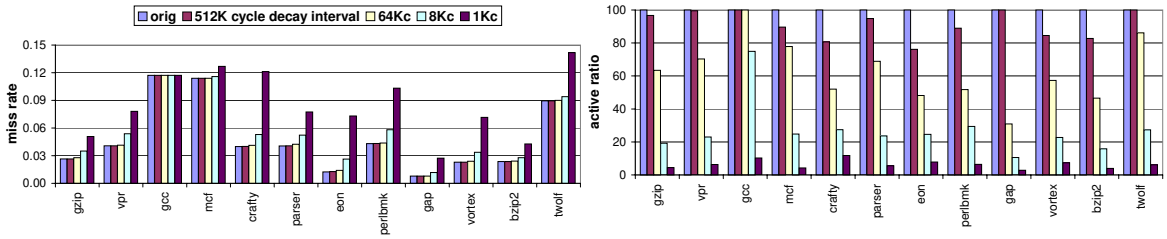

Figure 3.4: Miss rate and active ratio of a 32KB decay cache for SPECint 2000.

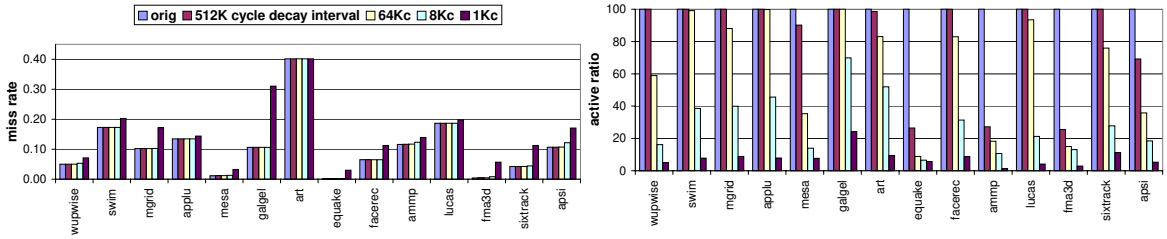

Figure 3.5: Miss rate and active ratio of a 32KB decay cache for SPECfp 2000.

Figure 3.6 plots similar data averaged over all the benchmarks. The upper curve corresponds to a traditional cache in which we vary the cache size and see the miss rate change. In a traditional cache, active size is just the cache size. The lower curve in this graph corresponds to a decay cache whose full size is fixed at 32KB. Although the decay cache's full size is fixed, we can vary the decay interval and see how this influences the active size.

(This is the apparent size based on the number of cache lines turned on.) Starting at the 16KB traditional cache and dropping downwards, one sees that the decay cache with the same active size has much better miss rate characteristics.

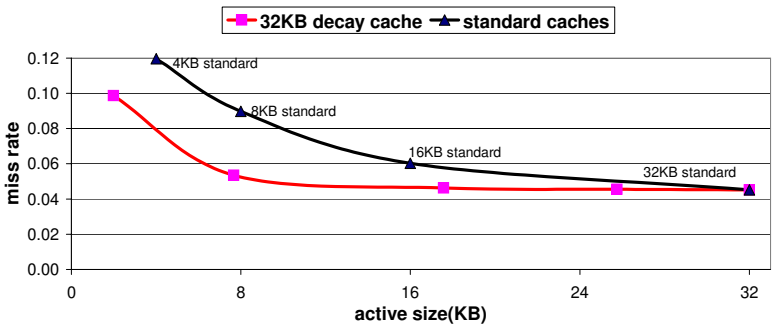

Figure 3.6: Comparison of standard 8KB, 16KB, and 32KB caches to a fixed-size 32KB decay cache with varying cache decay intervals. The different points in the decay curve represent different decay intervals. From left to right, they are: 1Kcycles, 8Kcycles, 64Kcycles, and 512Kcycles. Active size and miss rate are geometric means over all SPEC 2000 benchmarks.

Figure 3.7 shows the normalized cache leakage energy metric for the integer and floating point benchmarks. In this graph, we assume that *L2Access:leak* ratio is equal to 10 as discussed in section 3.3. We normalize to the leakage energy dissipated by the original 32KB L1 data cache with no decay scheme in use. Although the behaviors of each benchmark are unique, the general trend is that a decay interval of 8K cycles shows the best energy improvements. This is quite close to the roughly 10Kcycle interval suggested for worst-case bounding by the theoretical analysis. For the integer benchmarks, all of the decay intervals — including even 1Kcycle for some —result in net improvements. For the floating point benchmarks, 8Kcycle is also the best decay interval. All but one of the floating point benchmarks are improved by cache decay techniques for the full decay-interval range.

We also wanted to explore the sensitivity of our results to different ratios of dynamic L2 energy versus static L1 leakage. Figure 3.8 plots three curves of normalized cache leakage energy. Each curve represents the average of all the SPEC benchmarks. The

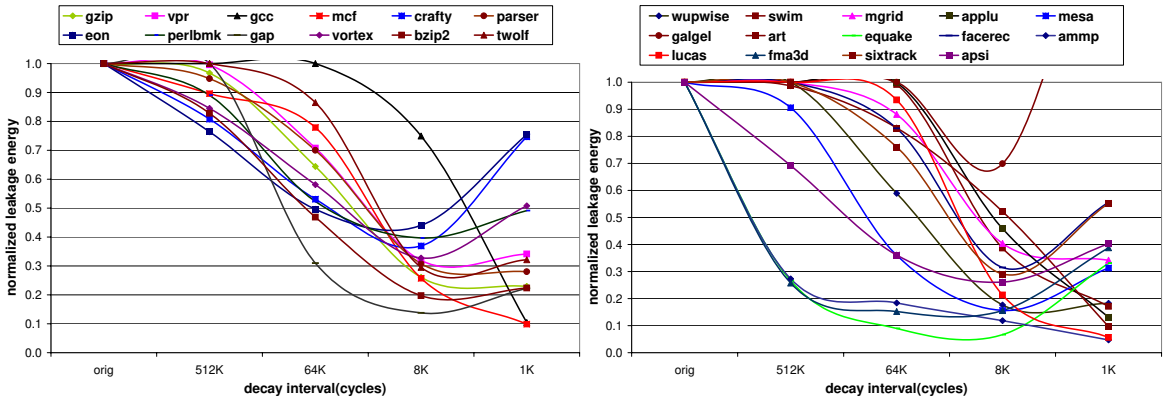

Figure 3.7: Normalized cache leakage energy for an *L2Access:leak* ratio of 10. This metric takes into account both static energy savings and dynamic energy overhead. Top graph shows SPECint2000; bottom graph shows SPECfp2000.

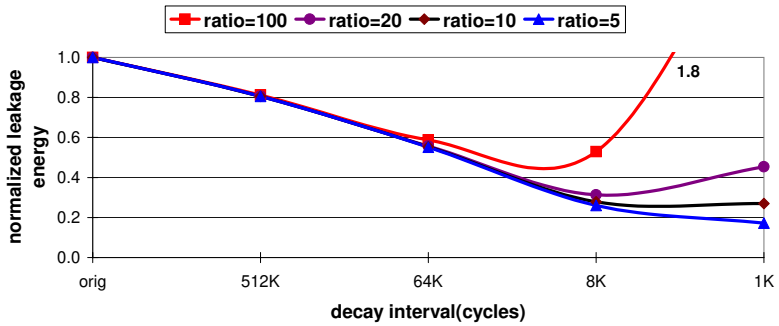

Figure 3.8: Normalized L1 data cache leakage energy averaged across SPEC suite for various *L2Access:leak* ratios.

curves correspond to *L2Access:leak* ratios of 5, 10, 20, and 100. All of the ratios show significant leakage improvements, with smaller ratios being especially favorable. When the *L2Access:leak* ratio equals 100, then small decay intervals (less than 8K cycles) are detrimental to both performance and power. This is because short decay intervals may induce extra cache misses by turning off cache lines prematurely; this effect is particularly bad when *L2Access:leak* is 100 because high ratios mean that the added energy cost of additional L2 misses is quite high.

To assess these results one needs to take into consideration the impact on performance. If cache decay slows down execution because of the increased miss rate then its power advantage diminishes. For the decay scenarios we consider, not only we do not observe

any slow-down but in fact we observe a very slight speed up in some cases, which we attribute to eager writebacks [23]. Beyond this point, however, cache decay is bound to slow down execution. For our simulated configuration, performance impact is negligible except for very small decay intervals: the 8Kcycle interval—which yields very low normalized leakage energy (Figures 3.7 and 3.8)—decreases IPC by 0.1% while the 1Kcycle interval—which we do not expect to be used widely—decreases IPC by 0.7%. Less aggressive processors might suffer comparably more from increased miss rates, which would make very small decay intervals undesirable.

In addition to the SPEC applications graphed here, we have also done some initial studies with MediaBench applications [46]. The results are even more successful than those presented here partly due to the generally poor reuse seen in streaming applications; MediaBench applications can make use of very aggressive decay policies. Since the working set of MediaBench can, however, be quite small (for gsm, only about 50% of the L1 data cache lines are *ever* touched) we do not present the results here.

3.7 Cache Decay: Adaptive Variants

So far we have investigated cache decay using a single decay interval for all of the cache. We have argued that such a decay interval can be chosen considering the relative cost of a miss to leakage power in order to bound worst-case performance. However, Figure 3.7 shows that in order to achieve best average-case results this choice should be application-specific. Even within an application, a single decay interval is not a match for every generation: generations with shorter dead times than the decay interval are ignored, while others are penalized by the obligatory wait for the decay interval to elapse. In this section we present an adaptive decay approach that chooses decay intervals at run-time to match the behavior of individual cache lines.

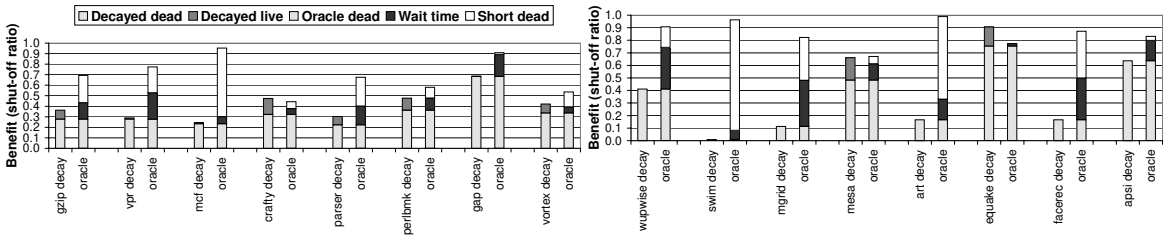

Figure 3.9: Lost opportunities for cache decay with 64Kcycle decay intervals. SPECint2000 and SPECfp2000.

Motivation for an adaptive approach: Figure 3.9 shows details about why a single decay interval cannot capture all the potential benefit of an oracle scheme. In this figure two bars are shown for each program: a decay bar on the left and an oracle bar on the right. Within the bar for the oracle-based approach, there are three regions. The lower region of the oracle bar corresponds to the lower region of the decay bar which is the benefit (the shut-off ratio) that comes from decaying truly dead cache lines. The two upper regions of the oracle bar represent benefit that the single-interval decay schemes of Section 3.6 cannot capture. The middle region of the oracle bar is the benefit lost while waiting for the decay interval to elapse. The upper region is lost benefit corresponding to dead periods that are shorter than the decay interval. On the other hand, the decay scheme can also mistakenly turn off *live* cache lines. Although this results in extraneous misses (*decay misses*) it also represents benefit in terms of leakage power. This effect is shown as the top region of the decay bars. For some SPECfp2000 programs the benefit from short dead periods is quite large in the oracle bars.

Implementation: An ideal decay scheme would choose automatically the best decay interval for each generation. Since this is not possible without prior knowledge of a generation's last access, we present here an adaptive approach to chose decay intervals per cache-line.

Our adaptive scheme attempts to choose the smallest possible decay interval (out of a predefined set of intervals) individually for each cache-line. The idea is to start with

a short decay interval, detect whether this was a mistake, and adjust the decay interval accordingly. A mistake in our case is to prematurely decay a cache-line and incur a decay miss. We can detect such mistakes if we leave the tags always powered-on but this is a significant price to pay (about 10% of the cache's leakage for a 32KB cache). Instead we opted for a scheme that infers possible mistakes according to how fast a miss appears after decay. We determined that this scheme works equally well or *better* than an exact scheme which dissipates tag leakage power.

The idea is to reset a line's 2-bit counter upon decay and then reuse it to gauge dead time (Figure 3.10). If dead time turns out to be very short (the local counter did not advance a single step) then chances are that we have made a mistake and incurred a decay-miss. But if the local counter reaches its maximum value while we are still in the dead period then chances are that this was a successful decay. Upon mistakes—misses with the counter at minimum value (00 in Figure 3.10)—we adjust the decay interval upwards; upon successes—misses with counter at maximum value (10)—we adjust it downwards. Misses with the counter at intermediate values (01 or 11) do not affect the decay interval.

We use exponentially increasing decay intervals similarly to Ethernet's exponential back-off collision algorithm but the set of decay intervals can be tailored to the situation. As we incur mistakes, for a cache line, we exponentially increase its decay interval. By backing-off a single step in the decay-interval progression rather than jumping to the smallest interval we introduce *hysteresis* in our algorithm.

Implementation of the adaptive decay scheme requires simple changes in the decay implementation discussed previously. We introduce a small field per cache line, called *decay speed field*, to select a decay interval. An *N*-bit field can select up to 2^N decay intervals. The decay-speed field selects different tick pulses coming from the same or different global cycle counters. This allows great flexibility in selecting the relative magnitude of the decay intervals. The value of this field is incremented whenever we incur a *perceived* decay miss

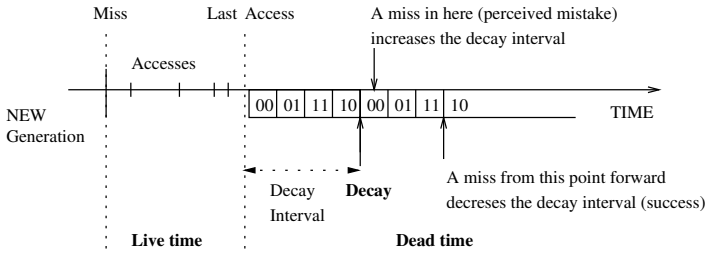

Figure 3.10: Adaptive decay.

and decremented on a *perceived* successful decay. We assume that higher value means longer decay interval.

Results: Figure 3.11 presents results of an adaptive scheme with 10 decay intervals (4-bit decay-speed field). The decay intervals range from 1K cycles to 512K cycles (the full range used in our previous experiments) and are successive powers-of-two. In the same figure we repeat results for the single-interval decay and for various standard caches. We also plot *iso-power* lines, lines on which total power dissipation remains constant (for $L2Access : leak = 10$). The adaptive decay scheme automatically converges to a point below the single-interval decay curve. This corresponds to a total power lower than the iso-power line *tangent to* the decay curve. This point has very good characteristics: significant reduction in active area and modest increase in miss ratio. Intuitively, we would expect this from the adaptive scheme since it tries to maximize benefit but also is aware of cost. This behavior is application-independent: the adaptive scheme tends to converge to points that are close or lower than the highest iso-power line tangent to the single-decay curve.

Discussion: Adaptive cache decay is essentially a feedback-control mechanism, which tries to be more aggressive when successful, but more conservative in case of failure. Our results show that the parameters in our adaptive scheme (see Figure 3.10) match well with the $L2Access : leak$ ratio of 10. When such ratio changes, the scheme can be adjusted to match the new ratio, either by re-defining what happens in each counter state, or by selecting a different decay interval range (for example, 8-1024K instead of 1-512K). Special care

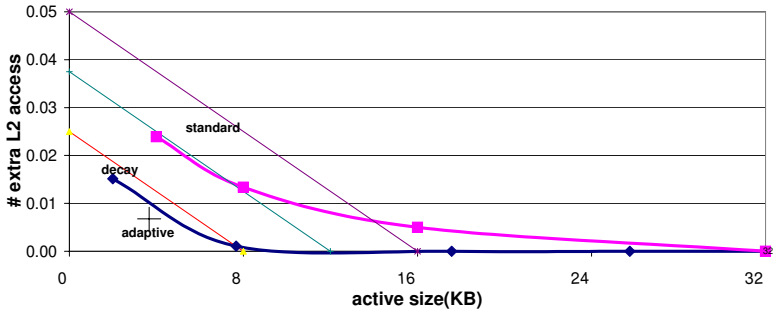

Figure 3.11: Effect of adaptive decay. Iso-power lines show constant power ($L2Access$: $leak = 10$). Results averaged over all SPEC2000.

must be taken when choosing the smallest decay interval of the range. If decay interval is smaller than typical access intervals, many cache lines will be turned off when they are still active, leading to an excessive number of decay-caused misses. To avid this, the value of the smallest decay interval should be safely away from those of typical access intervals.

Adaptive decay for set-associative caches: Figure 3.12 demonstrates the effect of adaptive decay for a 4-way 32KB data cache. As in the previous section, the decay interval starts with an aggressively small value but increases in the event of a decay caused miss. To identify a decay caused miss, instead of the counter-based huristic described above, we keep part of the cache tag always powered-on. If during a cache access, the lookup tag matches the partial powered-on tag but the data is decayed, then we declare it a decay caused miss and modify the decay interval to a more conservative (larger) value. On the other hand, if the lookup tag does not match the partial tag, indicating a "true" cache miss not caused by cache decay, then the decay interval is reset to the most aggressive (smallest) value. Since the powered-on partial tag leads to additional leakage power, we should choose it to be as small as possible. On the other hand, too few powered on bits will lead to aliasing effect where the lookup tag matches the partial tag but does not match the whole tag. In our experiment, we found a 5-bit partial tag effectively remove the aliasing effect with minimal additional leakage power.

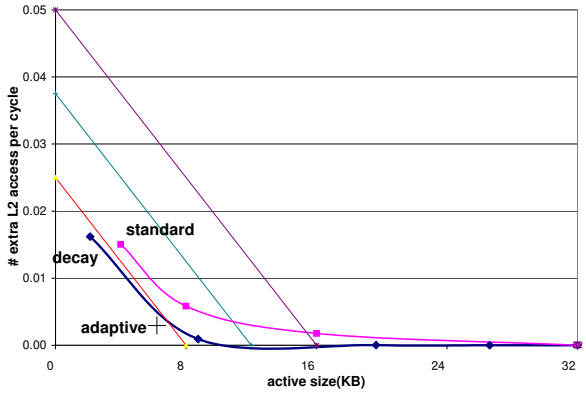

Figure 3.12: Effect of adaptive decay for a 4-way 32KB cache. Iso-power lines show constant power ($L2Access : leak = 10$). Results averaged over all SPEC2000.

3.8 Changes in the Generational Behavior and Decay

In this section we will first examine sensitivity of cache decay to cache size, associativity and block size. Then we show the effectiveness of cache decay for the instruction cache. Finally we will discuss how cache decay can be applied when multi-level cache hierarchies or multi-programming change the apparent generational behavior of cache lines.

3.8.1 Sensitivity to Cache Size, Associativity and Block Size

Cache characteristics usually vary with different cache geometries, namely cache size, associativity and block size. Now we explore the effect of changing these parameters on cache decay behavior. Figure 3.13 and 3.14 plot ActiveRatio-MissRate curves of different cache size, associativity and block size for SPEC2000 benchmark suite. Across the configurations, we observed trends consistent to the 32KB direct-mapped cache shown in the previous section. Cache decay constantly show a benefit even for considerably small caches.

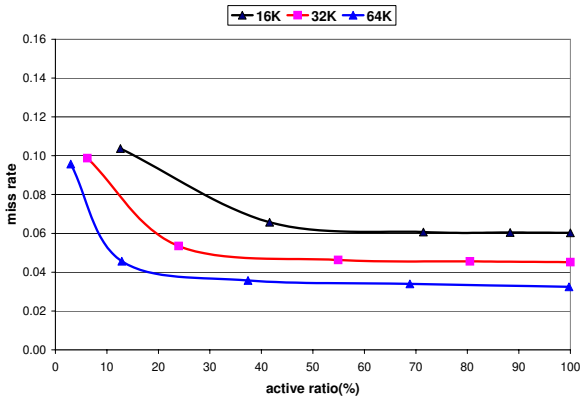

Figure 3.13: ActiveRatio-MissRate curve for mcf for different sizes of L1 data cache.

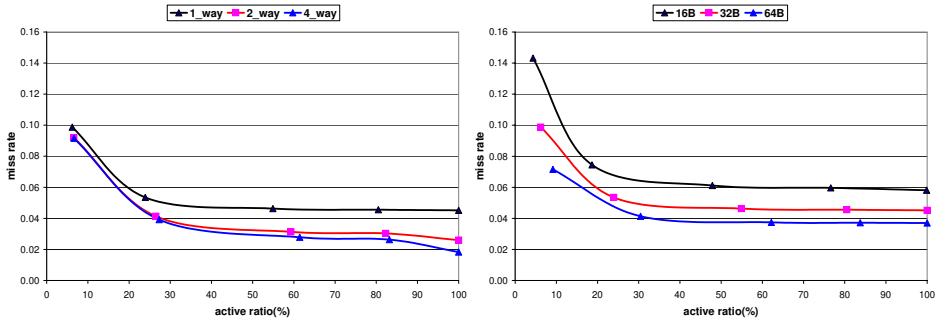

Figure 3.14: ActiveRatio-MissRate curve for mcf for different associativity(left) and block size(right) of a 32KB L1 data cache.

3.8.2 Instruction Cache

Cache decay can be applied to instruction caches since they typically exhibit even more locality than data caches. In fact, compared to data cache, instruction cache has the additional benefit that it does not have any writeback traffic. Figure 3.15 shows the normalized leakage energy for a 32KB L1 instruction cache. Notice that even without decay, the instruction cache is not fully touched during our simulation period. The figure shows that decay works very well except for very small decay intervals.

3.8.3 Multiple Levels of Cache Hierarchy

Cache decay is likely to be useful at multiple levels of the hierarchy since it can be usefully employed in any cache in which the active ratio is low enough to warrant line shut-offs.

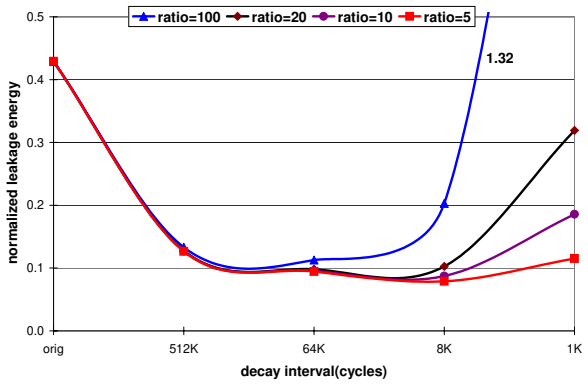

Figure 3.15: Normalized L1 instruction cache leakage energy averaged across SPEC suite for various *L2Access:leak* ratios.

For several reasons, the payoff is likely to increase as one moves outward in the hierarchy. First, a L2 or level-three cache is likely to be larger than the L1 cache, and therefore will dissipate more leakage power. Second, outer levels of the cache hierarchy are likely to have longer generations with larger dead time intervals. This means they are more amenable to our time-based decay strategies. On the other hand, the energy consumed by any extra L2 misses we induce could be quite large, especially if servicing them requires going off chip.

The major difference between L1 and L2 is the filtering of the reference stream that takes place in L1 which changes the distribution of the access intervals and dead periods in L2. Our data shows that the average access interval and dead time for L2 cache are 79K and 2.7M cycles respectively. Though access intervals and dead periods become significantly larger, their relative difference remains large and this allows decay to work.

The increased access intervals and dead times suggest we should consider much larger decay intervals for the L2 compared to those in the L1. This meshes well with the competitive analysis which also points to an increase in decay interval because the cost of an induced L2 cache miss is so much higher than the cost of an induced L1 cache miss. As a simple heuristic to choose a decay interval, we note that since there is a 100-fold increase in the dead periods in L2, we will also multiply our L1 decay interval by 100. Therefore a 64Kcycle decay interval in L1 translates to decay intervals on the order of 6400K cycles in

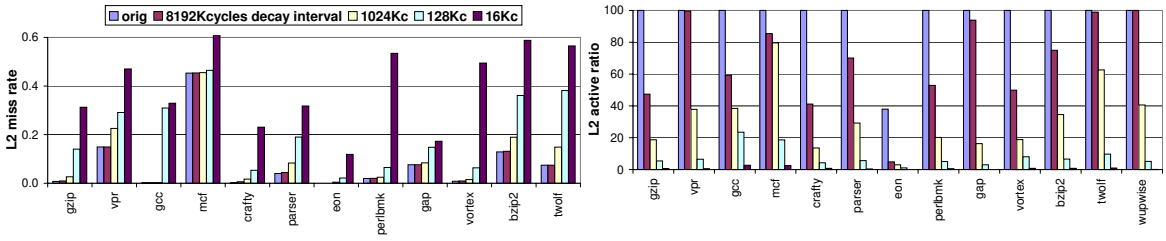

Figure 3.16: Miss rate and active ratio of a 1MB L2 decay cache for SPECint 2000. eon does not fully utilize the L2 cache.

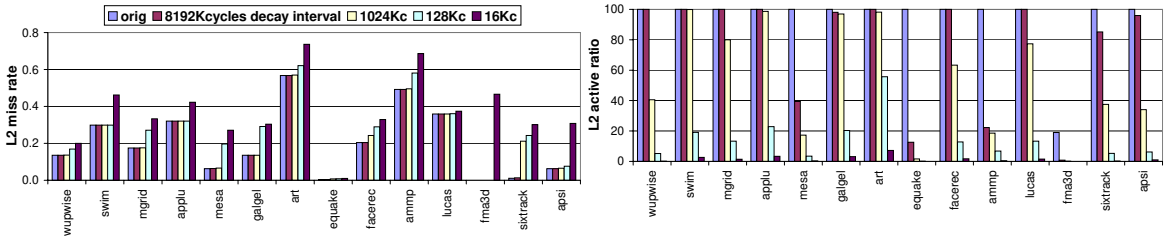

Figure 3.17: Miss rate and active ratio of a 1MB L2 decay cache for SPECfp 2000. fma3d does not fully utilize the L2 cache.

the L2.

Here, we assume that multilevel inclusion is preserved in the cache hierarchy [2]. Multilevel inclusion allows snooping on the lowest level tags only and simplifies writebacks and coherence protocols. *Inclusion bits* are used to indicate presence of a cache line in higher levels. For L2 cache lines that also reside in the L1 we can turn off only the data but not the tag. Figures 3.16 and 3.17 show miss rate and active ratio results for the 1MB L2 cache. As before, cache decay is quite effective at reducing the active ratio in the cache. Miss rates tend to be tolerable as long as one avoids very short decay intervals. (In this case, 128Kcycle is too short.)

It is natural to want to convert these miss rates and active ratios into energy estimates. This would require, however, coming up with estimates on the ratio of L2 leakage to the extra dynamic power of an induced L2 miss. This dynamic power is particularly hard to characterize since it would often require estimating power for an off-chip access to the next level of the hierarchy. Instead, we report the "breakeven" ratio. This is essentially the value

of *L2Access:leak* at which this scheme would break even for the L2 cache.

In these benchmarks, breakeven *L2Access:leak* ratios for an 1Mcycle decay interval range from 71 to 155,773 with an average of 2400. For a 128Kcycle decay interval, breakeven *L2Access:leak* ratios range from 16 to 59K with an average of 586. The art benchmark tends to have one of the lowest breakeven ratios; this is because its average L2 access interval is very close to its average L2 dead time so cache decay is very prone to inducing extra misses.

3.8.4 Multiprogramming

Our prior results all focus on a single application process using all of the cache. In many situations, however, the CPU will be time-shared and thus several applications will be sharing the cache. Multiprogramming can have several different effects on the data and policies we have presented. The key questions concern the impact of multiprogramming on the cache's dead times, live times, and active ratios.

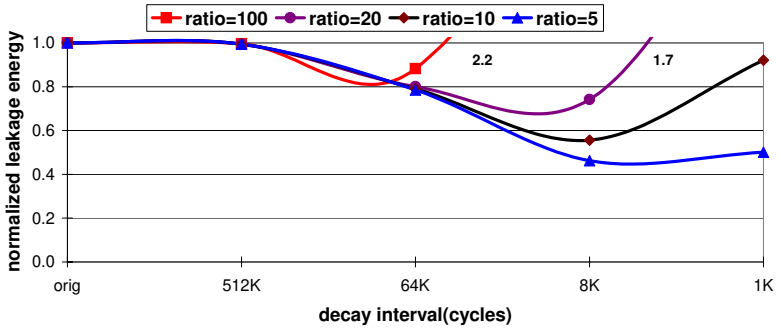

Figure 3.18: Normalized L1 data cache leakage energy for cache decay methods on a multiprogrammed workload.

To evaluate multiprogramming's impact on L1 cache decay effectiveness, we have done some studies of cache live/dead statistics for a multiprogramming workload. The workload was constructed as follows. We collected reference traces from six benchmarks individually: gcc, gzip, mgrid, swim, vpr and wupwise. In each trace, we recorded the address

referenced and the time at which each reference occurred. We then "sew" together pieces of the traces, with each benchmark getting 40ms time quanta in a round-robin rotation to approximate cache conflicts in a real system.

Multiprogramming retains the good behavior of the single-program runs. Average dead time remains roughly equal to the average dead times of the component applications. This is because the context switch interval is sufficiently coarse-grained that it does not impact many cache generations. In a workload where cache dead times are 14K cycles long on average, the context switch interval is many orders of magnitude larger. Thus, decay techniques remain effective for this workload. Multiprogramming also allows opportunities for more aggressive decay policies such as decaying items at the end of a process time quantum.

3.9 Related Work

Various researchers have noticed that a large percentage of blocks in a cache at any given time are never reused. Wood, Hill, and Kessler showed that for their benchmark suite dead time was typically at least 30% on average [77]. Similarly, Burger et al. showed that most of the data in a cache will not be used in the future [9]. They found cache "efficiencies" (their term for fraction of data that will be a read hit in the future before any evictions or writes) to be around 20% on average for their benchmarks. Most interestingly, they noted that fraction of dead time gets *worse* with higher miss rates, since lines spend more of their time about to be evicted.

Albonesi [1] proposed a cache design which can change the associativity of the cache according to application demands. Yang *et al.* [80] described a circuit technique called Gated-Vdd and an instruction cache organization that disables ways of a set-associative cache to match the capacity currently needed by the executing program. Similar to the

Cache Decay idea described in this chapter, Zhou *et al.* [82] describe techniques for disabling individual cache lines by inferring that lines which have not been used in a long time have *decayed*: they probably contain data that is not likely to be used again before replacement. Flautner *et al.* [19] have proposed a drowsy cache, which reduces static power consumption by putting cache lines in a low-power state where they retain their information but cannot be accessed. The authors propose putting the entire cache in the drowsy mode at periodic intervals. Nii *et al.* [56] proposed a low leakage circuit technique called Auto-Backgate-Controlled Multi-Threshold COMS (ABC-MTCMOS). With this technique, the threshold voltages of the transistors in the cell are dynamically increased when the cell is set to a low leakage mode by raising the source to body voltage in the circuit. Hanson *et al.* [24] compare various leakage reduction techniques for the L1 instruction cache, the L1 data cache, and the L2 unified cache. Heo *et al.* [26] propose to reduce the static energy associated with the bitlines in a RAM by simply tristating the drivers to the lines. The floating bitlines naturally settle at the voltage level that minimizes the leakage energy.

3.10 Chapter Summary

This chapter has described a novel method to reduce cache leakage power by keeping track of cache line idle times and exploiting generational characteristics of cache-line usage. We introduce the concept of cache decay, where individual cache lines are turned off (eliminating their leakage power) when they enter a dead period—the time between the last successful access and a line's eviction. We propose an energy-efficient technique that deduces entrance to the dead time with small error. Error in our technique translates into extraneous cache misses and writebacks which dissipate dynamic power and harm performance. Thus, our techniques must strike a balance between leakage power saved and dynamic power induced. Our evaluations span a range of assumed ratios between dynamic and static power,

in order to give both current and forward-looking predictions of cache decay's utility.

Our proposal of cache decay is a time-based working set algorithm over all cache lines. A cache line is kept on as long as it is re-accessed within a time window called "decay interval." In our implementation, a global counter provides a coarse time signal for small per-cache-line counters. Cache lines are "decayed" when a cache-line counter reaches its maximum value. This simple scheme works well for a wide range of applications, L1 and L2 cache sizes, and cache types (instruction, data). It also survives multiprogramming environments despite the increased occupancy of the cache. Compared to standard caches of various sizes, a decay cache offers better active size (for the same miss rate) or better miss rate (for the same active size) for all the cases we have examined.

Regulating a decay cache to achieve a desired balance between benefit and overhead is accomplished by adjusting the decay interval. Competitive on-line algorithm theories allow one to reason about appropriate decay intervals given a dynamic to static energy ratio. Specifically, competitive on-line algorithms teach us how to select a decay interval that bounds worst case behavior within a constant factor of an oracle scheme. To escape the burden of selecting an appropriate decay interval to optimize *average case behavior* for different situations (involving different applications, different cache architectures, and different power ratios), we propose adaptive decay algorithms that automatically converge to the desired behavior. The adaptive schemes involve selecting among a multitude of decay intervals per cache line and monitoring success (no extraneous misses) or failure (extraneous misses) for feedback.

Aside from the hardware-counter based decay mechanism, we also experimented a profile-based, rather than time-based, method to detect dead times [42]. By using a profile run to classify load/store instructions according to subsequent hit or miss events on the cache lines they access, one can further improve on timekeeping-based cache decay by eliminating the long wait times before lines are turned off.

With the increasing importance of leakage power in upcoming generations of CPUs, and the increasing size of on-chip memory, cache decay can be a useful architectural tool to save power or to rearrange the power budget within a chip.

From the viewpoint of the timekeeping methodology, cache decay is complementary to the timekeeping victim cache filter described in the previous chapter. More specifically, while the victim cache filter targets cache line generations with short dead times, cache decay mainly captures those with long dead times. Short dead times are good indicators of conflict misses, therefore they are good matches to conflict-oriented structures, such as the victim cache. On the other hand, long dead times contribute to the bulk of cache leakage consumption, and typically indicate cache lines that are likely no longer useful, so they are the natural target for leakage reduction techniques such as cache decay. Overall, both cache decay and the timekeeping victim cache filter demonstrate that, since timekeeping metrics are strongly predictive of program memory behavior, keeping track of them leads to effective hardware mechanisms for improving processor power and performance.

Chapter 4

Timekeeping Prefetch

In previous chapters, we described two applications of the timekeeping methodology: cache decay and a timekeeping victim cache filter. A common characteristic of these two mechanisms is that they both exploit lifetime characteristics within single generations. More specifically, cache decay tracks the idle time in the current generation and predicts whether the cache line is in its dead time. Timekeeping victim cache filter tracks the dead time of the current generation and predicts whether this generation is ended due to a conflict or capacity miss.

In contrast, the timekeeping prefetch mechanism, as will be described in this chapter, exploits lifetime characteristics across consecutive generations of the same cache line. This mechanism keeps track of timekeeping metrics in the previous generation of a cache line and uses them to deduce what will happen in the current generation. Specifically, we will show that there exists regularity among live times of consecutive generations of the same cache line. This means the live time in the previous generation can be a good estimate of the live time of the current generation. Since knowing the live time tells us when a cache line is dead, this naturally leads to a dead block predictor. In this chapter, we evaluate the effectiveness of such a predictor and we demonstrates how it can be used to construct a

highly effective hardware prefetcher.

4.1 Prefetching: Problem Overview

To attack the widening speed gap between processor and main memory, many computer architects have relied on prefetching mechanisms. Prefetching works by predicting what data will be required by the processor in the future and fetching them into caches *a priori*. Prefetching is somewhat similar to branch prediction, where addresses of to-be-executed instructions are predicted and associated instructions pre-loaded into the processor core. As in branch predictors, prefetching can be initiated in either hardware [4, 11, 34, 37, 38, 45, 62, 70] or software [47, 48, 54, 57]. Compared to software prefetching, hardware prefetchers have the advantage of transparency and run-time information availability. However, due to the lack of program semantic information, many hardware prefetchers have relied on capturing specific recurring patterns observed in memory reference streams. For example, stride prefetchers [4] target load instructions that stride through the address space. Stream buffers [38] attempt to capture reference streams formed by consecutive cache lines.

Correlation-based prefetching [11, 34, 37, 45, 70] is a more general prefetching scheme, attempting to exploit any correlation between a future memory reference and past memory behavior, including memory reference streams, load instruction addresses, and branch history. A common drawback in previous proposals on correlation-based prefetching is the relatively large size of their correlation tables, often 1-2 MB [37, 45]. These size requirements are comparable to current on-chip L2 caches and therefore bring up concerns about latency, power, and transistor budget overhead. Moreover, some prefetchers require instruction addresses, in addition to address traces. Passing information about instructions from the processor core to prefetchers complicates the processor design.

In general, prefetching can be thought of in terms of two sub-problems:

- Identifying when a cache line enters its dead time, so that it can be evicted to make room for prefetched data.
- Identifying which new block should be prefetched.

In this chapter, we demonstrate a highly effective prefetcher that can be constructed by integrating a tag-history-based prefetch address predictor and a live-time-history-based dead block predictor. We follow a sequence of steps to establish this claim:

- First, we show in Section 4.2 that *tags* and *tag sequences*, the necessary ingredient for tag correlating prefetching, are highly repetitive and thus form a solid basis for address predictions.
- Second, we show in Section 4.3 that live time across consecutive generations of the same cache line exhibit strong regularity. This leads to a live-time-history-based dead block predictor.
- Third, we show in Section 4.4 that a small, integrated history table can track both tag correlation history and live time history. Finally, in Section 4.5 we evaluate a timekeeping prefetcher based on such a history table.

4.2 Tag-History-Based Address Predictor

In this section we present the rationale behind our tag-based address predictor. We start from the well-known locality of memory references: programs tend to access addresses that match or are close to previously-accessed addresses. Traditionally memory reference locality has been interpreted in terms of complete addresses. Since cache tags are just the high order bits of memory addresses, intuitively the rule of locality should also apply. (Note that locality of tags are in accordance with locality found for virtual pages [3] and

TLB [64, 40]). In the following paragraphs we further formalize the locality of tags, with formula "$A \rightarrow B$" representing the relationship that "if A appeared in the recent past, then B will likely appear in the near future".

- Temporal locality states that recently accessed addresses are likely to be re-accessed in the near future. When re-references to an address occur, the corresponding tag and index will also re-appear. So temporal locality indicates that cache tags tend to recur within the same cache set. This line of thought can be represented by the following formula.

  $A \rightarrow A$

  $\Rightarrow tag(A) \rightarrow tag(A) \;\; and \;\; index(A) \rightarrow index(A)$

- Spatial locality says that items whose addresses are near each other tend to be referenced close in time. This correlation can be represented as follows:

  $A \rightarrow A + \delta$

  Depending on the relative size of δ, three situations could occur:

  1. $tag(A) = tag(A + \delta) \;\; and \;\; index(A) = index(A + \delta)$.

     This happens when δ is so small that A and $A + \delta$ remain in the same cache line. In this situation $tag(A)$ re-appears in the same set, accompanying the occurrence of $A + \delta$.

  2. $tag(A) = tag(A + \delta) \;\; but \;\; index(A) \neq index(A + \delta)$.

     This happens if δ is big enough to change the index but not enough to affect the tag. In this situation $tag(A)$ re-appears in another cache set when $A + \delta$ is referenced.

  3. $tag(A) \neq tag(A + \delta)$.

     This happens if δ is big enough to change the tag. In this situation $tag(A)$ will not re-appear when $A + \delta$ is referenced.

Spatial locality typically refers to addresses that are near each other, therefore δ is usually small enough so that the third situation rarely occurs. Combining situation 1 and 2, we can interpret spatial locality as: "cache tags tend to re-appear either in the same cache set, or in other cache sets." This interpretation also applies to temporal locality, where cache tags recur only in the same cache set. Thus, for both temporal and spatial locality:

$$A \rightarrow A \;\; or \;\; A \rightarrow A + \delta$$
$$\Rightarrow tag(A) \rightarrow tag(A)$$

Overall, tags exhibit recurring behavior due to the locality of references. Tags can be viewed as special per-cache-set tag sequences, with sequence length of 1. The question we want to investigate next is whether general tag sequences also exhibit recurring behavior. That is, is the following formula true?

$tag(A1), tag(A2), ..., tag(Ak) \rightarrow tag(A1), tag(A2), ..., tag(Ak)$

To test this, we profiled the tag sequences in SPEC2000 benchmarks. Without loss of generality, we look at sequences with length of three. The left graph of Figure 4.1 shows the number of unique three-tag sequences that appeared in the miss streams of a 32 KB direct-mapped L1 data cache. The right graph of Figure 4.1 gives the average number of times each three-tag sequence recurs. As we can see, for many benchmarks, each three-tag sequence recurs thousands of times, indicating a very repetitive behavior that can be exploited by a history-based predictor.

Since many tag sequences do exhibit recurring behavior, if a tag sequence of (A, B, C) appeared in the past, then we expect it to repeat itself in the future. Therefore, whenever we observed sequence of (A, B), we can predict C as the next tag. We utilize such an address predictor in our timekeeping prefetch and we evaluate its effectiveness in Section 4.5.

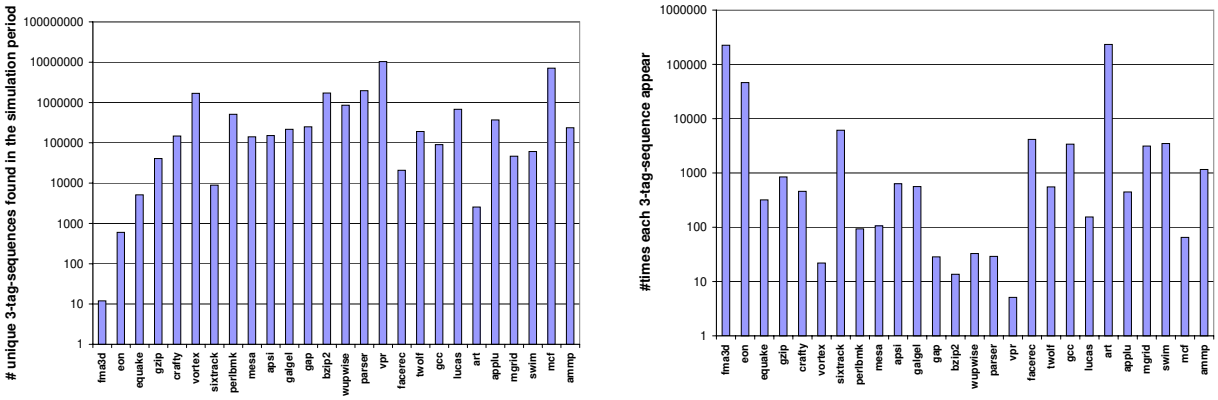

Figure 4.1: Number of unique three-tag sequences (left) and average number of times each sequence appears (right) for SPEC2000 benchmarks.

4.3 Live-Time-Based Dead Block Predictor

Now that we have addressed the address prediction, in this section we explain how to decide when to initiate a prefetch so that it will arrive on time, but without disrupting any useful data. Ideally, a new cache line should arrive right after the live time of the current cache line has elapsed, therefore a live time predictor can greatly facilitate the scheduling of a prefetch. As with other predictors, past history is our guide in predicting the live time of a block. The simplest history-based predictor is to predict that the live time of a block in the current generation will be the same as the live time of its previous generation. This works if there exists regularity between live times of consecutive generations of the same cache line. To test this, we profiled variability of consecutive live times per block using counters with a resolution of 16 cycles. Figure 4.2 shows the profiling results for selected SPEC2000 programs and for the geometric mean for all SPEC2000 programs. The benchmarks shown have significant speedup potential and will be discussed in detail in Section 4.5. The figure shows that a significant percentage (more than 20%) of the differences are less than 16 cycles, indicating that for many cache lines, their live times stay relatively stable across consecutive generations.

By exploiting this regularity of the live times we can construct a predictor for dead

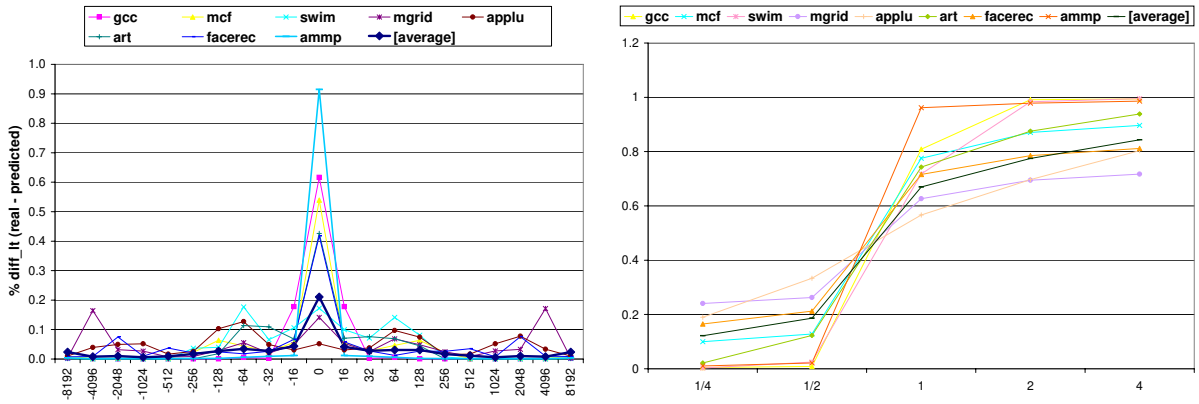

Figure 4.2: Absolute difference (left) and cumulative distribution of the relative ratio (right) of consecutive live times.

blocks as follows: at the start of a block's generation we predict that its live time is going to be similar to its last live time. After its predicted live time is over, we wait a brief interval and then we predict the block to be dead. The question is: how long should we wait before predicting the block is dead? To account for some variability in the live time we could add a fixed number of cycles to the predicted live time. Because live times have a wide range in magnitude, however, we chose instead to scale the added time to the predicted live time. To choose an appropriate scaling factor, the second graph of Figure 4.2 shows the cumulative distribution of the ratio of the current live time divided by the previous live time. As we can see in this graph, on average, about 80% of the current live times are less than twice the previous live time.

Thus, a simple heuristic is to predict that a block is dead at a time twice its previous live time from the start of its current generation. Additional justification for this predictor comes from our observation in Chapter 2 that dead times are typically much larger than live times. Using this predictor, Figure 4.3 shows the accuracy and coverage for the SPEC2000 programs. Coverage in this case refers to the percentage of blocks for which we do make a prediction while they are still in the cache. Blocks with a shorter *generation* than twice their predicted live time have already been evicted by the time of the prediction. On average

(for all SPEC2000), accuracy is around 75% and coverage about 70%. There is also a discernible trend for increased accuracy and coverage to the right of the graph towards the programs with significant percentage of capacity misses and significant potential for speedup.

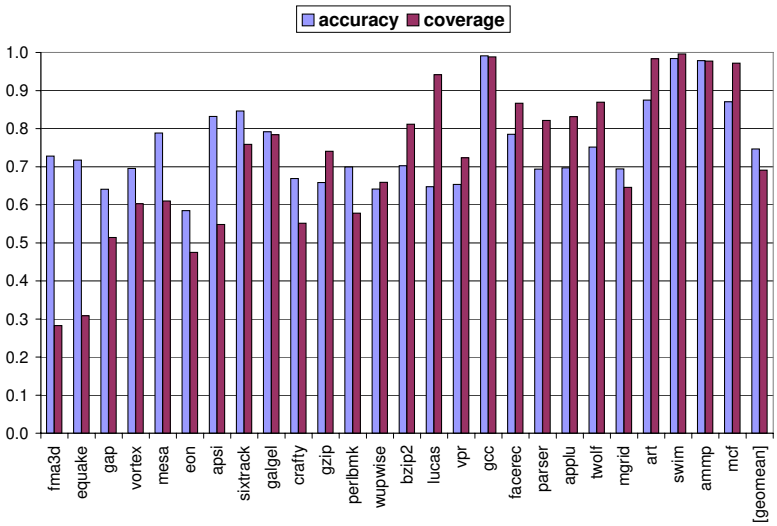

Figure 4.3: Accuracy and coverage of live time based dead-block prediction. Benchmarks are order from left to right according to their performance speedups with an ideal L1 data cache with no conflict or capacity misses.

4.4 Timekeeping Prefetch: Implementation

A full-fledged prefetching mechanism needs to establish both what to prefetch and when to prefetch it. Regarding what to prefetch, a tag-history-based predictor can help to provide accurate address prediction, as discussed in Section 4.2. Regarding when to prefetch, a live-time dead-block prediction is an efficient mechanism to schedule prefetches but this also requires a predictor structure to predict live times.

In this section we show that the *same* structure that can predict addresses can also predict live-times (or vice-versa), unifying the two predictors into a single structure. We propose a compact, history-based, predictor for both addresses and live-times that outper-

forms previous proposals for the whole of the SPEC2000. Furthermore, it requires only a tiny fraction of the area compared to prior proposals: about two orders of magnitude smaller than the Dead Block Correlating Predictor (DBCP) proposed by Lai *et al* in [45].

Our predictor is a correlation table not unlike the DBCP. In our case, the reference history used by our predictor is just the most recent miss address per frame, which is readily available in L1. In contrast, the DBCP approach also requires a PC trace which, in many cases, is complex to obtain from within the out-of-order core. We use miss address per cache frame, rather than the global miss trace of the cache. This means that a prediction that refers to a specific frame takes into account only the miss trace of this frame. The issue is complicated somewhat in set-associative caches where we use *per-set* miss trace history but we still perform all timekeeping and accounting on a per-frame basis. Per-set miss trace history removes some of the conflict misses that are dispersed within the history of the capacity misses. This is an advantage for our prefetching mechanism since it caters mostly to capacity misses as we will show later in this section.

Each predictor entry stores both the prediction for the prefetch address and the predicted live time of the block to be replaced by the prefetch. We use a 1-miss history to get these predictions. For example, assume that block A occupies a cache frame. At the point when block B replaces A, we access the predictor using the (per-frame) history (A,B). The predictor returns a prediction that block C should replace B *and* a prediction for the live time of B. Using the predicted live time of B, we apply our live-time-based dead-block prediction and we "declare" B to be dead at a time twice its predicted live time. At that point in time, we schedule the prefetch to C to occur. (One could also estimate when C needs to arrive, and exploit any slack to save power or smooth out bus contention.)

Figure 4.4 shows how we access the address correlation table and in particular the indexing mechanism we use. When block B replaces block A in a cache frame we add the *tags* of A and B (using truncated addition). When combined with A and B's common index,

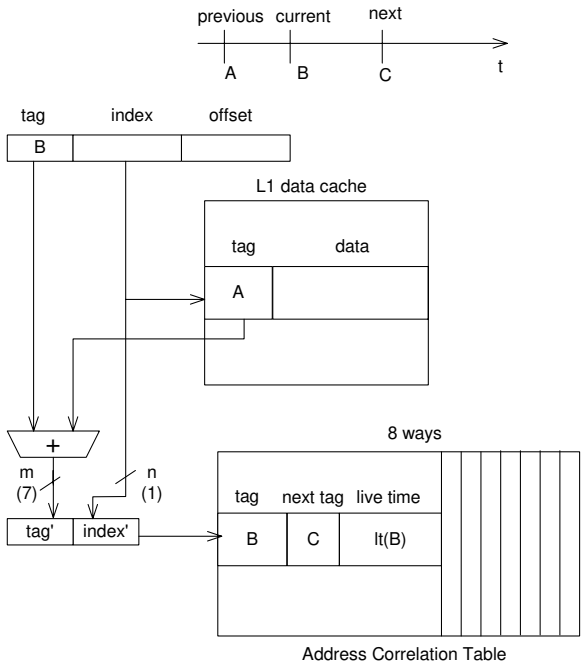

Figure 4.4: Structure of Timekeeping Address Correlation Table. Assuming a tag sequence of (A, B, C) in the current cache frame.

the sum of the tags gives us a pointer to the correlation table. The pointer is constructed by taking *m* bits from the sum of the tags and *n* bits from the index. The correlation table is typically set-associative so the pointer selects a set in this table. We then select the correct entry in the set by matching the tag of the block B to the identification tag in the predictor entry. The selected predictor entry predicts the *tag* of the block to be prefetched. The index is implied and is the same as in A and B. The same entry also gives a prediction for the live time of B. This live-time prediction is at the crux of our ability to do timely prefetch.

We have tested several sizes of this table ranging from megabytes to few kilobytes. Even very small tables work surprisingly well. An interesting observation arises when we index this table using mainly tag information and only partial index information (*n* less than 10). In this case, histories from different cache frames (or sets) may map to the same entry. This results in *constructive* aliasing and allows our table to have much smaller size than the table in [45]. The intuition behind this constructive aliasing is that often multiple distinct

data structures are traversed similarly. If accesses in one data structure imply accesses to another data structure, it does not matter what particular element is accessed in the one or the other. A contrived and simplistic example is to imagine a loop that adds elements of two arrays and stores them in a third array. Many triads of elements accessed can share a single entry in the correlation table as long as their tags remain the same! They all exhibit the same tag sequence pattern.

Within every cache line, we need the following timekeeping hardware for our prefetch: two counters, a register, and two extra tag fields. The two counters and the register are only 5 bits long each. The results we present in this chapter are for an 8KB table, 8-way set-associative correlation table. We index the table using seven bits from the sum of tags (m=7) and one bit from the cache index (n=1).

One counter (gt_counter) and one register (lt_register) are needed to track live time as follows: The counter measures generation time. It is initialized at the beginning of a generation and is continuously incremented by the global tick until the next miss. At this point the counter contains the generation time. At every intermediate hit the gt_counter is copied over to the lt_register so at any point in time the lt_register trails the gt_counter by one access. Thus, when a generation ends, the value of the lt_register is the time from the initial miss to the last access, i.e., the live time. An additional counter (prefetch_counter) and a tag field (next_tag) are needed to schedule a prefetch while another tag (prev_tag) is needed for predictor update as discussed below.

Figure 4.5 and 4.6 show the overall structure and operations associated with timekeeping prefetch. The correlation table sits besides the L1 cache. Assume we have the following sequence of blocks in a cache frame: (D, A, B, C). When a miss on address B attempts to replace block A, the lt_counter contains the live time of A, the prev_tag contains D, and the following actions occur:

1. A demand fetch is sent to L2 for block B.

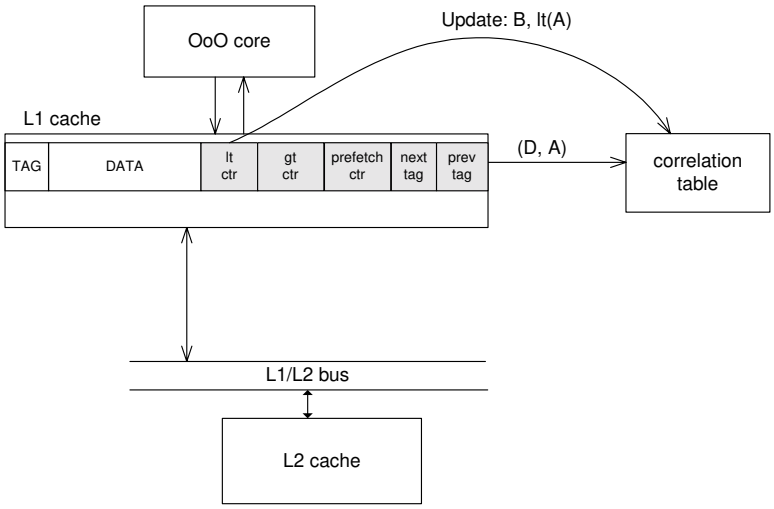

Figure 4.5: Update of a predictor entry.

2. Predictor update occurs as shown in Figure 4.5. An index is computed from A and its precursor D. The predictor table is accessed with history (D,A) and the entry corresponding to A is updated with B as the predicted next tag and lt(A) as the next prediction for the live time of A. A is installed in prev_tag.

3. Predictor access occurs as shown in Figure 4.6. An index is computed from B and the just evicted block A. The predictor is accessed with history (A,B) and an entry is selected that corresponds to B. Predictions for the live time of B and the next tag C are obtained and installed in the prefetch_counter and the next_tag respectively. (The live time is doubled by shifting one bit before it is installed in the prefetch counter.)

4. The prefetch counter is decremented with every tick. When it reaches zero the prefetch to C is put into an 128-entry prefetch queue.

4.5 Simulation Results

Figure 4.7 shows the IPC improvement of timekeeping prefetch over the base configuration. We include results for our 8KB timekeeping correlation table and we compare to the

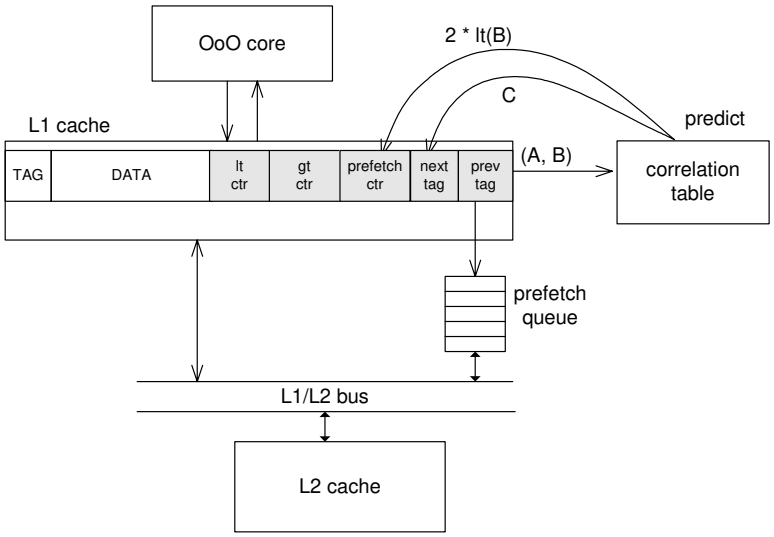

Figure 4.6: Predictor access to make a prediction (right).

prior proposed DBCP with a 2MB table. DBCP [45] is a correlation based prefetcher that correlates the liveness of a cache line and the next tag with PCs of previous memory instructions, in addition to addresses. Note that in [45], a critical miss predictor [18, 72] is proposed to filter the correlation entries. In our experiment, this filter is not incorporated in either DBCP or timekeeping prefetch. Our timekeeping prefetch achieves higher IPC improvement than DBCP in all SPEC2000 benchmarks except mcf and ammp (these benchmarks benefit greatly from a large history table, as explained later.) In addition, it improves performance for all but four of the SPEC2000 programs. Overall, our prefetch mechanism achieved 11% IPC improvement while DBCP only achieved about 7% improvement. Referring back to Figure 2.7, we see that our prefetching mechanism achieves significant speedup for many of the programs with a very large percentage of capacity misses (programs to the right of the graph in Figures 2.7 and 4.7) without harming those heavy on conflict misses (programs to the left of the graphs). The best performers are gcc, facerec, applu, mgrid, art, swim, ammp, and mcf. From the programs with the highest potential for speedup only two, twolf and parser, do not benefit from prefetch. These two programs exhibit very low accuracy in address prediction which results in a slight performance loss

for twolf; the same programs are problematic even with a 2MB DBCP.

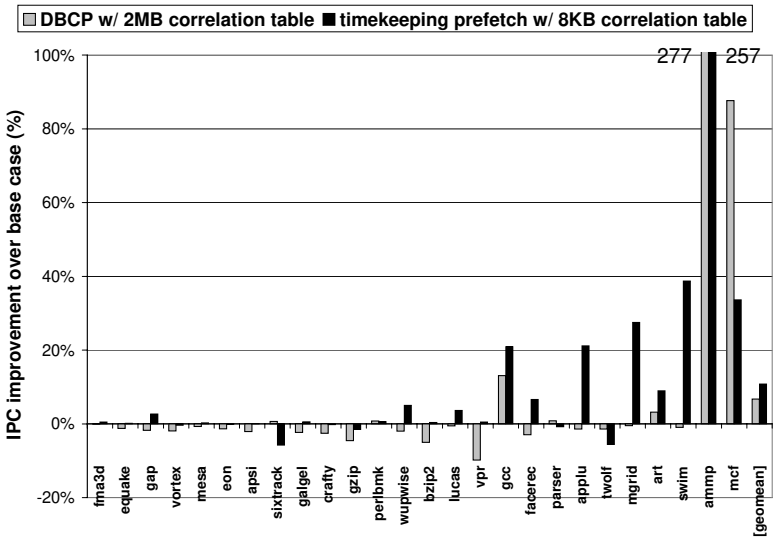

Figure 4.7: IPC improvement using timekeeping prefetch vs. DBCP prefetch

Considering only the eight best performers we see that, although the achieved speedups are significant, there is still room for improvement when compared to the ideal case. The differences are explained by close examination of the accuracy and coverage of our address prediction and the *timeliness* of our prefetches. Figure 4.8 shows the address accuracy and coverage for our 8KB address correlation table. Coverage in this case refers to the hit rate of the predictor; if we miss in the predictor we cannot make an address prediction.

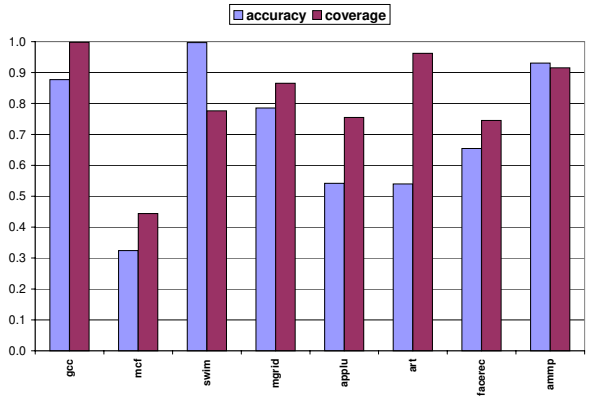

Figure 4.8: Address accuracy of the 8KB correlation table for the eight best performers

Figure 4.9 classifies the timeliness of the prefetches for the correct and wrong address

predictions. Each bar (from bottom to top) shows prefetches that are:

- "early," arrived early and displaced the current *live* block
- "discarded," thrown out of the 128-entry prefetch queue before been issued to the L2
- "timely," arrived within the dead time and before the next miss
- "started_but_not_timely," issued, but arrived late (after the next miss)
- "not_started," did not even issue before the next miss

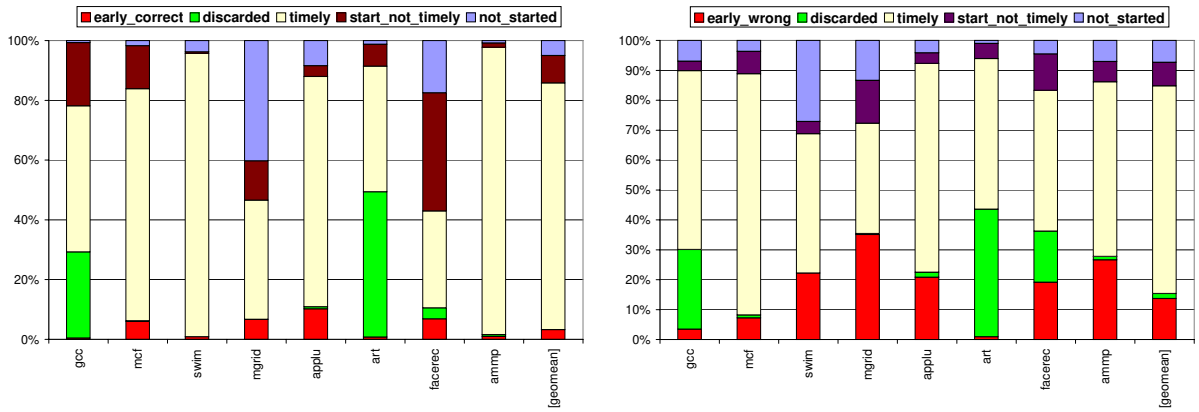

Figure 4.9: Timeliness of prefetches for correct (left) and wrong (right) address predictions

From the information of Figures 4.8 and 4.9 we can deduce the following. For two programs mgrid and facerec, while their address accuracy and coverage are fair, only 40% and 30% respectively of their correct prefetches are timely, while most of their other prefetches are late. This is because these two programs have short generation times and it is difficult to pinpoint their dead times. Two programs, art and to a lesser extent gcc, have a lot of discarded prefetches because of burstiness. This was also observed in DBCP prefetching. In addition art suffers from low accuracy in predicting the prefetch address. The reason why mcf does not achieve its full potential is also because of its low address accuracy. This program benefits from very large address correlation tables and this is the reason why it is

doing well with a 2MB DBCP. We observed better performance for mcf with our timekeeping prefetch when using a larger address correlation table. Finally, ammp, which speeds up by 257% — almost all of its potential — shows very good address accuracy and coverage and in addition shows very timely prefetches. As a general observation, the timeliness of our prefetches, especially with respect to earliness, correlates well with the accuracy of the address prediction: when we predict addresses correctly, we tend not to displace live blocks (Figure 4.9).

4.6 Related Work

Software prefetching, and more generally, compile-time analysis of memory access behavior, has been studied by many researchers [20, 47, 48, 54, 57]. Mowry *et al* successfully predicts what data references will likely miss in scientific codes that mainly employ matrices [54]. Ghosh *et al* describes methods for generating and solving equations that give a detailed representation of cache misses in loop-oriented scientific code [20]. Such a framework can be utilized to decide what addresses should be prefetched and when to start the prefetches. Other work target pointer-intensive applications and applications with recursive data structure and propose to insert compile-time prefetch instructions [47, 48, 57].

Compared to software prefetching, hardware prefetching [4, 11, 37, 34, 38, 45, 62, 70] usually requires extra hardware to track correlations between memory references with previous memory references and other information, such as memory instruction addresses and branch history. Baer and Chen proposed a early notion of correlation-based hardware prefetching for paged virtual-memory systems [3]. They also investigated a prefetching mechanism that captures load instructions that have constant strides [4]. Jouppi proposed a stream buffer that can be effective when there is a large amount of sequentiality in the reference stream [38]. Charney and Reeves were the first to propose a generalized correlation-

based hardware prefetching for caches [11]. In their scheme, the prefetcher is positioned between L1 and L2, and prefetches to L2 only. Joseph and Grunwald proposed a markov model for prefetching and proposed to store multiple targets with each prediction [37]. Lai *et al* were the first to propose a hardware predictor for dead blocks based on both PC traces and previous memory addresses [45]. They were also the first to propose prefetching according to per-cache-set memory reference behavior. Solihin *et al* proposed to use a user level thread for prefetching and store the correlation history in memory, instead of specific hardware tables [70].

4.7 Chapter Summary

In this chapter, we described a timekeeping prefetch scheme which tackles the problem of predicting when a block is dead and prefetches another block in anticipation of the next miss. As an application of the timekeeping methodology, timekeeping prefetch demonstrates how to exploit the lifetime regularities across consecutive generations of the same cache line. A key contribution in the work is the discovery that live times, when examined on a per-cache-frame basis, exhibit regularity. This enables predicting the live time of the current block based on its previous live times. Such a predictor allows us to schedule a prefetch to take place shortly after the block "dies." To implement a timekeeping prefetch we need both an tag-based address predictor and a live-time based dead-block predictor. We propose a novel history-based predictor that provides both predictions simultaneously. Our predictor is a correlation table accessed using the history of the previous and current misses in a frame. It predicts the live time of the current block, and the address to prefetch next. Because we index this predictor using mostly tag information we observe significant *constructive aliasing* both for addresses and live times. This allows us to outperform a 2MB DBCP predictor [45] using just 8KB of predictor state for all SPEC2000, with an

average IPC improvement of 11% over the base configuration.

Chapter 5

Timekeeping in Branch Predictors [1]

In previous chapters we show three applications of the timekeeping methodology in the memory system. The timekeeping methodology can be applied to other structures on-chip. As an example, in this chapter, we demonstrate an example of how it can be applied to branch predictors.

5.1 Introduction

In Chapter 3, we introduced cache decay, a timekeeping mechanism to attack the problem of increasing leakage power consumption. After caches, branch predictors are among the largest and most power-consuming array structures in current CPUs. Current predictors are 4–8 Kbytes in size, already the size of a small cache. They dissipate about 10% of the processor's total dynamic power dissipation [58]. Cycle-time, power-dissipation, and thermal concerns tend to keep predictors from growing larger. However, Jiménez *et al.* [36] pointed out that two-level predictors can avoid cycle-time constraints and that large second-level predictors can give substantial increases in prediction accuracy, resulting in predictors that

[1] The research presented in this chapter is a joint work by myself, Philo Juang, who is a fellow graduate student in our group, as well as other researchers from Princeton University, Agere Systems, and University of Virginia.

could be as large and have the same substantial leakage as first-level caches. Furthermore, Skadron *et al.* [67] found that the branch predictor is a thermal hot spot, as it is typically accessed every cycle. This indicates that branch predictors expend much more leakage energy than their sizes would suggest, because leakage power increases exponentially with temperature.

Applying decay techniques to caches has proven effective, so applying decay techniques to branch predictors is an obvious next step. Unfortunately, several factors complicate this task, for it is much less obvious when a branch predictor entry may be considered "dead" and can therefore be turned off with little performance impact.

- First, many branches may map to the same predictor entry. Since this sharing is sometimes beneficial, notions of cache conflicts and eviction do not translate directly into the branch prediction world.

- Second, a branch predictor entry is not simply valid or invalid, as in a cache. A branch predictor entry may have reached the "strongly not taken" state due to the effects of several different branches and may be useful to the next branch that accesses it, even if this branch has never been executed before.

- Third, branch predictor entries are too small to deactivate individually, so one must consider some larger collection, such as a row of predictor entries in the square array in which the predictor is likely implemented. The challenge here is that unlike the grouping of data into a cache line, the grouping of branch predictor entries in a row is not something for which application programmers and systems builders have a sense of spatial locality. This chapter evaluates design options related to these questions.

Further interesting questions arise when moving from simple *bimodal* branch predictors [69], which keep one two-bit counter per predictor entry, to multi-table predictors like

hybrid predictors [51], which operate several prediction structures in parallel. For example, hybrid predictors may encounter instances when one of the predictor components has decayed but the other has not. The chooser might be designed to pick the non-decayed component in such situations. For other branches, the chooser may exhibit a strong bias for one predictor component over the other. In this case, predictor entries that are not being selected might be deactivated.

A key characteristic of branch predictor contents is that they are both "transient" and "predictive". Transient means that they tend to be short-lived. Predictive means they are program hints so losing them does not affect the correctness of the execution. To exploit this characteristic we propose a more area-efficient and power-efficient implementation of branch predictor decay, using quasi-static 4-transistor (4T) RAM cells. 4T-based branch predictors can decay naturally, without extra hardware for tracking the idle time, or explicit control for gating the Vdd. Such a design can reduce leakage energy by 60-80%, with a cell area savings up to 33%.

5.2 Branch Predictors Studied

Although a wealth of dynamic branch predictors have been proposed, we focus on the effects of decay for a representative sample of predictor types: bimodal, gshare, and hybrid.

The bimodal predictor [69] consists of a simple *pattern history table* (PHT) of saturating two-bit counters, indexed by branch PC. This means that all dynamic executions of a particular branch site (a "static" branch) map to the same PHT entry, and means that there are never more PHT entries in use at any one time than there are active branch sites. This chapter models a 4 K-entry (8 Kbit) bimodal predictor. This is the configuration that appears in the Alpha 21064 [16], although the 21064 uses one-bit rather than two-bit counters. The Alpha 21164 [17] used a larger PHT of 8 K entries, but we conservatively choose

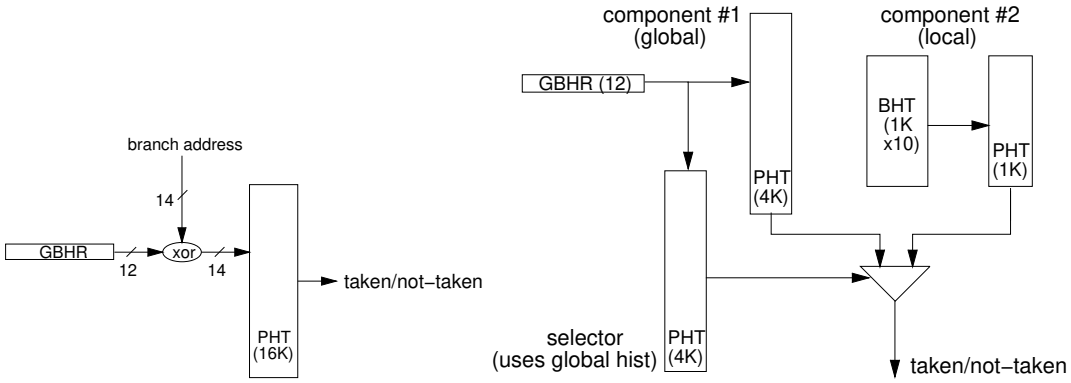

Figure 5.1: Gshare predictor in the Sun UltraSPARC-III (left) and 21264-style hybrid predictor (right).

the smaller PHT to make it more difficult to show benefits from decay.

The gshare predictor, shown in the left-hand portion of Figure 5.1, tries to detect and predict sequences of correlated branches by tracking a global history (the global branch history register or GBHR) of the outcomes of the N most recent branches. In gshare, the global branch history and the branch address are XOR'd to reduce aliasing. This chapter models a 16 K-entry gshare predictor in which 12 bits of history are XOR'd with 14 bits of branch address. This is the configuration that appears in the Sun UltraSPARC-III [71].

Instead of using global history, a two-level predictor can track local branch history on a per-branch basis. Local history is effective at exposing patterns in the behavior of individual branches. Because most programs have some branches that perform better with global history and others that perform better with local history, a hybrid predictor [10, 51] combines the two. It operates two independent branch predictor components in parallel and uses a third predictor—the *selector* or *chooser*—to learn for each branch which of the components is more accurate and selects its prediction. This chapter models a hybrid predictor with a 4K-entry selector that only uses 12 bits of global history to index its PHT; a global-history component predictor of the same configuration; and a local history predictor with a 1 K-entry, 10-bit wide BHT and a 1 K-entry PHT. This configuration appears in the Alpha 21264 [22] and is depicted in the right-hand portion of Figure 5.1.

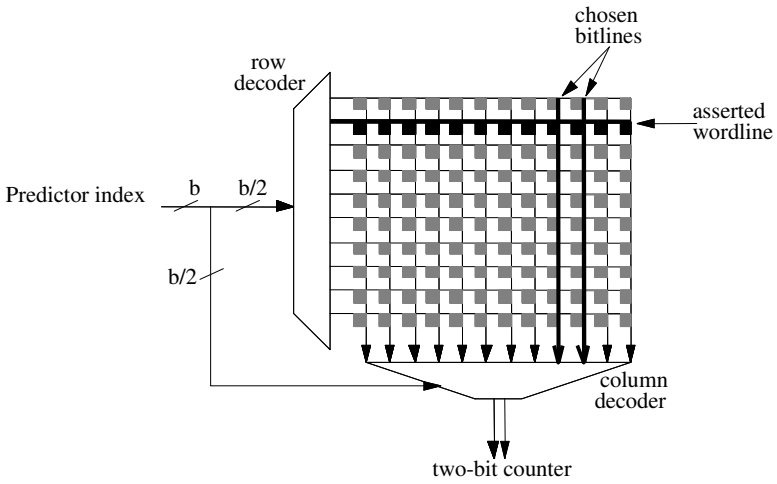

Figure 5.2: A schematic of a squarified branch predictor table of two-bit counters (the pattern history table).

Logically, branch predictors are arrays of counters that are typically just two bits wide. Physically, however, branch predictors are, like caches, implemented as square or nearly-square array structures, as shown in Figure 5.2. This helps to minimize the complexity of the row and column decoders and balance wordline and bitline length and delay. The predictor array is thus similar to a cache array, except that it needs no tags. For example, the 16K-entry gshare predictor discussed above can be laid out as a 128×128 array of 2-bit counters. Alternatively, it can be divided into 4 banks, each a 64×64 counter array. We refer to these two organizations as "unbanked" and "banked" respectively and will discuss their decay behavior in Section 5.5. Since branch counters are only 2 bits in size, a cost-effective choice for turning off these counters is at the granularity of rows in the array structure rather than individual entries.

5.3 Simulation Model Parameters

In this chapter we model a processor with microarchitectural parameters that most closely resemble the Intel PIII processor [15]. The main processor and memory hierarchy param-

Processor Core	
Instruction Window	16-RUU, 8-LSQ
Issue width	4 instructions per cycle
Functional Units	4 IntALU,1 IntMult/Div, 4 FPALU,1 FPMult/Div, 2 MemPorts
Memory Hierarchy	
L1 D-cache Size	16KB, 4-way, 32B blocks
L1 I-cache Size	16KB, 1-way, 32B blocks
L2	Unified, 256KB, 4-way LRU, 64B blocks,6-cycle latency, WB
Memory	18 cycles
TLB Size	128-entry, 30-cycle miss penalty
Branch target buffer	2048-entry, 4-way

Table 5.1: Configuration of simulated processor for branch predictor decay.

eters are shown in Table 5.1. For performance estimates and behavioral statistics, we use SimpleScalar's *sim-outorder* simulator. For energy estimates, we use the Wattch simulator [7].

Our results are evaluated using benchmarks from the SPEC2000 suite [73]. The benchmarks are compiled and statically linked for the Alpha ISA using the Compaq Alpha compiler with SPEC *peak* settings and include all linked libraries. We skip the first billion instructions of each program to avoid unrepresentative behavior at the beginning of the program's execution. We then simulate 500M (committed) instructions using the reference input set. To ensure reproducible results for each benchmark across multiple simulations, simulations are conducted with SimpleScalar's EIO traces.

5.4 Spatial and Temporal Locality in Branch Predictors

The first question in exploring decay for branch predictors is to determine how often an entire row of branch predictor entries is likely to lie idle long enough for decay techniques to be effective. In today's machines, branch predictor rows typically include 32-256 counter entries. Fortunately, program branches are clustered rather than random, and across all the

predictor organizations we examine, our experiments consistently show that some rows have heavy activity while others are idle and can be deactivated.

Clearly, programs exhibit spatial locality in the instruction cache. Over a short period of time, only one or a few small contiguous regions of the program are likely to be active, so branch instructions are likely to be close in terms of their PC. This also translates into spatial locality in branch-predictor accesses. For branch predictors, spatial locality means that at any point in the program, active rows are likely to have many counters active and idle rows are likely to be entirely idle. This is most true for the bimodal predictor, which is indexed only by PC. Indeed, the probability that two successive conditional branches fall into the same row in a 4 K-entry bimodal predictor is greater than 40% for all our benchmarks.

Yet this is not useful if the active rows change rapidly, so temporal locality is also necessary. One immediate factor that creates temporal locality is the fact that many benchmarks have small static branch footprints (the number of unique branch instruction sites that are executed), as seen in Table 5.2. Decay will therefore clearly help bimodal predictors, because each static branch touches only one predictor entry and we know from the data in Table 5.2 that they are clustered.

Other predictor structures, however, may not do as well. With gshare, the branch address is XOR'd with the global branch history, so that one branch can touch many PHT entries. We evaluate decay for gshare predictors in the next section. Hybrid predictors use global- and local-history predictors as components, which brings more design choices. We explore the design space for hybrid predictors in section 5.6.

	1K cycles	10Kc	100Kc	1Mc	Overall
gzip	24	32	45	103	281
vpr	31	45	58	65	742
gcc	2	9	79	193	512
mcf	65	83	92	116	565
crafty	104	305	592	855	1701
parser	53	90	157	294	2265
eon	81	289	357	415	652
perlbmk	90	453	631	1112	1541
gap	62	281	325	576	745
vortex	124	502	1227	1642	1996
bzip2	22	33	45	56	460
twolf	48	210	300	334	351
wupwise	42	52	53	55	193
swim	3	6	11	15	687
mgrid	3	6	9	25	500
applu	1	2	4	7	579
mesa	83	114	139	267	697
galgel	2	6	8	10	508
art	2	2	5	18	109
equake	167	192	193	202	226
facerec	7	24	25	39	144
ammp	11	26	105	230	794
lucas	3	3	3	4	242
fma3d	80	450	452	465	499
sixtrack	39	49	55	99	734
apsi	14	85	117	125	342
geomean	20	46	70	109	529

Table 5.2: Average number of static branches touched every sample interval for SPEC2000. The rightmost column labeled "Overall" gives the static branch footprint for the whole simulation period.

5.5 Decay with Basic Branch Predictors

Our techniques have the following general structure. At regular intervals, all rows of predictor entries not been used during the interval are assumed to have *decayed* and are therefore *deactivated*. The interval, called the *decay interval*, is measured in processor cycles and is a critical parameter for these schemes. The shorter the interval, the more opportunities for rows to be deactivated but the more likely it is to deactivate rows prematurely and induce extra mispredictions. Intervals long enough to minimize extra mispredictions, on the other hand, result in the deactivation of fewer entries.

If a predictor lookup tries to access a decayed row, the predictor signals that a prediction

cannot be made; the row is re-activated and possibly initialized to some desired starting state; in the meantime, a default prediction is made. Upon activation, our experiments use a default of not taken and initialize all the counters to 01. Thus, subsequent branches using the re-activated line start in the weakly-not-taken state.

The *active ratio* in a particular experiment is the average percentage of predictor rows found to be active (not decayed); it is a proxy for the actual leakage energy consumed by the predictor. Of course shorter decay intervals yield smaller active ratios (and larger leakage energy savings), but performance may suffer, since useful predictor entries are sometimes deactivated. Exploring this power-performance tradeoff is a key objective of this chapter.

To evaluate the net effectiveness of decay for reducing leakage energy, we combine the reduced value of leakage energy that was observed with decay, and the overhead energy associated with the decay technique. We then compare this to the original value for leakage energy. For each of the predictor types we study, we present plots of *normalized leakage energy* for different decay intervals, where the basis for normalization is the original value for leakage energy. This approach for measuring the net reduction in leakage energy is similar to the techniques used in cache decay, as described in Chapter 3.

Figure 5.3 shows the geometric mean of the active ratios across the benchmarks for both banked and unbanked 16K-entry gshare predictors and, as a reference, the 4K-entry bimodal predictor. As expected, the active ratios are quite small (*i.e.*, good from a decay point of view) for the bimodal predictor. For gshare predictors, the active ratios are larger. Yet significant numbers of rows remain untouched. This indicates that even for predictor structures designed to smear branch addresses over many entries, decay-based techniques still show significant promise for addressing leakage concerns.

We include data in Figure 5.3 for a banked version of gshare. Breaking the predictor into banks makes the active ratio smaller (better for decay) by reducing the granularity over which activity is measured. Indeed, the active ratio for gshare is 15–35% smaller if

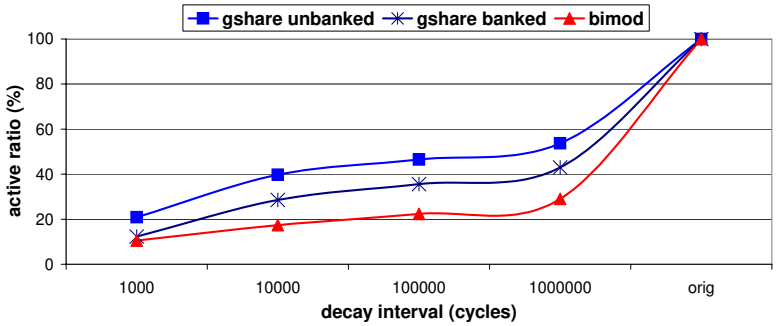

Figure 5.3: Mean active ratio for unbanked and banked gshare predictors and a bimodal predictor with different decay intervals. The rightmost label "orig" corresponds to non-decaying predictors.

it is broken into four banks of 4K entries each. Overall, these active ratios yield leakage energy savings of 40% for unbanked gshare and about 50% for banked gshare. Even greater savings can be achieved for structures directly indexed by PC: about 65% for bimodal predictors and 90% for the BTB [31].

5.6 Decay with Hybrid Predictors

With two competing components (the global component and the local component), hybrid predictors exhibit many interesting design choices when implementing decay. In this section we explore these design choices as well as their effect on decay in an Alpha 21264-style hybrid predictor.

Selection Policy The selection policy refers to the policy for choosing a prediction from one of the two component predictors. In a non-decaying 21264-style hybrid predictor, the chooser makes this decision using the global history; see Figure 5.1. However, when decay is enabled, it may happen that only one of the two components is active while the other is decayed. In this case, since the decayed component has lost its information, it is intuitively appealing to use the prediction from the active component, no matter what the chooser suggests. This policy is called "believe the active component", and is implemented in all

our experiments. It may also happen that both the two components are decayed, in which case all components are reactivated and the branch is predicted as "weakly not taken", as in bimodal and gshare predictors.

Wakeup Policy The wakeup policy refers to the decision of whether to reactivate a decayed row when it is accessed by a branch instruction. A naive policy would always wakeup any rows that are accessed. In a hybrid predictor, a more elegant policy is possible: the decayed component will be reactivated only if the chooser wants to select it. We refer to this policy as "believe the chooser".

In the situation when the accessed row in the chooser is decayed, we know that the corresponding row in the global component is also decayed in the 21264-style predictor. This is because the structural similarity between the chooser and global predictor: they are indexed, and thus decayed in exactly the same way; see Figure 5.1. In this case, the chooser has no useful information. If the local component is active, then we leave the chooser and the global component inactive and return the prediction from the local component. Otherwise (when the local component is also inactive), we reactivate all components and return a prediction of "weakly not taken".

Results Figure 5.4 and Figure 5.5 detail the active ratio and branch misprediction rate for naive decay, which always wakeup any rows that are accessed, with a 21264-style hybrid predictor. We see that even though the active ratios are higher than for bimodal or gshare predictors, decay has a negligible impact on the misprediction rate for intervals of 64K cycles or larger. Note that in order to compute active ratio sensibly on a multi-table structure, we compute it over all prediction and chooser bits in the structure. Overall, as Figure 5.6 shows, naive decay realizes strong reductions in energy savings—40% for a 64 K-cycle interval.

We can obtain even better energy savings by taking advantage of the "believe the

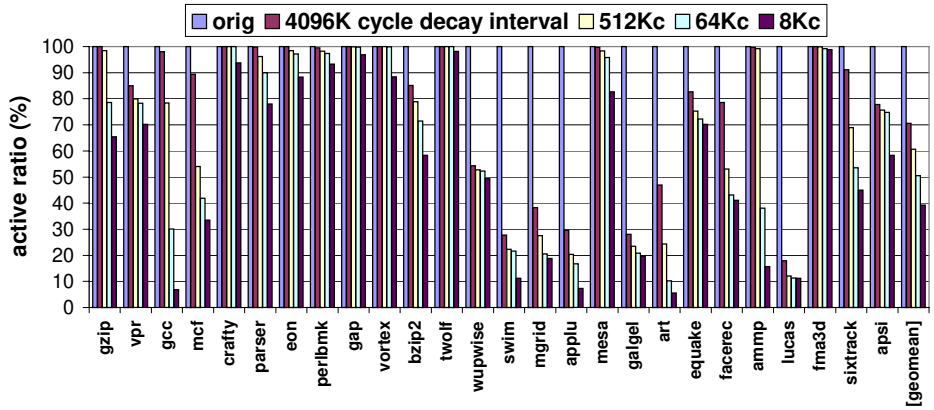

Figure 5.4: Active ratio for 21264's hybrid predictor.

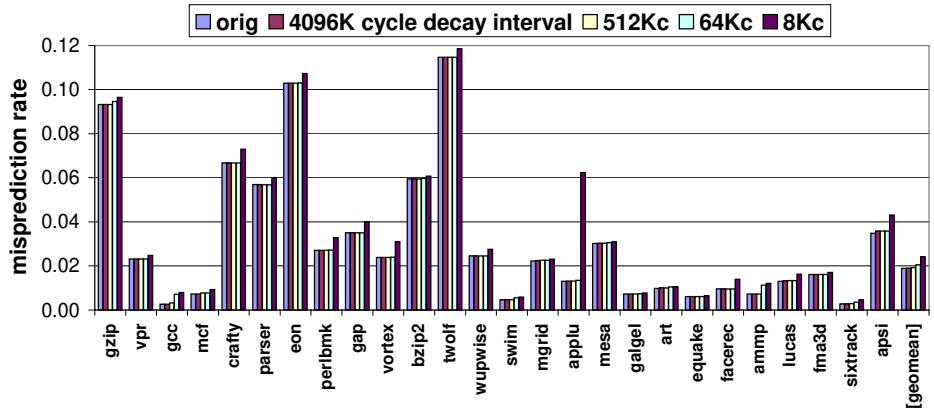

Figure 5.5: Misprediction rate for 21264's hybrid predictor.

chooser" wakeup policy. As shown in Figure 5.6, this more sophisticated policy leads to leakage power reductions about 50% better than the naive policy.

5.7 Branch Predictor Decay with Quasi-Static 4T Cells

The preceding sections demonstrated that decay techniques based on counters and standard 6T array structures could be successful in reducing branch predictor leakage energy. However, there is a downside to the counter-based techniques: there is a slight (5%) area overhead for implementing the idle time counters, and the gated-Vdd control for turning off the cache lines. This section examines a way of avoiding this hardware overhead by using quasi-static four-transistor (4T) memory cells for decay implementations. Quasi-static 4T

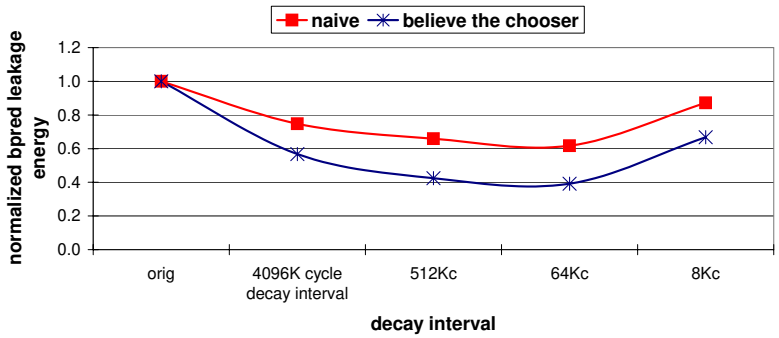

Figure 5.6: Normalized leakage energy of 21264-style Hybrid predictor with naive and "believe the chooser" wakeup policies.

memory cells have been mainly considered as a means of implementing DRAM within a logic fabrication process [65, 75]. In traditional uses, the perceived drawback of the 4T cells is that they are dynamic and need refresh; but this characteristic is actually the key for an elegant decay design. In contrast to previous 6T leakage control strategies, we do not have to turn off power to 4T cells. Instead, we let inactive cells decay naturally, thus avoiding any overhead associated with counting idle times or turning power on and off. Because of their use as embedded DRAM in some designs, 4T cells are already present in many design libraries, including those used by Agere Systems. We use the cells as they appear in the Agere library.

In addition, branch predictor contents are not architectural, meaning that if we unknowingly lose them, only performance might suffer but not correctness. This leads to a clean design without any decay counter hardware. The drawback in accessing decayed data is a potential bad prediction. As long as this is a rare event we can eliminate all the decay counter hardware and get similar benefits as in a 6T-based decay predictor. 4T cells are also smaller than 6T cells, thus offering area advantages too.

5.7.1 The Quasi-Static 4T Cell

Basic 4T RAM cells are well established and described in introductory VLSI textbooks [76]. 4T cells are similar to ordinary 6T cells but lack two transistors connected to Vdd that replenish the charge that is lost via leakage (Figure 5.7). Using exactly the same transistors as in an optimized 6T design a 4T cell requires only $2/3$ of the area compared to a 6T cell. 4T RAM cells naturally decay over time (without the need to switch them off); once they lose their charge they leak very little since there is no connection to Vdd.

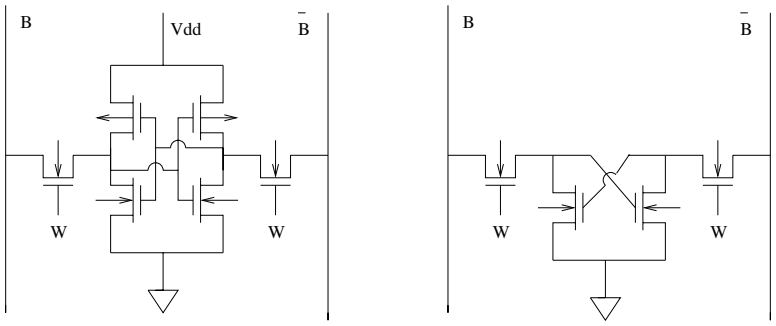

Figure 5.7: Circuit diagrams of the 6T SRAM cell (left) and the 4T quasi-static RAM cell (right).

Also importantly, 4T cells are automatically refreshed from the precharged bit lines whenever they are accessed. When a 4T cell is accessed, its internal high node is restored to high potential, refreshing the logical value stored in it; there is no need for a read-write cycle as in 1T DRAM. As the cell decays and leaks charge, the voltage difference of its internal nodes gradually drops to the point where the sense amplifiers cannot distinguish its logical value. Conservatively, this occurs when the node voltage differential drops below a threshold of the order of 100 mV (with 1.5V Vdd). Below this threshold we have a decayed state, where reading a 4T RAM cell may produce a random value —not necessarily a zero. Over a long time the cell reaches a steady state where both the high node and the low node of the cell "float" at about 30mV.

4T cells possess two characteristics fitting for decay: they are refreshed upon access

and decay over time if not accessed. In the rest of this section we discuss extensively the 4T decay design, including retention times, locality considerations, and simulation results.

5.7.2 Retention Times In 4T Cells

We define *retention time* to be the time from the last access to the time when the internal differential voltage of the cell drops below the detection threshold. Retention time depends on the leakage currents present in the 4T cell. Retention time is a critical parameter for a 4T design because when implemented in 4T cells, decay techniques have the cell's retention time as their natural decay interval.

To study retention times for the 4T branch predictor we chose the Agere COM2 0.16μ CMOS process for which we have accurate transistor models. Retention time is affected by the characteristics of the transistors themselves. For example, doubling the channel length and the gate oxide thickness can extend the retention time by lowering leakage currents. In contrast to standard 4T transistors, we refer to these transistors as slow-decay transistors.

The trade-off using slow-decay transistors is that the area advantage is reduced because RAM cells built upon these transistors are about $7/8$ of the 6T cell. Table 5.3 compares the three cell types for their access time and cell area.

	4T		6T
	standard	slow-decay	
access time (ps)	525	565	490
RAM cell area (relative)	0.66	0.88	1

Table 5.3: Comparison of three cell types: 4T standard, 4T slow-decay and 6T cells

Variations in temperature also result in large variations in retention times. Our designs target an operational temperature of $85°$C (appropriate for example for mobile processors) but we also discuss mechanisms to protect performance in situations where very high temperature ($125°$C) does not allow for sufficiently large retention times.

Based on these assumptions, we determine retention times for our technology through detailed transistor-level simulations. We simulate an access to a cell, followed by a long period in which the cell is left unread. During this time, leakage causes the cell's internal nodes to lose charge. Recall that the retention time is the duration between an access and the point at which the differential voltages of the 4T cells internal nodes lapsed to a value less than 100mV. We use 100mV as our criteria for the minimum voltage we would expect the sense amplifiers to distinguish. Reading a decayed cell produces a valid, though random, predictor value. (We model this randomness in our simulated results that follow. Table 5.4 gives the cell retention times in nanoseconds for the COM2 technology.

	standard			slow-decay		
	25C	85C	125C	25C	85C	125C
WCF	3.2K	0.44K	0.17K	264K	10.2K	1.94K
NOM	18K	1.7K	0.56K	1,040K	57.2K	9.4K
WCS	92K	8K	2K	1,480K	240K	38.4K

Table 5.4: Retention times in nanoseconds for standard and slow-decay versions of 4T cells at different operating temperatures. For a 1GHz (1ns cycle time) processor, one can also consider these retention times as cycle counts.

5.7.3 Locality Considerations

One of the main considerations in the 6T decay design is the granularity of decay. Our 6T designs operate with row granularity in order to reduce the counter overhead for decay. Granularity is also relevant in the 4T design but here it stems from the way 4T cells are refreshed. Branch predictors are typically laid out as a square, with each row having multiple neighboring predictors. In a squarified predictor, reading a row refreshes all the cells in a row because the wordline is asserted.

Retention time selection and locality granularity go together because large row granularities make the apparent rate of refresh much higher. Cells that would have decayed if left alone get refreshed coincidentally by nearby active cells. Thus 4T cells with short retention

times may not lose data as quickly if the row size is long enough. In contrast, in a design with very fine row granularity one would opt for 4T cells with very long retention times. Fine granularity leads to a very good decay ratio but the important cells must remain alive on their own (without the benefit of accidental refreshes) for considerable time.

5.7.4 Results for Decay Based on 4T Cells

We now examine the leakage and performance impact of branch predictor decay based on 4T structures. We considered a range of technologies, for this section, including COM2, COM3, and COM4. COM2 shows modest improvements with careful design, and future technologies improve significantly on this. We use slow-decay 4T cells in our design. As for the overall configuration, we use a 16K-entry gshare configuration as that appears in the Sun UltraSPARC-III [71]. We target an operational temperature of $85°$C; this leaves us a decay interval of 57,200 cycles.

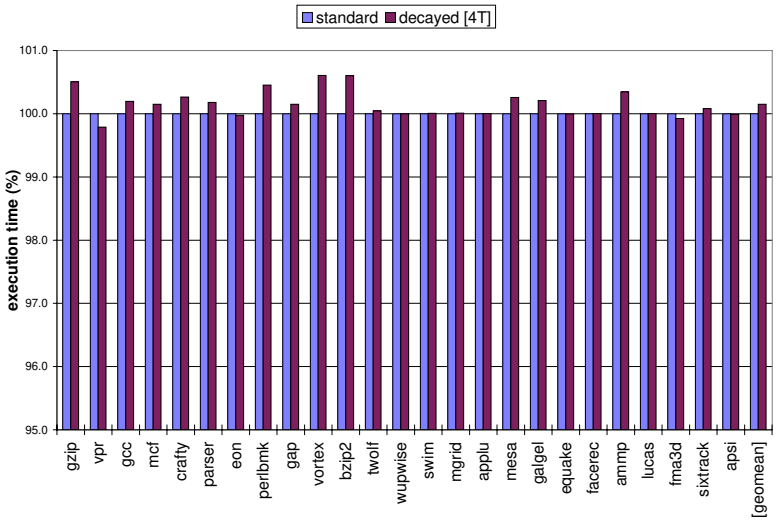

Figure 5.8: Normalized execution time of standard and 4T predictors. 4T predictors produce minimal performance losses.

Figure 5.8 shows the normalized execution time (in percentages) comparing conventional non-decaying 6T-based branch predictors and 4T based branch predictors. Note that

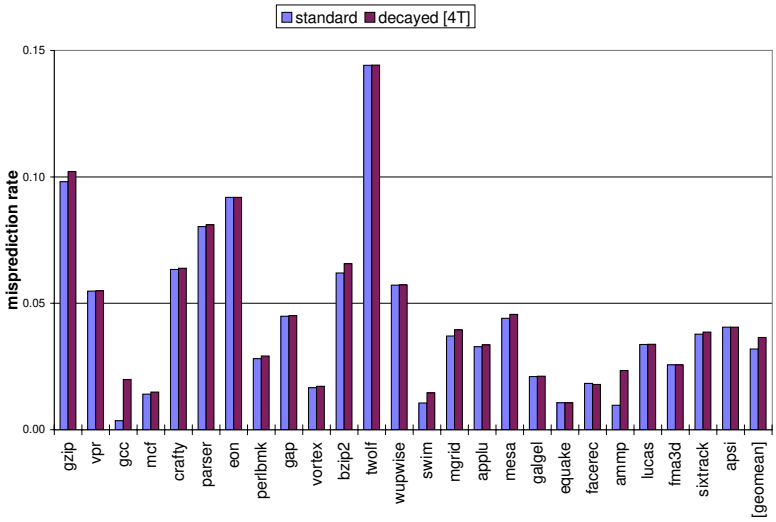

Figure 5.9: Misprediction rate of standard and 4T predictors. 4T predictors produce minimal prediction accuracy degradations.

the y-axis of the graph has a very limited range. From the graph, we see that execution time is virtually unchanged. That is, the performance impact of predicting branches based on decayed predictor entries is negligible. In fact, a few benchmarks actually improve slightly due to the random effects of reading decayed values. Furthermore, prediction accuracy (Figure 5.9) was also virtually unchanged. Over all the benchmarks, the overall prediction accuracy was down less than 0.5%. Figure 5.10 shows the active ratio of the direction counters. On average, we see a 15% active ratio, which directly translates into over 85% savings on leakage power over a traditional, non-decaying predictor.

Finally, the normal dynamic energy overhead of additional mispredictions must be included in our results. Using a calculation similar to that found in cache decay, we can evaluate the impact of additional dynamic overhead caused by decayed (and possibly mispredicted) reads. Note that this number is an energy calculation for the *entire* processor; that is, the dynamic overhead is energy expended by the processor due to a longer runtime.

Figure 5.11 shows the normalized leakage energy with 4T based branch predictor. The standard processor is defined at 1; a number lower than that indicates the processor

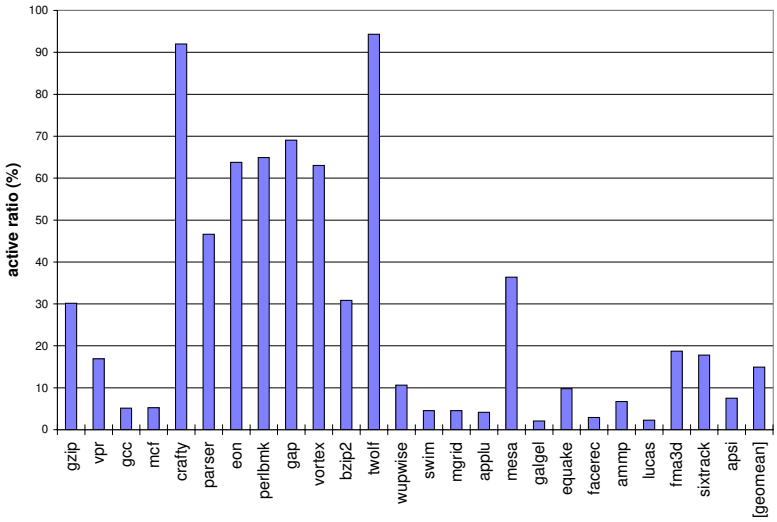

Figure 5.10: Active ratio of a 4T based predictor.

equipped with a particular branch predictor consumed less energy, and vice versa.

As shown in the plot, we see that a processor with a branch predictor using either the standard 4T (Figure 5.11, Left) or slow-decaying 4T (Figure 5.11, Right) cells consumes less energy under the COM3 and COM4 processes. At COM2, the branch predictor is decaying state so rapidly that a lot of useful information is being discarded, imposing a performance penalty so severe that the overall energy consumed by the processor actually increases.

More importantly, we see the impending concern over leakage power more clearly; at COM3 and COM4, where leakage energy has a much larger impact, we can very aggressively decay using standard 4T cells and still achieve an overall power savings.

Overall, 4T cells provide immense power savings with a minimal performance impact. We also see that the processor will consume less energy despite this performance impact, and that as leakage energy increases in influence versus dynamic energy, a 4T based branch predictor becomes much more effective.

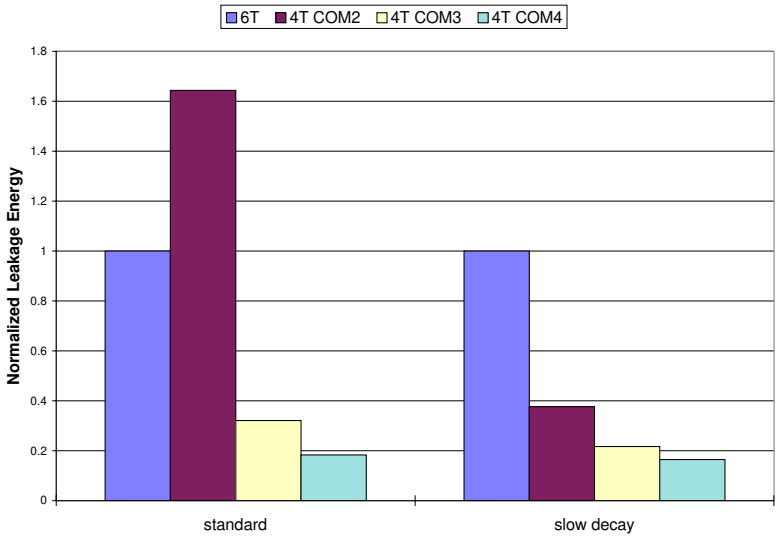

Figure 5.11: Normalized leakage energy for branch predictors with standard (left) and slow-decay (right) 4T cells.

5.8 Chapter Summary

In this chapter we presented an application of the timekeeping methodology to branch predictors. We first tried to extend the cache decay mechanism, described in Chapter 3, directly to branch predictors. However, due to structural differences between caches and branch predictors, implementation of decay strategies in branch predictors requires special consideration:

- First, because a single branch predictor entry is very small (typically only 2-bits wide), adding counters per predictor entry is not hardware-efficient. Instead, counters are added to each row in the predictor array, with each row containing 32-256 entries. We show that branch predictors exhibit strong spatial and temporal locality so that only a subset of rows are active at any point of time so that other rows can be turned off to cut off leakage consumption.

- Second, there are many design choices when applying decay to hybrid predictors, where multiple competing components co-exist. We found that a decay implementa-

tion that takes advantage of the multi-table structure of hybrid predictors can achieve leakage power reduction about 50% more than a naive policy.

- Third, because contents in branch predictors are "transient" and "predictive", a more area-efficient and energy-efficient implementation of decay, using 4-transistor RAM cells, can be used for building branch predictors. 4T cells decay naturally, keep contents long enough for maintaining performance, and have a cell area savings up to 33%.

Overall, this chapter demonstrated that the timekeeping methodology can not only apply to the memory system, but also to other structures such as branch predictors. When applying the timekeeping methodology to a particular structure, the specific characteristics of that structure should be carefully investigated and exploited to achieve the most effective implementation of the timekeeping methodology.

Chapter 6

Conclusions

The advance of semiconductor technology and computer architecture have enabled phenomenal development in microprocessors. In the pursuit of higher performance microprocessors, several key obstacles exist, including ever-increasing power consumption and the memory wall [55]. Past work in attacking these problems have mostly followed a "order-based" methodology, which exploits time-independent characteristics, such as event ordering and event interleaving.

6.1 Contributions

This thesis proposed and evaluated a new methodology, called the "timekeeping methodology", for improving processor power and performance.

First, the timekeeping methodology encourages a fundamentally new way of thinking about how time-dependent aspects of processor behavior can be exploited. We show quantitatively the extent to which detailed timing characteristics of past processor events are strongly predictive of future program behavior. We take the following three steps to illustrate the predictive power of timing characteristics of processor behavior.

- **Metrics**: First, we construct a set of useful metrics which characterize the time-dependent aspects of processor behavior, and we provide quantitative characterizations of the SPEC2000 benchmarks for these metrics.

- **Predictions**: Second, using these metrics, we introduce a fundamentally different approach for on-the-fly categorization of application reference patterns. We give reliable predictors of conflict misses, dead blocks and other key aspects of reference behavior, based on the statistical behavior of the proposed metrics.

- **Mechanisms**: Third, based on our ability to discover these reference patterns on-the-fly, we propose hardware structures that exploit this knowledge to improve performance.

We use the memory system as the main example to illustrate the predictive power of the timekeeping methodology. We propose three novel hardware mechanisms for improving memory performance and power, as shown in Figure 6.1.

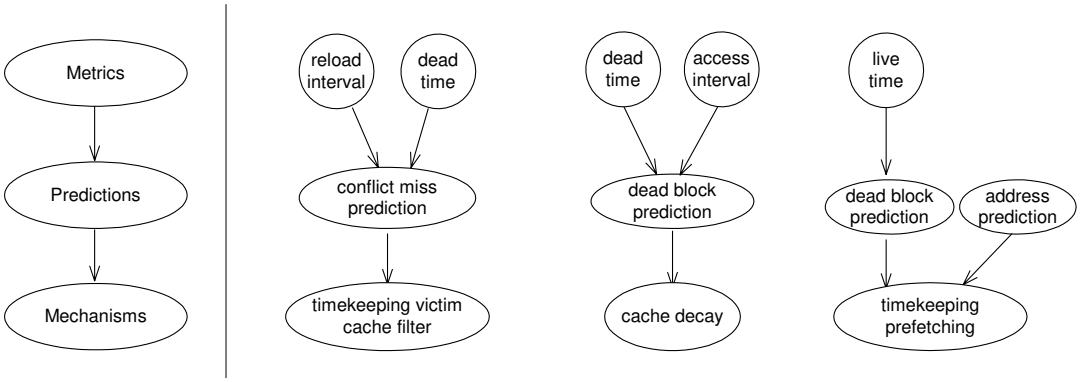

Figure 6.1: Work flow of the timekeeping techniques in the memory system.

- In the ”timekeeping victim cache filter” mechanism, we start by investigating the correlation between miss types and cache line dead times. We find that short dead times are good indicators of conflict misses. We exploit this correlation to optimize

a conflict-oriented structure, the victim cache. More specifically, we filter the victim cache traffic so that only victims with short dead times are allowed to enter the victim cache. This reduce the victim cache traffic by 87% while providing superior performance.

- In the "cache decay" mechanism, cache lines are turned off to save leakage energy if they stay idle for a long time. This is based on the observation that most access intervals are very short, while many dead times are long, indicating that if a cache line stays idle for a long time, it is probably at a dead time (*i.e.*, it is dead). Dead cache lines can be "turned off" to cut off leakage consumption, without impacting performance. Using simple 2-bit counters to dynamically gauge idle time, cache decay can reduce cache leakage energy by 4X, with minimal impact on performance.

- In the "timekeeping prefetch" mechanism, we construct an accurate prefetcher by keeping track of past history of cache line generations. We use live time and next line information observed in the previous generation as predictions for a cache line's current generation. After the predicted live time has elapsed, the predicted next line is prefetched into the cache. With an 8KB integrated history table for both live time and next line information, a timekeeping prefetcher provides an 11% average performance improvement for the whole SPEC2000 benchmark suite, outperforming an recent proposal with a much larger 2MB history table.

Finally, we use branch predictors as an example to illustrate how the timekeeping methodology can be applied outside the memory system. More specifically, we demonstrate how the decay strategies, which have been shown to be effective for caches, can be applied to various styles of branch predictors. Our experiments confirm that decay can be applied to branch predictors, but to achieve a successful and efficient implementation, structural and data characteristics of branch predictors should be carefully identified and

exploited.

- First, since single branch predictor entry is very small, it is not hardware-efficient to add counters at the per-entry granularity. Instead, we propose to apply decay at the granularity of one row in the predictor array. Because of the strong spatial and temporal locality in branch predictors, often there are many rows remain untouched during any time period, so that decay at row-granularity can achieve 40-60% leakage savings.

- Second, in some predictors, such as hybrid predictors, there are multiple competing components. We show that when taking advantage of this structural characteristic, about 50% more leakage savings can be achieved.

- Third, a key characteristic of branch predictor contents is that they are "transient" (they are typically used soon after creation) and "predictive" (losing them does not affect the correctness of the processor). To exploit this characteristic, we propose to build branch predictors with special energy-efficient circuits such as 4-transistor memory cells. A 4T-based branch predictor can save up to 33% in cell area while reducing the leakage energy by 60-80%. More broadly, the effectiveness of 4T-based branch predictors suggests how transient data should be supported in power-aware processors.

6.2 Future Directions

While the proposed timekeeping mechanisms are interesting and highly-effective by themselves, as applications of the timekeeping methodology they demonstrate the power of this new methodology. We expect in our future work, as well as work by other researchers,

more applications of this methodology will be proposed to meet the power and performance challenges of future microprocessors.

- Timekeeping for cache replacement: If perfect knowledge about future is known, an optimal replacement policy (OPT) can be devised: OPT discards the line of a set whose next reference is furthest in the future. Though perfect knowledge about future is not available in practice, the timekeeping methodology can provide predictions of future memory behavior through observation of past behavior. In future work, we intend to exploit the predictive power of the timekeeping methodology for improving cache replacement policies.

- Timekeeping for bus scheduling: Transaction scheduling on buses greatly affects the processor performance and bus power consumption. Current processors often schedule transactions based on when they are issued, but not when they are required to be finished (*i.e.*, their degree of urgency). The main reason for this is because the time when transactions need to be finished is hard to discern with typical trace-based analysis. With the timekeeping methodology, such information can be obtained by keeping track of past time behavior and construct predictions based on them. Overall, timekeeping for bus scheduling is a promising topic to be explored.

- Timekeeping in the processor core: In this thesis we mainly focused on applying the timekeeping methodology to data in processors. Another major type of information in processors contains instructions. In components such as the instruction window, instructions exhibit generational lifetime behavior similar to data in caches. Investigating key timing characteristics of instructions' timing behavior, and exploiting them to improve processor performance and power consumption, is a promising direction in further applying the timekeeping methodology.

- Timekeeping in storage system: Applying the timekeeping methodology to other

levels of computer systems is another promising direction. In this thesis we quantified lifetime characteristics of cache lines and showed how to exploit them for better performance or lower power consumption. In the future we intend to expand our investigation to lifetime characteristics of other objects, such as allocated memory blocks, opened files, web connections, etc. Characteristics found in such investigations can be exploited to improve overall system performance, similar to what we have done to the memory system.

- Timekeeping in real-time computing: A key advantage of the timekeeping methodology is that it not only predicts what will happen in the future, but also estimates when it will happen. This is possible because the timekeeping methodology investigates regularities between past and future timing behavior, and deduces future time intervals based on what have occurred in the past. Knowing when future events happen is crucial for real-time computing, which usually benefits from predictability but suffers from randomness.

- Combining the timekeeping methodology with theoretical underpinnings: The timekeeping methodology is an empirical study, in the sense that first typical program behavior is investigated and then common patterns discovered are exploited. A promising avenue for future work is to integrate this empirical study with theories such as "Little's Law". Such theories could provide theoretical guidance for applying the timekeeping methodology.

6.3 Chapter Summary

Challenges abound in the research of computer architecture. In prior work, to attack these challenges most architects limited their investigations to the time-independent as-

pects of processor behavior (such as event ordering and interleaving), while letting the time-dependent aspects (such as time intervals between events) play a lesser role. In contrast, this thesis has proposed focusing on the time-dependent aspects of processor behavior. We show quantitatively that the timing characteristics of program lifetime behavior can be strongly predictive of future processor events, and thus can provide powerful ways of understanding and improving program behavior. We illustrate the power of this "timekeeping" methodology with a group of concrete hardware mechanisms, each keeping track of key time intervals at run time, using them to deduce future processor behavior on-the-fly, and exploiting knowledge of future events for improving processor power and performance. In our future work, as well as work by others, we expect that the timekeeping methodology, described in this thesis, will offer researchers even more intuition for further understanding and optimizing processor behavior.

Bibliography

[1] D. H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999.

[2] J. Baer and W. Wang. On the Inclusion Property in Multi-level Cache Hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988.

[3] J.-L. Baer and T.-F. Chen. Dynamic Improvements of Locality in Virtual Memory Systems. *IEEE Transactions on Software Engineering*, 1976.

[4] J.-L. Baer and T.-F. Chen. An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty. In *Proceedings of the 5th International Conference on Supercomputing*, pages 176–186, Nov. 1991.

[5] S. Borkar. Design Challenges of Technology Scaling. *IEEE Micro*, 19(4), 1999.

[6] W. J. Bowhill, S. L. Bell, B. J. Benschneider, A. J. Black, S. M. Britton, R. W. Castelino, D. R. Donchin, J. H. Edmondson, H. R. F. III, P. E. Gronowski, A. K. Jain, P. L. Kroesen, M. E. Lamere, B. J. Loughlin, S. Mehta, R. O. Mueller, R. P. Preston, S. Santhanam, T. A. Shedd, M. J. Smith, and S. C. Thierauf. Circuit Im-

plementation of a 300-MHz 64-bit Second-Generation CMOS Alpha CPU. *Digital Technical Journal*, 7(1):100–118, 1995.

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

[8] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: the SimpleScalar Tool Set. Tech. Report TR-1308, Univ. of Wisconsin-Madison Computer Sciences Dept., July 1996.

[9] D. Burger, J. Goodman, and A. Kagi. The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors. Tech. Report TR-1216, Univ. of Wisconsin-Madison Computer Sciences Dept.

[10] P.-Y. Chang, E. Hao, and Y. N. Patt. Alternative Implementations of Hybrid Branch Predictors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 252–57, Dec. 1995.

[11] M. J. Charney and A. P. Reeves. Generalized Correlation-Based Hardware Prefetching. Technical Report EE-CEG-95-1, School of Electrical Engineering, Cornell University, 1995.

[12] Z. Chen, L. Wei, M. Johnson, and K. Roy. Estimation of Standby Leakage Power in CMOS Circuits Considering Accurate Modeling of Transistor Stacks. In *Proceedings of the 1998 International Symposium on Lower Power Electronics and Design*, 1998.

[13] J. Collins and D. Tullsen. Hardware Identification of Cache Conflict Misses. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 126–135, 1999.

[14] J. Dean, J. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997.

[15] K. Diefendorff. Pentium III = Pentium II + SSE. *Microprocessor Report*, Mar. 8 1999.

[16] Digital Semiconductor. *DECchip 21064/21064A Alpha AXP Microprocessors: Hardware Reference Manual*, Jun. 1994.

[17] Digital Semiconductor. *Alpha 21164 Microprocessor: Hardware Reference Manual*, Apr. 1995.

[18] B. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[19] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy Caches: Techniques for Reducing Leakage Power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[20] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: An Analytical Representation of Cache Misses. In *International Conference on Supercomputing*, pages 317–324, 1997.

[21] D. Gross and C. M. Harris. *Fundamentals of Queueing Theory*. John Wiley and Sons, 1997.

[22] L. Gwennap. Digital 21264 Sets New Standard. *Microprocessor Report*, pages 11–16, Oct. 28, 1996.

[23] H.-H. Lee, G. S. Tyson, M. Farrens. Eager Writeback - a Technique for Improving Bandwidth Utilization. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2000.

[24] H. Hanson, M. Hrishikesh, V. Agarwal, S. W. Keckler, and D. Burger. Static Energy Reduction Techniques for Microprocessor Caches. In *Proceedings of the 2001 International Conference on Computer Design*, 2001.

[25] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, Inc., San Mateo, California, 1995. Second Edition.

[26] S. Heo, K. Barr, M. Hampton, and K. Asanovic. Dynamic Fine-Grain Leakage Reduction using Leakage-Biased Bitlines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[27] M. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California at Berkeley, Nov. 1987.

[28] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, 2001. 1st quarter.

[29] M. A. Holliday. Techniques for Cache and Memory Simulation Using Address Reference Traces. *International Journal in Computer Simulation*, 1(2), 1991.

[30] Z. Hu, P. Juang, P. Diodato, S. Kaxiras, K. Skadron, M. Martonosi, and D. W. Clark. Managing Leakage for Transient Data: Decay and Quasi-Static 4T Memory Cells. In *Proceedings of the 2002 International Symposium on Lower Power Electronics and Design*, 2002.

[31] Z. Hu, P. Juang, K. Skadron, D. Clark, and M. Martonosi. Applying Decay Strategies to Branch Predictors for Leakage Energy Savings. Tech. Report CS-2001-24, Univ. of Virginia.

[32] Z. Hu, P. Juang, K. Skadron, D. Clark, and M. Martonosi. Applying Decay Strategies to Branch Predictors for Leakage Energy Savings. In *International Conference on Computer Design*, 2002.

[33] Z. Hu, S. Kaxiras, and M. Martonosi. Let Caches Decay: Reducing Leakage Energy via Exploitation of Cache Generational Behavior. *ACM Transactions on Computer Systems*, May 2002.

[34] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[35] J. A. Butts and G. Sohi. A Static Power Model for Architects. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2000.

[36] D. A. Jiménez, S. W. Keckler, and C. Lin. The Impact of Delay on the Design of Branch Predictors. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 67–77, Dec. 2000.

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

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

[39] M. B. Kamble and K. Ghose. Analytical Energy Dissipation Models for Low Power Caches. In *Proceedings of the 1997 International Symposium on Lower Power Electronics and Design*, 1997.

[40] G. B. Kandiraju and A. Sivasubramaniam. Going the Distance for TLB Prefetching: An Application-driven Study. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[41] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In *Proceedings of the 12thSymposium on Operating System Principles*, 1991.

[42] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[43] S. Kaxiras, Z. Hu, G. Narlikar, and R. McLellan. Cache-Line Decay: A Mechanism to Reduce Cache Leakage Power. In *Workshop on Power-Aware Computer Systems (PACS), In conjunction with the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[44] T. Kimbrel and A. Karlin. Near-Optimal Parallel Prefetching and Caching. *SIAM Journal on computing*, 2000.

[45] A.-C. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction and Dead-Block Correlating Prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[46] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, Dec. 1997.

[47] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. Spaid: Software Prefetching in Pointer and Call-Intensive Environments. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 231–236, 1995.

[48] C.-K. Luk and T. C. Mowry. Compiler Based Prefetching for Recursive Data Structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Oct. 1996.

[49] N. R. Mahapatra and B. Venkatrao. The Processor-Memory Bottleneck: Problems and Solutions. *ACM Crossroads*, 1999.

[50] T. Mathisen. Pentium Secrets. *Byte*, pages 191–192, July 1994.

[51] S. McFarling. Combining Branch Predictors. Tech. Note TN-36, Compaq WRL, June 1993.

[52] A. Mendelson, D. Thiébaut, and D. K. Pradhan. Modeling Live and Dead Lines in Cache Memory Systems. *IEEE Transactions on Computers*, 42(1):1–16, Jan. 1993.

[53] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, pages 114–17, Apr. 1965.

[54] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct. 1992.

[55] T. Mudge. Strategic Directions in Computer Architecture. *ACM Computing Surveys*, 28(4):671–678, 1996.

[56] 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 the 1998 International Symposium on Lower Power Electronics and Design*, 1998.

[57] T. Ozawa, Y. Kimura, and S. Nishizaki. Cache Miss Heuristics and Preloading Techniques for General-Purpose Programs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Dec. 1995.

[58] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan. Power Issues Related to Branch Prediction. pages 233–44, Feb. 2002.

[59] M. D. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *Proceedings of the 2000 International Symposium on Lower Power Electronics and Design*, 2000.

[60] T. R. Puzak. *Analysis of Cache Replacement Algorithms*. PhD thesis, University of Massachusetts, Amherst, 1985.

[61] T. Romer, W. Ohlrich, A. Karlin, and B. Bershad. Reducing TLB and Memory Overhead Using Online Superpage Promotion. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[62] A. Roth, A. Moshovos, and G. S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[63] S. Sair and M. Charney. Memory Behavior of the SPEC2000 Benchmark Suite. Technical report, IBM RC-21852, 2000.

[64] A. Saulsbury, F. Dahlgren, and P. Stenstrom. Recency-Based TLB Preloading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[65] S. Schuster, L. Terman, and R. Franch. A 4-Device CMOS Static RAM Cell Using Sub-Threshold Conduction. In *Symposium on VLSI Technology, Systems, and Applications*, 1987.

[66] Semiconductor Industry Association. The International Technology Roadmap for Semiconductors, 2001. http://www.semichips.org.

[67] K. Skadron, T. Abdelzaher, and M. R. Stan. Control-Theoretic Techniques and Thermal-RC Modeling for Accurate and Localized Dynamic Thermal Management. pages 17–28, Feb. 2002.

[68] A. J. Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, Sept. 1982.

[69] J. E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135–48, May 1981.

[70] Y. Solihin, J. Lee, and J. Torrellas. Using a User-Level Memory Thread for Correlation Prefetching. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[71] P. Song. UltraSparc-3 Aims at MP Servers. *Microprocessor Report*, pages 29–34, Oct. 27 1997.

[72] S. Srinivasan, R. Ju, A. Lebeck, and C. Wilkerson. Locality vs. Criticality. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[73] The Standard Performance Evaluation Corporation. WWW Site. http://www.spec.org, Dec. 2000.

[74] R. A. Uhlig and T. N. Mudge. Trace-Driven Memory Simulation: A Survey. *ACM Computing Surveys*, 29(2):128–170, 1997.

[75] A. G. Varadi. Quasi-Static MOS Memory Array with Standby Operation. US Patent Number 4,120,047.

[76] W. Wolf. *Modern VLSI Design: Systems on Silicon*. 1998. Prentice-Hall.

[77] D. A. Wood, M. D. Hill, and R. E. Kessler. A Model for Estimating Trace-Sample Miss Ratios. In *Proceedings of the 1991 Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 79–89, June 1991.

[78] C. E. Wu, Y. H. Liu, C. Benveniste, C. L. Chen, and W. Chiang. Trace-Based Analysis and Tuning for Distributed Parallel Applications. In *Proceedings of the International Conference on Distributed Computing Systems*, 1994.

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

[80] S.-H. Yang, M. D. 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 the Eighth International Symposium on High-Performance Computer Architecture*, Jan. 2001.

[81] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proceedings of the 10th International Conference on Supercomputing*, 1996.

[82] H. Zhou, M. Toburen, E. Rotenberg, and T. Conte. Adaptive Mode Control: A Static-Power-Efficient Cache Design. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.