Copyright

by

Haiming Liu

2009

The Dissertation Committee for Haiming Liu

certifies that this is the approved version of the following dissertation:

Hardware Techniques to Improve Cache Efficiency

Committee:

Douglas C. Burger, Supervisor

Derek Chiou

Stephen W. Keckler

Calvin Lin

Kathryn S. McKinley

Hardware Techniques to Improve Cache Efficiency

by

Haiming Liu, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2009

To my wife and my parents.

Acknowledgments

First, I would like to thank my advisor, Doug Burger, for his advice, encouragements, and support during my graduate study. Doug has been a perfect role model of a great researcher. His passion and dedication to research and his way of creative thinking will always be my source of inspiration in my future career. I am forever grateful to Doug for showing me the skills to conduct research and his commitment to support over the years.

I would also like to thank Steve Keckler, co-leader of the CART research group and the TRIPS project, for sharing his insights and advice on numerous occasions. It is amazing to see the broadness of Steve's knowledge and his ability to catch important details during the TRIPS project. I would like to thank the other members of my dissertation committee, Kathryn McKinley, Derek Chiou, and Calvin Lin, for their helpful feedbacks on this dissertation work. Special thanks to Kathryn McKinley for taking the time to go over my dissertation so carefully and providing great feedbacks.

I am very thankful to my fellow students of the CART research group, who have been such a wonderful group of people to work with. I would like to thank Mark Gebhart, Sibi Govindan, Paul Gratz, Boris Grot, Heather Hanson, Changkyu Kim, Ramadass Nagarajan, Nitya Ranganathan, Behnam Robatmili, Karu Sankaralingam, Simha Sethumadhavan, Sadia Sharif, Premkishore Shivakumar, and Renee St. Amant for their discussions and help through the years. Special thanks to Paul

Gratz for always being available to provide feedbacks on paper drafts and practice talks and for being patient when there were bugs in the MT testbench during the validation of the Memory Tile in the TRIPS prototype. Special thanks to Sibi Govindan for always being willing to help and for sharing his hilarious comments from time to time. I would also like to thank Mario Marino for his feedbacks while he was working at the CART group. Thanks also go to Gem Naivar for always being patient and taking care of various miscellany.

I would like to thank Mike Ferdman for various discussions about the cache burst work and for willing to present the paper at Micro with an extremely tight schedule when I was not able to attend. I would also like to thank Jaehyuk Huh for his help with the MPsauce simulator and with the paper submission. In particular, I would like to thank Babak Falsafi, for sharing his insights on the cache burst work and for his extremely helpful suggestions about the presentation of the cache burst paper at Micro.

I would acknowledge that this work was supported by the Defense Advanced Research Projects Agency under contract F33615-03-C-4106 and by NSF grant EIA-0303609.

Finally, I would like to thank my family. My parents have always been supportive throughout the years. Without their support, I would not have come to this point. I wanted to thank my wife, Yan Li, who has shared all the difficulties through the past several years and has been constantly given me support and encouragements. I owe her so much for all her love and support and look forward to our life together in the many years to come.

Haiming Liu

The University of Texas at Austin

May 2009

Hardware Techniques to Improve Cache Efficiency

Publication No. _____

Haiming Liu, Ph.D.
The University of Texas at Austin, 2009

Supervisor: Douglas C. Burger

Modern microprocessors devote a large portion of their chip area to caches in order to bridge the speed and bandwidth gap between the core and main memory. One known problem with caches is that they are usually used with low efficiency; only a small fraction of the cache stores data that will be used before getting evicted. As the focus of microprocessor design shifts towards achieving higher performance-per-watt, cache efficiency is becoming increasingly important. This dissertation proposes techniques to improve both data cache efficiency in general and instruction cache efficiency for Explicit Data Graph Execution (EDGE) architectures.

To improve the efficiency of data caches and L2 caches, dead blocks (blocks that will not be referenced again before their eviction from the cache) should be identified and evicted early. Prior schemes predict the death of a block immediately

after it is accessed, based on the individual reference history of the block. Such schemes result in lower prediction accuracy and coverage. We delay the prediction to achieve better prediction accuracy and coverage. For the L1 cache, we propose a new class of dead-block prediction schemes that predict dead blocks based on cache bursts. A cache burst begins when a block moves into the MRU position and ends when it moves out of the MRU position. Cache burst history is more predictable than individual reference history and results in better dead-block prediction accuracy and coverage. Experiment results show that predicting the death of a block at the end of a burst gives the best tradeoff between timeliness and prediction accuracy/coverage. We also propose mechanisms to improve counting-based dead-block predictors, which work best at the L2 cache. These mechanisms handle reference-count variations, which cause problems for existing counting-based dead-block predictors. The new schemes can identify the majority of the dead blocks with approximately 90% or higher accuracy. For a 64KB, two-way L1 D-cache, 96% of the dead blocks can be identified with a 96% accuracy, half way into a block's dead time. For a 64KB, four-way L1 cache, the prediction accuracy and coverage are 92% and 91% respectively. At any moment, the average fraction of the dead blocks that has been correctly detected for a two-way or four-way L1 cache is approximately 49% or 67% respectively. For a 1MB, 16-way set-associative L2 cache, 66% of the dead blocks can be identified with a 89% accuracy, $1/16^{th}$ way into a block's dead time. At any moment, 63% of the dead blocks in such an L2 cache, on average, has been correctly identified by the dead-block predictor. The ability to accurately identify the majority of the dead blocks in the cache long before their eviction can lead to not only higher cache efficiency, but also reduced power consumption or higher reliability.

In this dissertation, we use the dead-block information to improve cache efficiency and performance by three techniques: replacement optimization, cache

bypassing, and prefetching into dead blocks. Replacement optimization evicts blocks that become dead after several reuses, before they reach the LRU position. Cache bypassing identifies blocks that cause cache misses but will not be reused if they are written into the cache and do not store these blocks in the cache. Prefetching into dead blocks replaces dead blocks with prefetched blocks that are likely to be referenced in the future. Simulation results show that replacement optimization or bypassing improves performance by 5% and prefetching into dead blocks improves performance by 12% over the baseline prefetching scheme for the L1 cache and by 13% over the baseline prefetching scheme for the L2 cache. Each of these three techniques can turn part of the identified dead blocks into live blocks. As new techniques that can better utilize the space of the dead blocks are found, the dead-block information is likely to become more valuable.

Compared to RISC architectures, the instruction cache in EDGE architectures faces challenges such as higher miss rate, because of the increase in code size, and longer miss penalty, because of the large block size and the distributed microarchitecture. To improve the instruction cache efficiency in EDGE architectures, we decouple the next-block prediction from the instruction fetch so that the next-block prediction can run ahead of instruction fetch and the predicted blocks can be prefetched into the instruction cache before they cause any I-cache misses. In particular, we discuss how to decouple the next-block prediction from the instruction fetch and how to control the run-ahead distance of the next-block predictor in a fully distributed microarchitecture. The performance benefit of such a look-ahead instruction prefetching scheme is then evaluated and the run-ahead distance that gives the best performance improvement is identified. In addition to prefetching, we also estimate the performance benefit of storing variable-sized blocks in the instruction cache. Such schemes reduce the inefficiency caused by storing NOPs in the I-cache and enable the I-cache to store more blocks with the same capacity. Simula-

tion results show that look-ahead instruction prefetching and storing variable-sized blocks can improve the performance of the benchmarks that have high I-cache miss rates by 17% and 18% respectively, out of an ideal 30% performance improvement only achievable by a perfect I-cache. Such techniques will close the gap in I-cache hit rates between EDGE architectures and RISC architectures, although the latter will still have higher I-cache hit rates because of the smaller code size.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Technology advances [83] in the past several decades, along with tradeoffs in device performance, capacity, and cost, have resulted in a huge speed gap between microprocessor core and main memory [62, 103]. Nowadays, the cycle time of modern microprocessor cores ranges from 0.2ns to 1ns while DRAM latency is around 50ns. As a result, main memory accesses take hundreds of processor cycles. To bridge the speed gap between microprocessor core and main memory, caches [99], in the form of smaller and faster on-chip SRAM memories, are widely used.

Technology advances and software demands have also caused total cache capacity to grow, by adding more levels in the cache hierarchy and increasing the capacity of the caches. On one hand, technology scaling results in more and faster transistors in the microprocessor core. At the same time, advances in architecture and microarchitecture research result in improved microprocessor designs. Together, these two factors cause the microprocessor core to become more powerful and capable of processing more data in the same amount of time. On the other hand, software programs are getting larger and more complex and program footprints (in

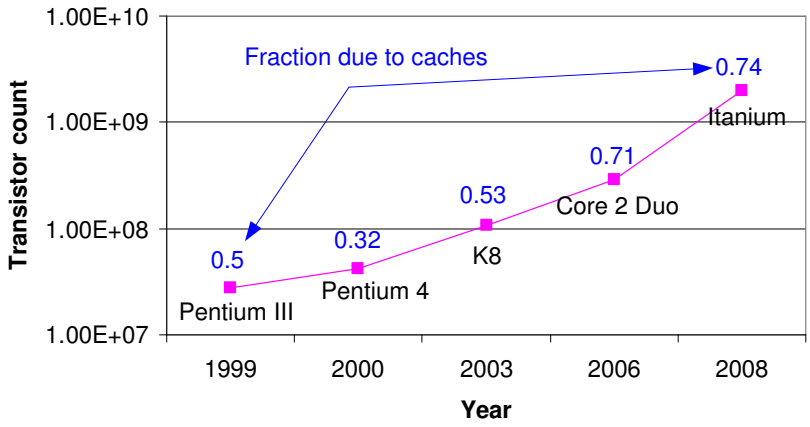


Figure 1.1: Transistors spent on caches

terms of both data and instructions) have increased significantly, requiring larger caches to hold the working set. To feed the microprocessor core with more data and still provide fast access to the most commonly used data, multiple levels of caches are used such that the L1 cache provides fast access and lower levels of caches provide larger capacity. This trend is shown in Figure 1.1, which shows both the total number of transistors and the fraction of the transistors due to caches in some of the major microprocessors from Intel and AMD in the past several years. The Y axis in Figure 1.1 is the total transistor count of each chip. The numbers above each data point are the fraction of the transistors due to caches in each chip. Figure 1.1 shows that not only the total number of transistors of each chip has been increasing over the years, but also the fraction of the transistors due to caches has been increasing, too. The end result is that as much as three quarters of the transistors on a microprocessor chip area are devoted to caches nowadays [1] and this percentage may increase in the future.

Although a large fraction of the transistors have been devoted to caches,

caches are not utilized very well. Previous studies [10, 63] have found that only a small fraction of the cache capacity, typically less than 20%, stores useful data while the majority of the cache capacity stores data that are just waiting to be evicted from the cache. This dissertation studies hardware techniques to utilize caches more efficiently by reducing the fraction of the useless information stored in the cache.

1.1 The Cache Efficiency Problem

The concept of cache efficiency was first proposed by Burger et al. in [10], where it is defined as the average fraction of the cache capacity that stores useful data. Cache efficiency can be measured at the cache block level or at the byte level. Block-level efficiency treats a cache block as the fundamental element and does not differentiate references to different bytes within a block. Byte-level efficiency treats each individual byte in a block as the fundamental element. While byte-level efficiency gives more detailed information about the utilization of the cache, from the hardware point of view, a cache block is the fundamental element that the cache hardware manages so block-level efficiency is a more natural metric unless the hardware is modified to manage the cache in smaller units. This dissertation focuses on block-level efficiency only.

Block-level cache efficiency is measured by the fraction of live cache blocks out of all the blocks in a cache [10]. A block is *live* if it will be referenced again before its eviction from the cache. A cache block is *dead* if it will not be referenced, either by a load or a store instruction, before its eviction from the cache. For any cycle during the execution of a program, the fraction of the live blocks in that cycle can be computed. The arithmetic mean of these live-block fractions across the execution time of a program, or cache efficiency, can be computed as:

$$E = \frac{\sum_{i=0}^{A \times S - 1} U_i}{N \times A \times S} \tag{1.1}$$

In Equation 1.1, A is the associativity of the cache, S is the number of sets, N is the execution time in cycles, and U_i is the total number of cycles that cache block i is live.

When running the SPEC 2000 benchmarks, the average efficiency of a two-way, 64KB L1 data cache is only 8% and the efficiency of a 16-way, 1MB L2 cache is only 17%, indicating the poor utilization of the caches and significant opportunities for improvement. The efficiency of a 80KB, four-way instruction cache in an EDGE architecture processor is higher at 29%, but still has plenty of room for improvement.

The root cause of low cache efficiencies is that blocks die, reside in the cache for a long period of time with no accesses, and then are finally evicted. With LRU or pseudo-LRU replacement, upon the last access to a block, multiple replacements to that set must occur before the dead block is evicted [29, 64, 102], which can take thousands of cycles. The interval between the last access to a block and its eviction from the cache is called the dead time of the block. Likewise, the interval between the first access to a block in the cache, i.e., the access which brings the block into the cache, and the last access to the block while it is in the cache, is called the live time. Prior work has shown that the dead time is usually at least one order of magnitude longer than the live time [29]. As a result, the average cache efficiency defined in Equation 1.1 can be very low.

1.2 Techniques to Increase Cache Efficiency

To increase cache efficiency, the hardware should store more live blocks in the cache without increasing the capacity of the cache. Because of the differences in how data caches and instruction caches are organized and how they interact with other components of a microprocessor, the techniques for improving the efficiency of these caches are different.

1.2.1 Techniques to Increase Data Cache Efficiency

The first part of the dissertation studies techniques to increase data cache efficiency. To increase data cache efficiency, the hardware should identify the dead blocks early and replace them with live blocks.

- Identifying dead blocks early: the earlier the hardware knows that a block has become dead, the more opportunity there is to improve cache efficiency.
- Eliminating dead blocks from the cache early: once a dead block is identified, either it should be evicted from the cache so that other blocks get more opportunities to get reused or a new block that is likely to be referenced in the future should be brought into the cache to replace the dead block.

It is also possible to increase the data cache efficiency by prefetching without a mechanism to identify dead blocks. Such schemes either choose to be conservative in initiating prefetches for pollution considerations, or can cause live blocks to be evicted if the prefetcher is too aggressive. If the prefetcher is too conservative, only a small portion of the dead blocks will be replaced with live blocks. On the other hand, if the prefetcher is too aggressive, the prefetched blocks can evict live blocks in the cache and cancel the improvement in cache efficiency. Ideally, the prefetcher should

only prefetch into the dead blocks in the cache. As we demonstrate later in this dissertation, the capability to identify dead blocks early improves the effectiveness of prefetching.

Identifying Dead Blocks Early

A block turns dead on its last access before its eviction from the cache. The identification of a dead block should be done between the last access to the block and its eviction from the cache. Since the hardware does not know with certainty whether an access to a block is the last access, the identification of a block as dead before its eviction is a speculative action called *dead-block prediction*.

Three approaches for dead-block prediction have been proposed: trace-based, counting-based, and time-based. Trace-based predictors [17, 48] predict a block dead once it has been accessed by a certain sequence of instructions. Counting-based predictors [39, 40] predict a block dead once it been accessed a certain number of times. Time-based predictors [4, 29, 37] predict a block dead once it has not been accessed for a certain number of cycles.

Most prior dead-block prediction schemes predict whether a block has died immediately after the block is accessed. While this approach identifies dead blocks as early as possible, it sacrifices prediction accuracy and coverage because a block just accessed may be accessed again soon. There is a tradeoff between the timeliness and accuracy/coverage of dead-block prediction. The earlier the prediction is made, the more useful it is. At the same time, the later the prediction is made, the more accurate it is. This dissertation quantifies this tradeoff by making dead-block predictions at different points during the dead time of a block. The results show that making dead-block predictions when a block just becomes non-MRU gives the

best tradeoff between timeliness and prediction accuracy/coverage.

Prior dead-block prediction schemes also update the history of a block *every time* the block is referenced. A prediction about whether a block has died is made based on the individual reference history of each block. However, how a block is accessed in the L1 data cache may depend on which control flow path the program takes, the value or offset of the referenced data in the block, and other parameters. These variations can cause the individual reference history of a block to be irregular and cause problems for existing dead-block prediction schemes. To address this problem, we propose a new class of dead-block prediction schemes for the L1 cache that predict dead blocks based on the *cache burst* history of each block. A cache burst begins when a block moves into the MRU position and ends when it moves out of the MRU position. In these new dead-block prediction schemes, the contiguous references a block receives in the MRU position are grouped into one cache burst. A prediction about whether a block has died is made only when it moves out the MRU position, using the block's cache burst history. Since cache burst history hides the irregularity of individual references within a cache burst, it is easier to predict than individual reference history for L1 caches.

Cache bursts only work well at the L1 cache. For the L2 cache, counting-based predictors work best. However, existing counting-based predictors can suffer problems caused by reference-count variations. We propose two mechanisms to tolerate reference-count variations by filtering out sporadic smaller reference counts and by using more up-to-date reference-count history information.

The new schemes proposed in this dissertation can identify the majority of the dead blocks with approximately 90% or higher accuracy. For a 64KB, two-way L1 D-cache, 96% of the dead blocks can be identified with a 96% accuracy, half way

into a block's dead time. For a 64KB, four-way L1 cache, the prediction accuracy and coverage are 92% and 91% respectively. At any moment, the average fraction of the dead blocks that has been correctly detected for a two-way or four-way L1 cache is approximately 49% or 67% respectively. For a 1MB, 16-way set-associative L2 cache, 66% of the dead blocks can be identified with a 89% accuracy, $1/16^{th}$ way into a block's dead time. At any moment, 63% of the dead blocks in such an L2 cache, on average, has been correctly identified by the dead-block predictor. The ability to accurately identify the majority of the dead blocks in the cache long before their evition time can lead to not only higher cache efficiency, but also reduced power consumption or higher reliability.

Eliminating Dead Blocks Early

Identifying dead blocks early is only the first step towards improving cache efficiency. The second step is to use the dead-block information to reduce the number of cache misses. Three optimizations are possible: replacement optimizations, cache bypassing, and prefetching into dead blocks.

Replacement optimizations: With LRU replacement, the hardware always chooses the LRU block for replacement on a cache miss. Using the dead-block information, the hardware can choose to replace a dead block not in the LRU position for replacement. This reduces the time a dead block stays in the cache and gives other blocks in the same set more opportunities to get reused.

Cache bypassing: When a cache miss occurs, most replacement algorithms will pick a block already in the cache for replacement; the block causing the miss is always inserted into the cache. However, if the block causing the miss dies immediately after it is inserted into the cache, bypassing it (not inserting it into the cache)

is a better choice. Bypassing can be especially effective for those applications that have a working set larger than the capacity of the cache and cause most blocks to be evicted before they get the chance to be reused.

Prefetching into dead blocks: Replacement optimizations and cache bypassing have the limitation that successful bypassing or early replacement of dead blocks does not always reduce the miss rate of the cache. Prefetching into dead blocks is a more aggressive technique that tries to replace the dead blocks with blocks that may be referenced in the future. Using dead-block prediction to trigger prefetches has two benefits. First, dead blocks provide some ideal location to store prefetched blocks without causing pollution. When applied to different levels of caches, this property can cause different tradeoffs between the aggressiveness of prefetching and the resulting pollution. Second, a long dead time gives sufficient time for prefetched blocks to arrive at the cache before they are referenced.

1.2.2 Techniques to Increase Instruction Cache Efficiency in EDGE Architectures

The second part of this dissertation studies techniques to increase the instruction cache efficiency in EDGE architectures. EDGE architectures are designed to sustain high-performance execution of general-purpose single-threaded programs in future technologies where wire delay will make traditional superscalar designs increasingly harder to be implemented due to the high cost and complexity of their centralized structures. EDGE architectures employ a distributed microarchitecture to address the wire-delay problem.

EDGE architectures feature an execution model called block-atomic execution, which groups many instructions into a *block*. A program for EDGE archi-

tectures consists of many blocks. The instructions within a block always commit atomically: either all of them commit or none of them commit. The block-atomic execution model, along with the need for a distributed microarchitecture to tolerate wire delay and increase concurrency, requires a new instruction cache design.

This dissertation presents the design and implementation of the distributed instruction cache in the TRIPS prototype, a microarchitecture instantiation of EDGE architectures, implemented in silicon. This I-cache design is then extended to TFlex, another microarchitecture instantiation of EDGE architectures that shares the same instruction set architecture as TRIPS but addresses the limitations of TRIPS.

Compared to superscalar architectures, EDGE architectures put more pressure on the instruction cache for two reasons. One reason is that the size of the program code on EDGE architectures is larger. Another reason is that a large portion of the instruction cache can be wasted if the hardware can only store fixed-sized blocks because many instructions within a block can be NOPs. Our experience with the TRIPS prototype indicates that for some applications, the instruction cache is a performance bottleneck because of frequent I-cache misses. This dissertation studies how to reduce the I-cache miss rate without increasing the capacity of the I-cache, i.e., improving the I-cache efficiency.

Improving I-cache Efficiency through Prefetching

While the techniques discussed earlier to improve the data cache efficiency can also be applied to instruction caches, they require extra hardware for dead-block prediction and prefetching address prediction. A more effective approach to improve the I-cache efficiency in EDGE architectures is to take advantage of the next-block

prediction mechanism that already exists in the microarchitecture and use it to guide prefetching blocks into the instruction cache. This approach has low hardware overhead and at the same time can accurately predict the addresses of future blocks that will be needed by the program. To prefetch the future blocks that will cause I-cache misses into the instruction cache in time, the next-block prediction should be decoupled from instruction fetch and run some distance ahead of instruction fetch.

To decouple next-block prediction from instruction fetch, some buffering mechanism is needed to store the block addresses produced by the next-block prediction hardware. This buffering mechanism is called a fetch target buffer [75, 76]. The next-block prediction hardware produces addresses of blocks that will be executed in the future and enqueues these addresses into the fetch target buffer. The instruction fetch hardware consumes these addresses by dequeuing them from the fetch target buffer.

The fetch target buffer is fairly straightforward to implement in a microarchitecture where branch prediction and instruction fetch are centralized because the position of an entry in the fetch target buffer is solely determined by the age of the entry. Implementing a fetch target buffer in a distributed microarchitecure is more challenging because the position of an entry in the fetch target buffer is determined not only by the age of entry, but also by the address of the fetch target. A distributed microarchitecture also makes it more challenging to control how far the next-block prediction runs ahead of the instruction fetch. The maximum distance between the next-block prediction and the instruction fetch affects both the effectiveness and cost of the fetch target buffer. This dissertation studies how to implement the fetch target buffer in a distributed microarchitecture and what run-ahead distance gives the best performance improvement.

Improving I-cache Efficiency through Variable-sized Blocks

Complementary to prefetching, the I-cache efficiency in EDGE architectures can also be improved by storing variable-sized blocks in the instruction cache. The block-atomic execution model of EDGE architectures requires the compiler to group many instructions into a block. If the compiler can not fill a block with useful instructions, NOPs will be used to pad the unused space. The TRIPS prototype always stores a full block in the instruction cache, even if the block has only a few useful instructions. Storing fixed-sized lblocks in the I-cache can waste a lot of space and result in low I-cache efficiency. This dissertation discusses how to store variable-sized blocks in the instruction cache to reduce the space wasted by storing NOPs. It also estimates the potential improvement that can be achieved by storing variable-sized blocks in the instruction cache and by combining it with instruction cache prefetching.

1.3 Contributions

This dissertation addresses the problem of how to utilize caches, the largest component of modern microprocessors, more efficiency. It shows that by using simple hardware mechanisms, the majority of the dead blocks in the cache can be accurately identified shortly after their last access in the cache. The ability to identify dead blocks early creates opportunities for performance improvements, power reduction, and improved reliability. While this dissertation only explores how to use the dead-block information to improve performance, the dead-block information can prove more valuable as new techniques that can utilize the identified dead space more effectively are found.

Although the instruction cache only accounts for a small portion of the total cache capacity, it is highly critical to the overall performance of a microprocessor.

This dissertation shows the design and implementation of an instruction cache for a new architecture called EDGE architectures, and how the performance of the instruction cache in EDGE architectures can be significantly improved through simple techniques such as prefetching.

This dissertation makes the following contributions:

- We quantify the tradeoffs in prediction timeliness, accuracy, and coverage when dead-block predictions are made at different points during the dead time of a block and find that making dead-block predictions when a block just moves out of the MRU position gives the best tradeoff.

- We formulate the concept of cache bursts, which exploits mechanisms already in the cache efficiently and matches well with the characteristics of memory accesses in set-associative L1 caches.

- For the L1 cache, we propose dead-block predictors that make predictions based on cache bursts instead of individual references. Cache bursts hide irregular cache access patterns within a burst and are more predictable than individual references.

- To mitigate the effects of reference-count variations, which cause prior counting-based dead-block predictors to have lower prediction accuracy and coverage, we propose two mechanisms to improve counting-based dead-block predictors: (1) filter out sporadic smaller reference counts; and (2) use more up-to-date reference-count history information.

- Using the dead-block information, we evaluate three optimizations to increase cache efficiency by eliminating dead blocks early: replacement optimization,

cache bypassing, and prefetching into dead blocks. Prior work only uses one particular dead-block prediction scheme for a subset of these optimizations.

- We present a working design and implementation of a distributed instruction cache for TRIPS, a microarchitecture instantiation of EDGE architectures, which has been prototyped in an actual chip.

- We propose how to implement look-ahead instruction prefetching by decoupling the next-block prediction from the instruction fetch in a distributed microarchitecture. We investigate the tradeoffs involved in choosing the right run-ahead distance and identify the run-ahead distance that gives the best performance improvement.

- We present an I-cache design that can store variable-sized blocks in the I-cache with low hardware complexity and estimate the potential performance benefit of such a design.

1.4 Dissertation Organization

The rest of the dissertation is organized as follows.

Chapter 2 investigates how to identify dead blocks early through dead-block prediction. We investigate what is the best time to make dead-block predictions and what information the dead-block predictor should maintain to make predictions. We propose new dead-block predictors for the L1 cache and the L2 cache with improved prediction accuracy and coverage over prior dead-block predictors. For the L1 cache, we formulate the concept of cache bursts, which hide the irregularity of individual references, and propose dead-block predictors that make predictions based on cache bursts. We also propose mechanisms to improve the prediction

coverage and accuracy of counting-based dead-block predictors proposed by prior work.

Chapter 3 studies how dead-block prediction can be used to improve data cache efficiency. We use dead-block prediction to evict dead blocks early by bypassing zero-reuse blocks and choosing dead blocks over LRU blocks for replacement. We also use dead-block prediction along with prefetching at both L1 and L2 to improve the effectiveness of prefetching.

Chapter 4 presents the design and implementation of the distributed instruction cache in the TRIPS prototype. It also extends the TRIPS I-cache design to TFlex, a microarchitecture instantiation of EDGE architectures that shares the same instruction set architecture as TRIPS but addresses many of its limitations.

Chapter 5 studies how to improve the I-cache efficiency in EDGE architectures through look-ahead instruction prefetching and storing variable-sized blocks in the I-cache. We discuss how look-ahead instruction prefetching can be implemented by decoupling the next-block prediction from the instruction fetch in a distributed microarchitecture and identify the run-ahead distance that gives the best performance improvement. We also discuss how to store variable-sized blocks in the I-cache and evaluate its potential performance improvement.

Chapter 6 presents the conclusions of this work and discusses potential future work on how to make better use of the space occupied by dead blocks in the cache.

Chapter 2

Identifying Dead Blocks Early through Dead-block Prediction

The root cause of low cache efficiency is that dead blocks stay in the cache for too long. As a first step to increase cache efficiency, the dead blocks must be identified early, for which we use dead-block prediction.

One question about dead-block prediction is when to predict the death of a block. Most prior dead-block predictors predict the death of a block immediately after the block is accessed, as shown in Figure 2.1(a), which shows a sequence of accesses to three blocks, A, B, and C, in the same set of a two-way associative cache. P(A) in the figure indicates a prediction about whether block A has died. While this approach identifies dead blocks as early as possible, it sacrifices prediction accuracy and coverage because a block just accessed may be accessed again soon. There is a tradeoff between the timeliness and accuracy/coverage of dead-block prediction. The earlier the prediction is made, the more useful it is. On the other hand, the later the prediction is made, the less likely it is to mispredict. In this chapter, we

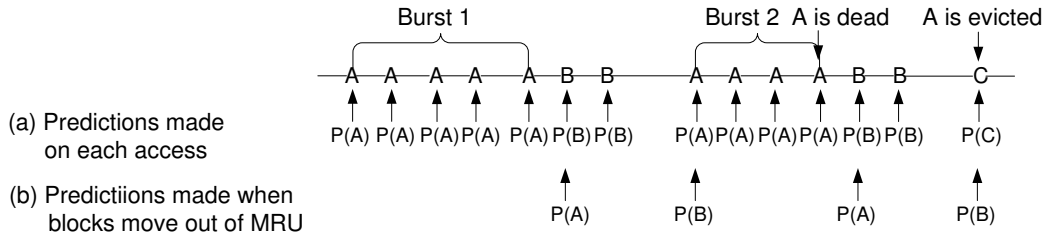


Figure 2.1: Predicting dead blocks at different times

quantify this tradeoff by making dead-block predictions at different points during the dead time of a block.

Another question about dead-block prediction is what information the predictor should maintain to make dead-block predictions. Prior dead-block predictors maintain the history about individual references to a block and use this history to make future predictions. However, for the L1 cache, how a block is accessed may depend on the control-flow path the program takes, the value or offset of the referenced data in the block, and other parameters. These variations can cause the individual reference history of a block to be irregular and cause problems for existing dead-block predictors. To address this problem, we propose a new class of dead-block predictors for the L1 cache that predict dead blocks based on the *cache burst* history of each block. A cache burst begins when a block moves into the MRU position and ends when it moves out of the MRU position. In these new dead-block prediction schemes, the contiguous references a block receives in the MRU position are grouped into one cache burst. In Figure 2.1, block A receives two cache bursts. A prediction about whether a block has died is made only when it becomes non-MRU, using the block's cache burst history. Because cache burst history hides the irregularity in individual references, it is easier to predict than individual reference history for L1 caches.

One category of dead-block predictors, counting-based predictors, have been shown to work better than other dead-block predictors at the L2 cache [39]. However, the effectiveness of counting-based predictors proposed by prior work can be severely affected by reference-count variations. We propose several mechanisms to mitigate the effect of reference-count variations on counting-based predictors and show that they result in better dead-block prediction accuracy and coverage.

Dead-block prediction can be made either based on the address of the referenced block or the program counter (PC) of the instruction that references a block. Using the block address for dead block prediction requires much larger storage overhead because usually the number of cache blocks a program references is much larger than the number of static loads and stores in a program. Therefore, most prior dead block predictors use PCs of loads and stores to make predictions [4, 17, 29, 37, 39, 48, 53]. For the same reason, we only investigate PC-based dead-block predictors in this chapter.

2.1 Related Work

Several schemes have been proposed to predict dead cache blocks. Based on how dead-block predictions are made, these schemes can be classified into three categories: trace-based, counting-based, and time-based. Trace-based predictors [17, 48, 53] record the sequence of instructions that have referenced a block while it is in the cache and predict a block dead once it has been accessed by the same sequence of instructions the next time the block is brought into the cache. Counting-based predictors [39, 40] record how many times a block has been referenced in the cache and predict a block dead once it been accessed the same number of times the next time the block is brought into the cache. Unlike trace-based predictors, counting-

based predictors do not care which instructions have referenced a block. Time-based predictors [4, 29, 37] record either the live time of a block or the number of cycles between two consecutive accesses to a block and predict a block dead if it has not been referenced for a certain number of cycles.

Trace-based Predictors: For uniprocessors, Lai et al. are the first to propose the concept of dead-block prediction [48] and a trace-based dead block predictor for the L1 data cache, called DBP. Because we use DBP in this document to refer to dead-block prediction in general, to avoid confusion, we use the name Reference Trace Predictor (RefTrace) to refer to the predictor proposed in [48]. RefTrace records the sequence of instructions that have referenced a block by hashing the PCs of these instructions together. A history table is used to learn the trace values (sequences of references) that result in dead blocks by observing the trace value of each evicted block. Blocks brought into the cache by the same instruction but referenced along different paths will have different trace values upon eviction. The different sequences of references conceptually form a tree embedded in the history table, with the root of the tree being the instruction that caused the miss and each leaf indicating dead blocks. Each entry in the history table indicates the likelihood that the corresponding trace value will result in a dead block. Aliasing can occur if a reference sequence, which results in dead blocks in some cases, is a prefix of other longer sequences.

Counting-based Predictors: Kharbutli et al. later proposed a counting-based dead-block predictor, Live Time Predictor [39], for L2 caches. In this document, we use the name RefCount to denote that it is a counting-based predictor. In RefCount, each block in the cache is augmented with a counter which records how

many times the block has been referenced and the PC of the instruction that first brought the block into the cache. When the counter reaches a threshold value, the block is predicted dead. The threshold is dynamically learned using a history table by observing the reference count and recorded PC of each evicted block. The index into the history table is a hash of the PC recorded in a block and the block address. Compared to RefTrace, RefCount uses only the PC of the instruction that brought a block into the cache to make predictions, and can not distinguish blocks that are brought into the cache by the same instruction but are referenced by different instruction sequences.

Time-based Predictors: Hu et al. proposed a time-based dead-block predictor, Timekeeping (TK) [29], for the L1 cache. TK dynamically learns the number of cycles a block stays alive and if the block is not accessed in more than twice this number of cycles, it is predicted dead. Abella et al. proposed [4] another time-based dead-block predictor for the L2 cache. They observed that both the inter-access time between hits to the same block and the dead time correlate with the reference counts of a block. They also predict a block dead if it has not been accessed in a certain number of cycles, but the cycle count is derived from how many times the block has been accessed.

Compared to trace-based and counting-based predictors, time-based predictors are more complex to implement in hardware and incur more overhead for the following reasons. First, on a cache access, trace-based and counting-based predictors only need to check the cache block being accessed to make a prediction about whether it has died. In contrast, time-based predictors need to check all the cache blocks in the same set as the block being accessed to determine if they have died. Second, besides the PC, time-based predictors need to keep track of the number of

cycles between two accesses to the same block, which can be large and requires more bits to store, especially for the L2 cache. In contrast, besides the PC, counting-based predictors only need to keep track a reference count, which only require several bits whereas trace-based predictors do not need to keep track of any other information besides the trace.

Besides the implementation complexity and overhead considerations, the traces and reference counts of blocks are more closely correlated to the memory reference behavior of a program than the number of cycles between accesses to the same block.

Because of these reasons, this dissertation only considers trace-based and counting-based dead-block predictors.

2.2 Cache Bursts: Tolerating Irregularity of Individual References in the L1 Cache

In this section, we investigate dead-block prediction for set-associative L1 data caches. Accesses to the L1 and L2 caches have different characteristics and these characteristics should be considered when designing dead-block predictors for each cache level. One characteristic of L1 accesses is that several references to the same cache block are usually clustered together because of temporal or spatial locality. Another characteristic is that the accesses a block receives in the L1 cache are frequently affected by control and data dependences. We show how these access characteristics can cause problems for prior dead-block predictors and propose a simple but effective mechanism to achieve better dead-block prediction accuracy and coverage.

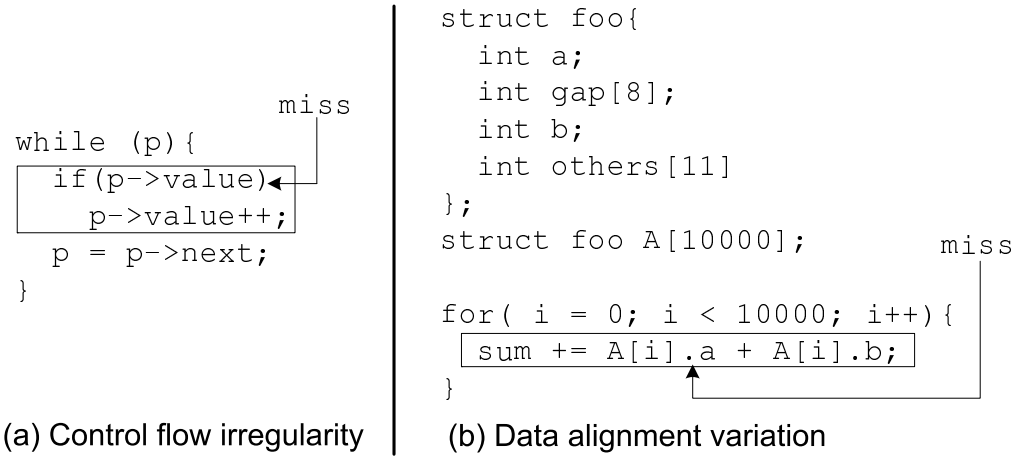


Figure 2.2: Examples of individual reference history irregularity in L1 accesses

2.2.1 Irregularity of Individual References in L1 Accesses

All prior dead-block predictors try to find regular patterns in the individual reference history of each block. However, individual reference histories can be volatile and irregular because how a block is accessed may depend on the control flow path the program takes, the value or offset of the referenced data in the block, and other parameters, all of which can change dynamically and may not show any regular pattern (RefTrace can handle control flow dependence to some extent). While this irregularity of individual references can be filtered out by the L1 cache and may not be observed by the L2 cache, it can occur frequently at the L1 cache. Figure 2.2 shows two examples of reference variance.

Figure 2.2(a) shows how control-flow irregularity can lead to irregular reference history. Suppose the first access to $p \rightarrow value$ always misses and $p \rightarrow value$ will not be referenced after the iteration. Depending on whether $p \rightarrow value$ is zero, the block which has $p \rightarrow value$ can be accessed either once or twice. However, it is not possible to find a regular pattern in the individual reference history of each block

because some of the blocks are referenced only by the load instruction while others are referenced by both the load and the store.

Figure 2.2(b) shows how data alignment variation can cause the same problem. Suppose the cache block size is 64 bytes and the access to *A[i].a* always misses. Because of data alignment differences, *A[i].a* and *A[i].b* can be located in the same block or in two adjacent blocks. If they are located in the same block, the block will be accessed twice before eviction. Otherwise, the block that has *A[i].a* will only be accessed once. Again, it is not possible to find a regular pattern in the individual reference history of each block that has *A[i].a* because some blocks will be accessed only by one load instruction and others will be accessed by both loads.

This irregularity in individual reference history can cause problems for existing dead-block predictors: neither RefCount nor RefTrace can handle the two examples in Figure 2.2 well because neither can predict exactly after which access a block becomes dead.

2.2.2 Grouping References into Bursts

The problem with trying to find regular patterns in the individual reference history of each block is that the predictor observes events at excessively fine granularity. If we increase the granularity at which the predictor observes the events, it may be able to find regular patterns not observable at the finer granularity. Because L1 cache accesses tend to be bursty in the sense that several accesses to the same block are usually clustered in a short interval, an effective strategy is to predict dead blocks by cache bursts instead of individual references. We formally define cache bursts as follows:

Definition A *cache burst* is the contiguous group of cache accesses a block receives

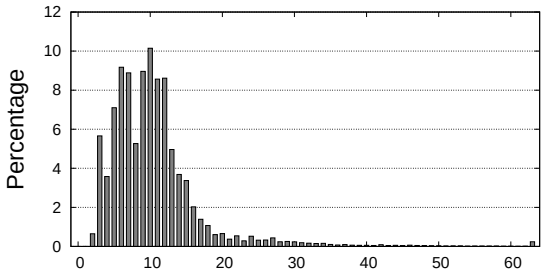


Figure 2.3: Reference count distribution

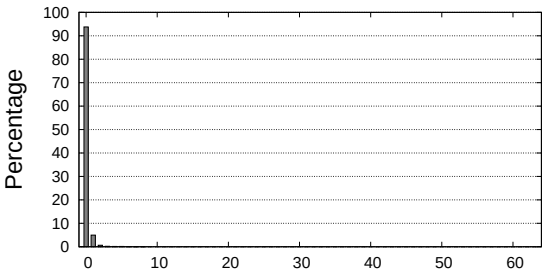


Figure 2.4: Burst count distribution

while it is in the MRU position of its cache set with no intervening references to any other block in the same set.

Although the references within a cache burst may be irregular, the cache-burst history can still be regular. Examining the two examples using bursts, there still is a regular pattern. In Figure 2.2(a), the block containing $p \rightarrow value$ will become dead after exactly one cache burst, regardless of whether $p \rightarrow value$ is zero. In Figure 2.2(b), the block containing *A[i].a* will also become dead after exactly one cache burst, regardless of whether *A[i].b* is located in the same block.

The experiment results in Figure 2.3 and 2.4 confirm that cache bursts are more regular than individual references. Figure 2.3 shows the reference count distribution of the blocks brought into the L1 data cache by the same instruction in *sphinx* [51]. This particular instruction causes the most misses in the L1 data cache. The X axis is the reference count. The Y axis shows for a given reference count, what percentage of the blocks (out of all the blocks brought into the cache by this instruction) receive that number of references before getting evicted from the L1 cache. Figure 2.4 shows the corresponding burst count distribution for the same instruction. The figures indicate that burst count is much more predictable than

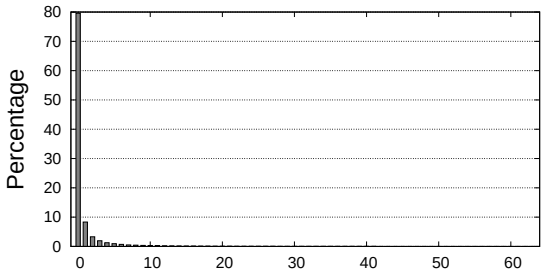


Figure 2.5: Average reference count distribution. X axis is the rank of each component, not the actual reference count.

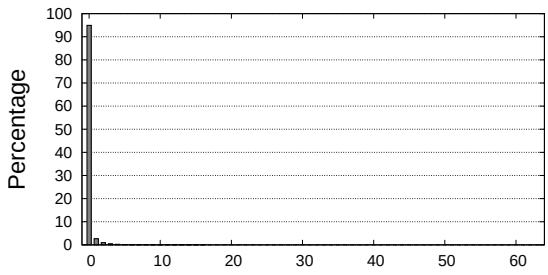


Figure 2.6: Average burst count distribution. X axis is the rank of each component, not the actual burst count.

reference count in the L1 data cache.

Figure 2.5 and Figure 2.6 show the average reference count and burst count distribution of all the instructions across all the single-threaded benchmarks we study. The average distribution is derived as follows. First, we get the reference count (burst count) distribution for each instruction in each application. Then we sort the bars in each instruction's distribution in descending order of their heights. After the sort, we lose the actual reference count information: the X axis is not the reference count any more, instead, it is the rank of each bar in the sorted distribution of each instruction. Then we compute the weighted average of the bars with the same rank in each sorted distribution. The weight of an instruction is the total number of blocks brought into the cache by the instruction. After we get the average distribution of one application, we compute the average across all applications with equal weight. Figures 2.5 and 2.6 give a global view of the reference count and burst count distribution and they confirm that burst count is more predictable than reference count in the L1 data cache.

2.2.3 Burst-based Dead-block Predictors

Because cache bursts are easier to predict than individual references, we propose a new class of dead-block predictors that predict based on cache bursts instead of individual references of each block. These burst-based predictors are adapted from the existing dead-block predictors that predict dead blocks based on the individual reference history of each block. Cache bursts begin when a block moves into the MRU position and end when it moves out of the MRU position, at which point a dead block prediction is made, typically $1/n^{th}$ into the dead time, where n is the set associativity.

Burst-Count Predictors

A Burst-Count Predictor (BurstCount) uses the same structure as a reference counting predictor, like the RefCount predictor [39] or the RefCount+ predictor discussed later in Section 2.3, except that it counts cache bursts instead of individual references. In this dissertation, we use BurstCount to refer to the burst-based predictor derived from the RefCount+ predictor, which is an improved version of the RefCount predictor [39]. When a block is filled into the MRU position of its set, its burst count is set to 0. Unlike a reference counting predictor, which increments the reference count every time a block is accessed, the burst count is incremented only when the block moves from a non-MRU position into the MRU position. If the block is accessed in the MRU position, the burst count does not change. A prediction is made only when a block becomes non-MRU.

Besides the higher dead-block prediction accuracy and coverage, as shown later in Section 2.5, BurstCount also consumes less energy than RefCount. RefCount needs to read the history table and update the reference count stored in the accessed

block on every cache hit. In contrast, BurstCount only reads the history table when a block moves out of the MRU position and updates the reference count when a block moves into the MRU position.

Burst-Trace Predictors

Similarly, a Burst-Trace Predictor (BurstTrace) uses the same structure as a reference-trace predictor [48]. The difference is how the trace is constructed. A reference-trace predictor constructs a trace of individual references whereas a burst-trace predictor constructs a trace of bursts. In RefTrace, every time a block is accessed, the PC of the load/store instruction is hashed into the trace stored along with the block. In BurstTrace, the PC of a load/store instruction is hashed into the trace only if the access starts a new burst. Therefore, the trace value of a block is updated only when it is first brought into the cache and when it moves from non-MRU position into the MRU position. If it is accessed in the MRU position, the trace does not change. When a block moves out of the MRU position, its trace value is checked to determine if the block has died.

Like BurstCount, BurstTrace has higher prediction accuracy and coverage but consumes less energy than RefTrace. It also has another advantage over RefTrace. In RefTrace, if some blocks are accessed very frequently in the cache, they will generate a large number of different trace values. These different trace values will pollute the history table but do not provide much useful information. BurstTrace can avoid or reduce such an effect because it generates fewer trace values.

Discussions about Burst-based Predictors

The introduction of cache bursts adds a new dimension to the design space of dead-block predictors. Based on the metric used to make dead-block predictions, dead-

		By references	By bursts
Prediction metric	Trace	RefTrace [48]	BurstTrace
	Counting	RefCount [40]	BurstCount
	Time	TimeKeeping [29], IATAC [4]	Future work

Table 2.1: A taxonomy of dead-block prediction schemes

block predictors can be classified into trace-based, counting-based, and time-based. Based on how the state of a block is updated, dead-block predictors can be classified into reference-based and burst-based. Table 2.1 classifies the possible dead-block predictors using this taxonomy.

Burst-based predictors have the following limitations:

- The burst concept is not applicable to directly-mapped caches. For directly-mapped caches, a block becomes dead whenever there is a reference to a different block in the same set of the cache.
- There is no additional benefit of using the burst history over the reference history at the L2 cache because accesses to the L2 cache are already filtered by the L1 cache. In fact, the L1 cache may filter out more than one burst of a cache block and cause the L2 cache to observe oven fewer accesses to a block.

2.3 Improving Counting-based Dead-block Predictors

While burst-based predictors work well for the L1 cache, they do not benefit the L2 cache because most of the irregularity in individual references has already been filtered out by the L1. Prior work [39] found counting-based predictors are better suited for the L2 than trace-based predictors because the filtering effect of the L1 prevents trace-based predictors from seeing the complete reference history of a block.

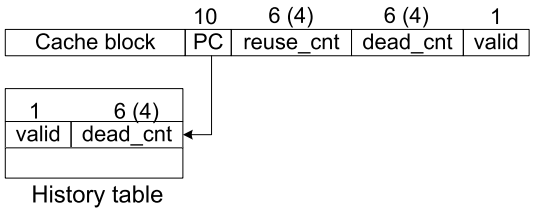


Figure 2.7: Structure of the RefCount Predictor

One problem with counting-based dead-block predictors is reference count variation: blocks brought into the cache by the same instruction can receive different number of references in the cache. One cause for reference count variations is control flow irregularity. For example, two cache blocks referenced by the same load/store instruction can subsequently have different reference patterns depending on the control flow path taken by the program. Another cause for reference count variations is that some cache sets may have more conflict misses than others.

To deal with variations in reference counts associated with the same instruction, the RefCount predictor [39] uses a confidence bit (the *valid* bit in Figure 2.7) to decide whether an earlier reference count should be used in later predictions: when a block is evicted from the cache, its reference count (*reuse_cnt*) is compared with the threshold (*dead_cnt*) stored in the history table. The confidence bit is set if the new reference count equals the old threshold and cleared otherwise. When a new block is filled into the cache on a cache miss, the threshold and the confidence bit is copied from the entry in the history table into the block. When the block is accessed later, the reference count is incremented and a prediction is made based on the threshold and the confidence bit in the block. The threshold in the history table will not be used for prediction if the confidence bit is cleared. Figure 2.7 shows the diagram of the RefCount predictor.

There are two problems with this scheme, both of which are caused by ref-

erence count variations. The first problem is caused by a smaller reference count following a larger reference count and the second problem is caused by a larger reference count following a smaller one.

2.3.1 Filtering Temporary Small Reference Counts

When a smaller reference count follows a larger one, RefCount will clear the confidence bit and stop predicting until two contiguous reference counts of the same value are observed. In an extreme situation when the reference count alternates between two different values, the confidence bit will never be set and no predictions will ever be made. Obviously, clearing the confidence bit in such cases may be unnecessary and reduce prediction coverage. A better way to handle such cases is to continue to use the larger reference count as the threshold without clearing the confidence bit.

To address this problem caused by smaller reference count, we use a counter, *filter_cnt* shown in Figure 2.8, to filter out noise (smaller reference count) so that the confidence bit is not cleared unnecessarily. When the predictor sees a new reference count on a cache eviction, it compares the new value with the current threshold. If the new reference count is smaller than the current threshold, the predictor will continue to use the current threshold and ignore the new reference count.

While the *filter_cnt* can filter out infrequent small reference counts, it also prevents the predictor from switching to a smaller reference count once a larger value has been established as the threshold. This is undesirable because it prevents the predictor from adapting to the dynamic changes in the memory access behavior of a program. To solve this problem, we use a saturating counter, *sat_cnt* shown in Figure 2.8, to determine when the threshold should be changed. The idea is that if the predictor sees a smaller reference count, it will switch to the smaller value only if

the smaller value appears several times, i.e., when the saturating counter saturates.

2.3.2 Sensitivity to Large Reference Counts

While a counting-based predictor should show some hysteresis when it sees a smaller reference count following a larger one, it should be very sensitive when the contrary happens. That is, as soon as the predictor observes that the current threshold is too small, it should immediately discard the current threshold or establish a new one if possible. Otherwise, it will make premature dead-block predictions when cache blocks are not dead and result in lower prediction accuracy. We discuss two potential problems that can arise when a larger reference count follows a smaller one in RefCount.

Early Detection

The first problem is related to when the predictor should determine that the current threshold is too small. RefCount does not detect this until a block is evicted from the cache, even if the reference count of the accessed block already exceeds the threshold in the history table. Detecting the switch to a larger threshold at cache eviction time may be too late and can cause more mispredictions. In fact, a switch to a larger reference count can be detected much earlier. For example, on a cache hit to a block, if the block has already been accessed five times but the corresponding threshold in the history table is only four, it is better to clear the confidence bit so that future predictions do not use four as the threshold, even though the exact value of the new threshold is not known at this time. This early detection mechanism makes the predictor respond to mispredictions more quickly and improves prediction accuracy.

Using Up-to-date History Information

The second problem is whether the predictor is using the most up-to-date history information to make predictions. As shown in Figure 2.7, in RefCount, each block copies the threshold and confidence bit from the history table when the block is filled into the cache and uses the copied information to make predictions thereafter. However, the threshold and confidence bit stored in each block can become outdated as the history table gets updated. For example, when a block is filled into the cache, the predicted reference count for the block may be three. But later the predicted reference count may change to four. If the predictor still uses three as the predicted reference count for the block, it may predict the block dead too early. A better approach is to remove the threshold and confidence bit stored in each block, as shown in Figure 2.8. Instead, when the predictor makes a prediction, it reads the threshold and confidence bit from the history table, which has the most up-to-date information. Removing the per-block threshold and confidence bit also saves area. For a 1MB L2 cache with 64B blocks, this results in a savings of 80K bits, about 28% of the total overhead of the RefCount predictor. Of course, the history table will be accessed more frequently. The increased accesses to the history table adds little energy overhead because the L2 cache is accessed only when L1 caches miss. Additionally, when used at the L1 cache, the frequency of history table lookups are mitigated in the burst scheme because predictions are made only when a block becomes non-MRU.

2.3.3 RefCount+: An Improved Counting-based Dead-block Predictor

With the inclusion of the changes discussed above, we call the resulting predictor RefCount+, which is inspired by RefCount [39] but addresses the problems caused by reference count variations. Figure 2.8 shows the major structures of RefCount+.

Figure 2.9 shows how RefCount+ works. *CacheFillAction* is performed whenever a block is filled into the cache. *CacheHitAction* increments the reference count on every cache hit. *Predict* is performed to check if a block is dead. It clears the confidence bit after the prediction if the current reference count exceeds the threshold in the history table. *CacheEvictAction* shows how the dead block predictor is trained. *CacheEvictAction*is performed on cache evictions and it updates the threshold, confidence bit (*valid*), *filter_cnt* and *sat_cnt*. The *CacheEvictAction* algorithm is a major difference between RefCount+ and RefCount. In RefCount, when a block is evicted, the predictor only compares the new reference count with the current threshold and sets or clears the confidence bit depending on the result of the comparison. The current threshold is then changed to the new reference count. In RefCount+, the *filter_cnt* and *sat_cnt* record extra state information so the new value of the confidence bit and the threshold also depend on the current value of

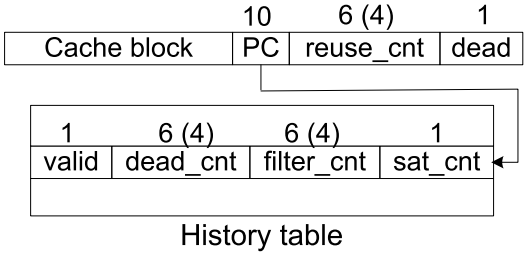


Figure 2.8: Structure of the RefCount+ Predictor

filter_cnt and *sat_cnt*.

CacheFillAction: when a block, B, is filled into the cache, initialize reuse counter and record PC which caused the miss.

```
B.pc = hash of PC of the instruction
       which caused the miss;

B.reuse_cnt = 0;
B.dead = 0;
```

CacheHitAction: when a block, B, is accessed, increment reuse counter.

```
B.reuse_cnt++;
```

Predict: Predict using information from history table (For L1 cache, this is performed when a block moves out of the MRU position). Update history table if necessary.

```
Access entry E from history table using
index B.pc;

if (E.valid and B.reuse_cnt >=
    E.dead_cnt) then
  B.dead = 1;
end

if(B.reuse_cnt > E.dead_cnt)then
    E.valid = 0;
end
```

CacheEvictAction: when a block, B, is evicted, update entry E in history table indexed by B.pc.

```
if(!E.valid) then
  E->sat_cnt++;
  if(E.filter_cnt < B.reuse_cnt)then
    if(E.sat_cnt == 1) E.sat_cnt--;
    E.filter_cnt = B.reuse_cnt;
  end
  if(E.sat_cnt == 1)then
    E.dead_cnt = E.filter_cnt;
    E.valid = 1;
    E.filter_cnt = B.reuse_cnt;
    E.sat_cnt = 0;
  end
else
  if(B.reuse_cnt > E.dead_cnt)then
    E->dead_cnt = B.reuse_cnt;
    E->sat_cnt = 0;
  end
  else
    if(B.reuse_cnt == E.filter_cnt)then
      E.sat_cnt++;
      if(E.sat_cnt == 1)then
        E.dead_cnt = B.reuse_cnt;
        E.sat_cnt = 0;
      end
    end
    else
      E.sat_cnt = 0;
      E.filter_cnt = B.reuse_cnt;
    end
  end
end
```

Figure 2.9: Detailed Implementation Algorithms for RefCount+

2.4 Accuracy vs. Timeliness: When to Predict

One question not answered by prior work on dead-block prediction is when dead-block predictions should be performed. The dead time of a block begins with the last access to the block and ends with its eviction from the cache. Dead block prediction can be made at any point in this interval. Almost all prior dead-block prediction schemes predict whether a block has died immediately after it is referenced, when the block is still in the MRU position. Higher prediction accuracy and coverage can

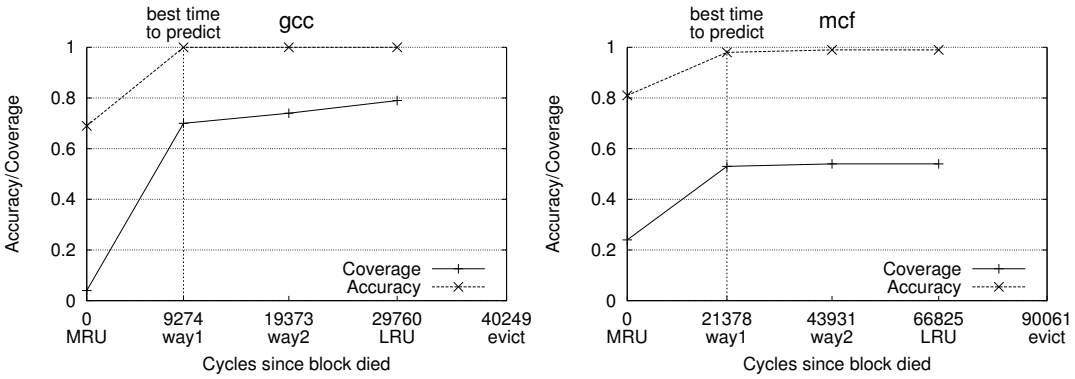


Figure 2.10: Prediction accuracy/coverage when predictions are made at different depths of the LRU stack for a 4-way L1 cache

be achieved if dead-block predictions are made later because it is less likely to make premature predictions. At the same time, predictions made closer to the end of a block's dead time are less useful because they leave more of the dead time exposed.

Figure 2.10 shows the accuracy and coverage of the RefCount+ predictor when dead-block predictions are made at different depths of the LRU stack after a block's last access. The results are obtained using *sim-alpha* with a 4-way, 64KB L1 cache. Other parameters of the simulation are listed in Table 2.2. For a 4-way set-associative cache, a block is placed in the MRU position when it is accessed for the last time before its eviction from the cache. Afterwards, it moves down the LRU stack until it is evicted from the cache. The X axis shows the average number of cycles between the last access to a block and its movement into each position of the LRU stack. The last number on the X axis is the average number of cycles between the last access to a block and its eviction from the cache, i.e., the dead time. As expected, accuracy increases as predictions are made later. Coverage also increases because delaying the prediction does not miss any opportunity to identify dead blocks and the increase in accuracy causes more dead blocks to be

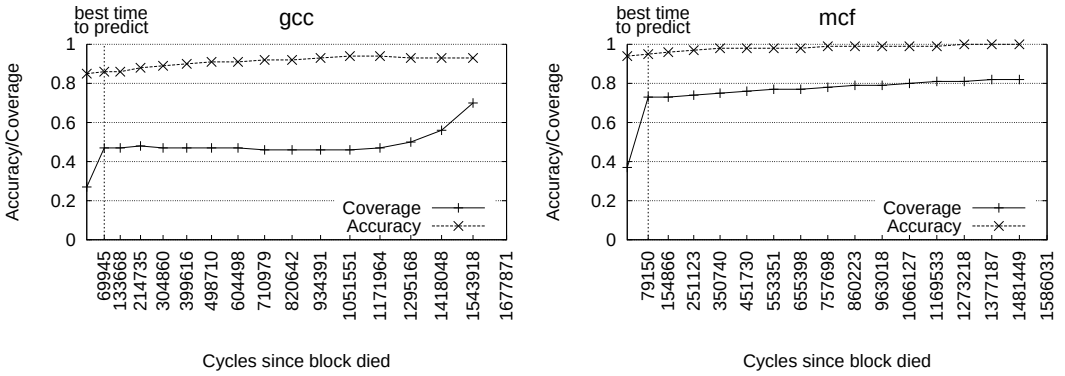


Figure 2.11: Prediction accuracy/coverage when predictions are made at different depths of the LRU stack for a 16-way L2 cache

correctly identified. The "knee" of the curves is located at way one of the LRU stack, indicating that making predictions when a block just becomes non-MRU gives the best tradeoff between timeliness and accuracy/coverage. For a 4-way set-associative cache, the time the predictor loses by delaying prediction until a block becomes non-MRU is about $1/4^{th}$ of the dead time.

Figure 2.11 shows the prediction accuracy and coverage of the RefCount+ predictor for a 16-way L2 cache when dead-block predictions are made at different positions of the LRU stack. The capacity of the cache is 1MB. Again, the prediction accuracy and coverage increase as predictions are made later. In a 16-way L2 cache, the interval between the time a block moves into position one of the LRU stack and the time it moves into the LRU position is long and the difference in prediction accuracy and coverage when dead-block predictions are made at these two positions is larger compared to the L1 cache. For a 16-way set-associative cache, the time the predictor loses by delaying prediction until a block becomes non-MRU is about $1/16^{th}$ of the dead time.

2.5 Results

In this section, we compare the overhead and prediction accuracy and coverage of each dead block predictor and find the best predictor for the L1 data cache and L2 cache respectively. Prediction accuracy is calculated as the number of correct dead block predictions divided by the total number dead block predictions ever made by each predictor. Prediction coverage is calculated as the number of blocks evicted from the cache that are correctly predicted dead divided by the total number of cache evictions. A block is correctly predicted dead if when it is evicted from the cache, it has already been predicted dead by the dead block predictor. A block which is not predicted dead when evicted from the cache, but was prematurely predicted dead before the last access, is not counted as a correctly predicted block.

2.5.1 Methodology

We evaluate the prediction accuracy and coverage of each dead-block predictor by simulating both single-threaded benchmarks running on a single processor and multi-threaded benchmarks running on a chip multiprocessor.

Simulators & Benchmarks

The simulator used for single-threaded benchmarks is *sim-alpha* [16], which is a cycle-accurate, execution-driven simulator that simulates an Alpha 21264 processor [38]. *sim-alpha* can only simulate user-level instructions so it does not model any instructions executed by the operating system. The parameters of the simulated machine are shown in Table 2.2.

Besides the 11 benchmarks from SPEC [93] 2000, we also use two benchmarks from Versabench [72] (*corner_turn* and *vpenta*), a speech recognition application

Issue width	6-way out of order(4 integer, 2 floating point)
Inst. window	80-entry reorder buffer, 32-entry Load/Store queue each
L1 I-cache	64KB, 2-way LRU, 64B cacheline, 1-cycle w/ set prediction
L1 D-cache	64KB, 2-way LRU, 64B cacheline, 3-cycle
L2 cache	1MB, 16-way LRU, 64B cacheline, 12-cycle
Main memory	200-cycle, 16B bus width

Table 2.2: Configuration of simulated SP machine

(*sphinx* [51]), and *stream* [59]. For each benchmark, we simulate up to 2 billion instructions identified by SimPoint [84].

The simulator used for multi-threaded benchmarks is MP-sauce [31]. MP-sauce is a cycle-accurate, execution-driven, full-system simulator derived from IBM's SimOS-PPC. AIX 4.3 runs on the simulator as the simulated OS. We simulate a 16-way CMP. The timing model of each processor core is based on *sim-outorder* in SimpleScalar with changes to model CMPs. The parameters of the simulated machine are shown in Table 2.3.

# of processors	16
Issue width	4-way out of order
Instruction window	64-entry RUU, 32-entry Load/store queue
L1 I-cache	64KB, 2-way LRU, 64B cacheline, 2-cycle
L1 D-cache	64KB, 2-way LRU, 64B cacheline, 2-cycle
L2 cache	1MB private per core, 8-way LRU, 64B cacheline, 13-cycle
Coherence protocol	Snoop-based MOESI
Main memory	200-cycle

Table 2.3: Configuration of simulated MP machine

We evaluate three server benchmarks (SPECWeb99, TPC-W, and SPECjbb) and five scientific applications from SPLASH-2 [101]. Table 2.4 shows the application parameters for these multi-threaded workloads.

Application	Dataset/Parameters
SPECweb	Apache web server, file set: 230MB
SPECjbb	IBM JVM version 1.1.8, 16 warehouses
TPC-W	185MB databases using Apache & MySQL
barnes	16K particles
FFT	262144 data points
lu	512×512, 16×16 blocks
ocean	258×258 grid
radix	1M integers

Table 2.4: Application parameters for multi-threaded workloads

Because of the cache coherence protocol used in a chip multiprocessor, the definition of correctly predicted block is slightly different from the previous definition which does not consider the effect of the cache coherence protocol. When considering the cache coherence protocol, a block is correctly predicted dead if when it is evicted from the cache or when it is invalidated by the cache coherence protocol, whichever happens first, it has already been predicted dead by the dead-block predictor. As a result, prediction coverage is calculated as the total number of correctly predicted dead blocks divided by the total number of cache blocks evicted from the cache or invalidated by the cache coherence protocol. Prediction accuracy is still the number of correct dead block predictions divided by the total number dead block predictions ever made by each predictor.

2.5.2 Dead-block Prediction at the L1 Cache

We compare the overhead as well as the prediction accuracy and coverage of different dead-block predictors for the L1 data cache. The dead-block predictors we evaluate include the RefTrace predictor proposed in [48], the RefCount predictor proposed in [39], the RefCount+ predictor, which addresses problems caused by reference

count variations, the BurstTrace predictor, which is a burst-based predictor derived from RefTrace, and the BurstCount predictor, which is also a burst-based predictor derived from RefCount+.

Predictor Overhead

Overhead	RefTrace	BurstTrace	RefCount	RefCount+	BurstCount
History table entries	1K	1K	2K	1K	1K
History table	0.25KB	0.25KB	1.75KB	1.75KB	1.75KB
Per-block (bits)	10	10	21	17	17
Total overhead	1.5KB	1.5KB	4.4KB	3.9KB	3.9KB
Relative overhead	2.3%	2.3%	6.9%	6.1%	6.1%

Table 2.5: Overhead of different dead-block predictors for a 64KB L1 cache

The overhead of each dead block predictor includes the overhead caused by the history table and the extra bits added to each block. The size of RefCount is scaled down from [40] to make it comparable with other predictors. It uses a 2K-entry history table; the index into the table is a hash with 8 bits from the PC and 3 bits from the block address. When calculating the predictor overhead, we assume a 64KB L1 D-cache with 64-byte blocks.

Table 2.5 shows the overhead of each predictor. The three counting-based predictors (RefCount, RefCount, BurstCount) cause an overhead of about 6%–7% of the L1 data cache capacity while the two trace-based predictors (RefTrace, BurstTrace) cause an overhead of less than 3%.

Prediction Accuracy and Coverage

Tables 2.6 and 2.7 list the prediction accuracy and coverage of each dead block predictor for the single-threaded benchmarks. We can draw several conclusions

Application	RefTrace	BurstTrace	RefCount	RefCount+	BurstCount
swim	0.96	1.00	0.90	1.00	1.00
mgrid	0.82	0.97	0.23	1.00	0.99
applu	0.74	0.96	0.18	1.00	0.99
gcc	0.94	0.99	0.49	1.00	0.99
art	0.95	0.99	0.85	1.00	0.99
mcf	0.82	0.97	0.71	0.99	0.98
ammp	0.69	0.90	0.40	0.95	0.95
lucas	0.92	0.98	0.64	1.00	0.99
parser	0.45	0.84	0.12	0.78	0.83
perlbmk	0.85	0.92	0.14	0.80	0.77
gap	0.77	0.98	0.19	1.00	0.99
sphinx	0.66	0.93	0.30	0.89	0.92
corner_turn	0.96	0.99	1.00	1.00	1.00
stream	1.00	1.00	1.00	1.00	1.00
vpenta	1.00	1.00	0.97	1.00	1.00
GeoMean	0.82	0.96	0.43	0.96	0.96

Table 2.6: Prediction accuracy of DL1 DBPs (Single-threaded benchmarks)

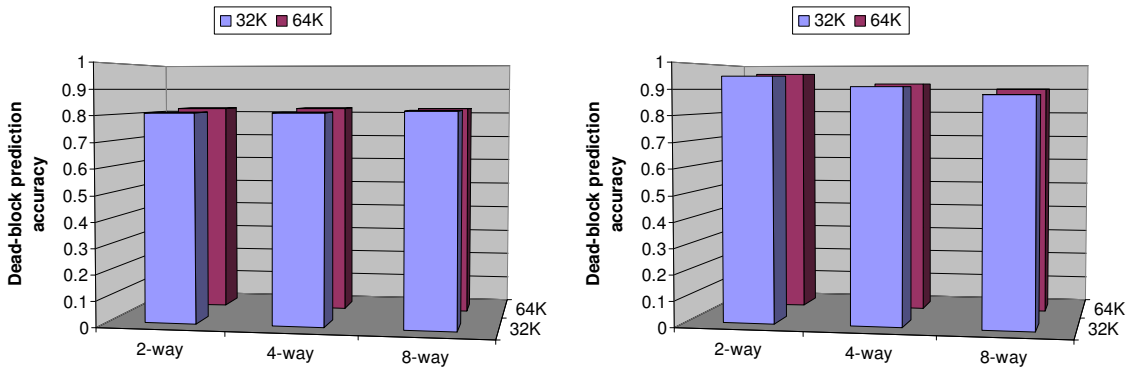
from Tables 2.6 and 2.7.

First, the two burst-based predictors (BurstTrace, BurstCount) significantly outperform the corresponding reference-based predictors (RefTrace, RefCount+): on average, BurstTrace makes 50% more correct predictions than RefTrace with higher accuracy and BurstCount makes 25% more correct predictions than RefCount+ with the same accuracy. The reason for the improvement of the burst-based predictors over the reference-based predictors is that burst history is more regular than individual reference history in set-associative L1 caches. The increased regularity of the burst-history increases dead-block prediction accuracy and coverage because the burst patterns are more predictable. The improvement in dead-block prediction accuracy and coverage also comes with much reduced power consumption and no increase in area.

Application	RefTrace	BurstTrace	RefCount	RefCount+	BurstCount
swim	0.90	1.00	0.78	0.97	1.00
mgrid	0.68	0.98	0.65	0.83	0.91
applu	0.45	0.98	0.75	0.78	0.95
gcc	0.65	0.97	0.69	0.74	0.93
art	0.96	0.99	0.91	0.90	0.97
mcf	0.75	0.99	0.47	0.54	0.93
ammp	0.54	0.94	0.56	0.68	0.77
lucas	0.90	0.97	0.99	0.96	0.88
parser	0.17	0.85	0.21	0.29	0.54
perlbmk	0.28	0.85	0.61	0.57	0.60
gap	0.42	0.96	0.39	0.41	0.88
sphinx	0.47	0.95	0.27	0.37	0.79
corner_turn	1.00	0.98	0.98	1.00	1.00
stream	1.00	1.00	1.00	1.00	1.00
vpenta	0.98	1.00	0.75	0.98	0.99
GeoMean	0.61	0.96	0.61	0.69	0.86

Table 2.7: Prediction coverage of DL1 DBPs (Single-threaded benchmarks)

Second, the improved counting-based predictor, RefCount+, has significantly higher prediction coverage and accuracy than RefCount: on average, RefCount+ makes 13% more correct predictions than RefCount with much higher accuracy (96% vs. 43%). The improvement in accuracy comes from two sources. First, RefCount+ makes a prediction only when a block moves out of the MRU while RefCount makes a prediction every time a block is referenced. If RefCount delays the prediction until a block becomes non-MRU, it will have an average prediction accuracy of 91%. Second, RefCount+ detects switches to larger reference counts more quickly to reduce the probability of prematurely predicting live blocks as dead. RefCount+ achieves higher prediction coverage because of its ability to filter out infrequent smaller reference counts, which reduces the frequency that dead-block prediction is stopped because of the occurrence of a temporary smaller reference count.



(a) RefTrace

(b) BurstTrace

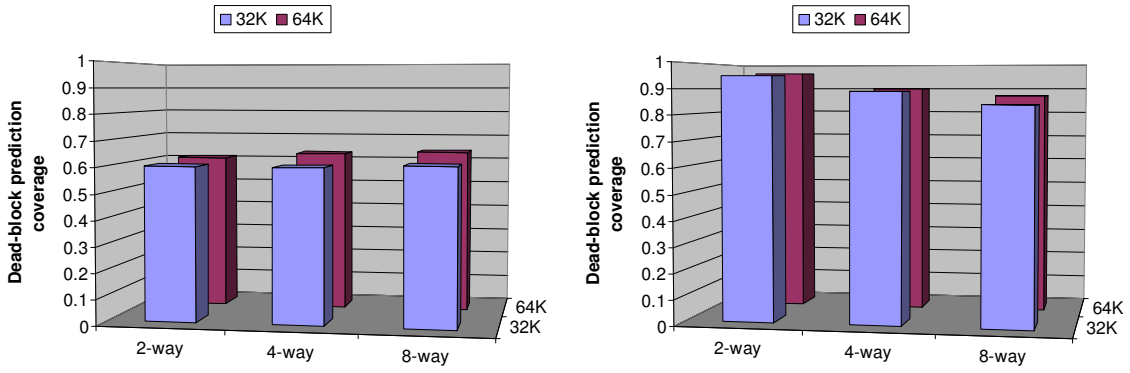
Figure 2.12: Dead-block prediction accuracy of RefTrace and BurstTrace with different cache configs

Third, between BurstTrace and BurstCount, BurstTrace has similar accuracy but much higher coverage. This is because BurstTrace can differentiate the different reference patterns of a block when it is accessed on different control flow paths. On the other hand, BurstCount does not record the sequence of the bursts on different control flow paths so unless the burst counts are the same, it is not able to make a prediction when different control flow paths interleave.

Last, of the five dead block predictors listed in Table 2.6 and 2.7, BurstTrace achieves the best accuracy and coverage. Furthermore, it also incurs the smallest overhead (Table 2.5). This makes BurstTrace the most appealing predictor for the L1 data cache.

Figures 2.12 and 2.13 show the dead-block prediction accuracy and coverage of the RefTrace predictor and the BurstTrace predictor with different cache sizes and associativities. For all configurations, BurstTrace has both higher accuracy and coverage than RefTrace.

The prediction accuracy and coverage of RefTrace remain mostly unchanged



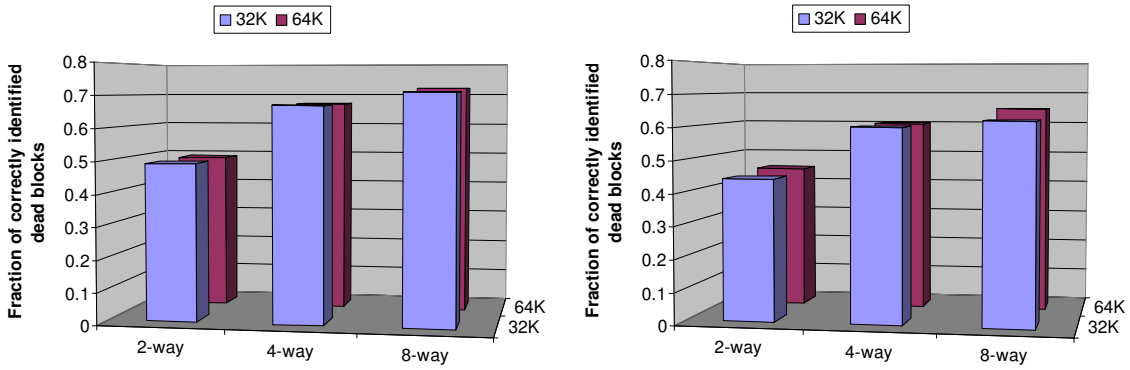
(a) RefTrace

(b) BurstTrace

Figure 2.13: Dead-block prediction coverage of RefTrace and BurstTrace with different cache configs

as the associativity increases. For BurstTrace, however, both the prediction accuracy and coverage drops slightly with higher associativity. The reason behind this trend is that as the associativity increases and the total number of sets decreases, a block will go through more bursts and the average length of a burst will become shorter. As a result, the benefits of a burst-based predictor in delaying dead-block predictions and hiding irregularity of individual references are reduced. On the other hand, a change in the the cache size has little effect on the the prediction accuracy and coverage of both RefTrace and BurstTrace, except that RefTrace has a slightly higher coverage when the cache size is increased from 32KB to 64KB.

For the BurstTrace predictor, while the overall dead-block prediction accuracy and coverage drop slightly with higher cache associativity, the fraction of the dead blocks that have been correctly identified as dead at any given moment actually increases. This trend is shown in Figure 2.14. Figure 2.14(a) shows the average fraction of the dead blocks that have been correctly identified as dead at any given cycle whereas Figure 2.14(b) shows the average fraction of all the blocks that has



(a) Out of all dead blocks

(b) Out of all blocks in the cache

Figure 2.14: Fraction of correctly identified dead blocks by BurstTrace at any given time

been correctly identified as dead. With a two-way cache, about 45% of all the blocks in the cache can be correctly identified as dead by BurstTrace at any moment. With a four-way cache, about 60% of all the blocks in the cache can be correctly identified as dead at any moment. With an eight-way cache, the fraction of the blocks that can be correctly identified at any moment increases to about 65%. The fraction of the blocks that have been correctly identified as dead at any moment increases with higher associativity because the time a block stays in the MRU decreases as the the associativity increases. As a result, the BurstTrace predictor makes predictions earlier and this earlier prediction more than offsets the drop in overall prediction coverage and accuracy because of higher associativity.

Table 2.8 and 2.9 show the prediction accuracy and coverage of each predictor for the multi-threaded benchmarks. For multi-threaded benchmarks, the benefit of using burst history over individual access history is more pronounced: both burst-based predictors (BurstTrace, BurstCount) significantly outperform the corresponding hit-based predictors (RefTrace, RefCount): BurstTrace makes 70% more correct

Application	RefTrace	BurstTrace	RefCount	RefCount+	BurstCount
SPECweb	0.85	0.86	0.78	0.93	0.93
SPECjbb	0.69	0.89	0.65	0.88	0.92
TPC-W	0.62	0.89	0.42	0.82	0.88
barnes	0.68	0.93	0.59	0.75	0.85
FFT	0.91	0.94	0.90	0.99	0.95
lu	0.85	0.90	0.88	0.98	0.98
ocean	0.88	0.95	0.93	0.98	0.97
radix	0.82	0.89	0.84	0.90	0.88
GeoMean	0.78	0.91	0.72	0.90	0.92

Table 2.8: Prediction accuracy of DL1 DBPs (Multi-threaded workloads)

Application	RefTrace	BurstTrace	RefCount	RefCount+	BurstCount
SPECweb	0.30	0.57	0.26	0.37	0.52
SPECjbb	0.15	0.59	0.24	0.28	0.51
TPC-W	0.09	0.41	0.14	0.16	0.37
barnes	0.36	0.81	0.29	0.24	0.50
FFT	0.67	0.73	0.58	0.62	0.60
lu	0.81	0.81	0.81	0.66	0.75
ocean	0.56	0.78	0.32	0.61	0.73
radix	0.86	0.78	0.58	0.58	0.61
GeoMean	0.37	0.67	0.35	0.39	0.56

Table 2.9: Prediction coverage of DL1 DBPs (Multi-threaded workloads)

predictions than RefTrace with much higher accuracy and BurstCount makes 40% more correct predictions than RefCount with higher accuracy. Again, RefCount+ significantly outperforms RefCount with higher coverage and much higher accuracy because of its ability to handle reference count variations better. Of the five predictors, BurstTrace is still the best choice because of its highest coverage, lowest overhead (Table 2.5), and close to the highest accuracy.

Comparing the prediction accuracy and coverage for single-threaded and multi-threaded benchmarks, we observe that the multi-threaded workloads tend

to have lower dead block prediction accuracy and coverage. This could be caused by the cache coherence protocol because the cache invalidations as a result of the coherence protocol can make dead-block predictions harder.

2.5.3 Dead Block Prediction at the L2 Cache

Next, we compare the overhead, prediction accuracy, and prediction coverage of different dead-block predictors for the L2 cache. The dead-block predictors we compare include the RefTrace predictor proposed in [48], the RefCount predictor proposed in [39], and the RefCount+ predictor, which addresses the problems caused by reference count variations. We do not include the two burst-based predictors (BurstTrace, BurstCount) because their prediction accuracy and coverage are slightly lower than those of the corresponding reference-based predictors.

Predictor Overhead

Table 2.10 shows the overhead of each predictor. We assume a 1MB L2 cache with 64-byte blocks. A RefTrace predictor with a history table of 65536 entries causes an overhead of about 5% of the capacity of the L2 cache while the RefCount and RefCount+ predictors cause an overhead of 3.5% and 3% respectively.

Compared to the dead-block predictor overhead of the L1 cache shown in Table 2.5, the dead block predictor overhead at the L2 cache is dominated by the extra bits added to each cache block. This means for large caches, a dead-block predictor that adds fewer bits per block is preferred in terms of area overhead.

Overhead	RefTrace	RefCount	RefCount+
History table entries	65536	2048	2048
History table	16KB	1.25KB	2.5KB
Per-block (bits)	16	17	13
Total overhead	48KB	35.25KB	28.5KB
Relative overhead	4.7%	3.4%	2.8%

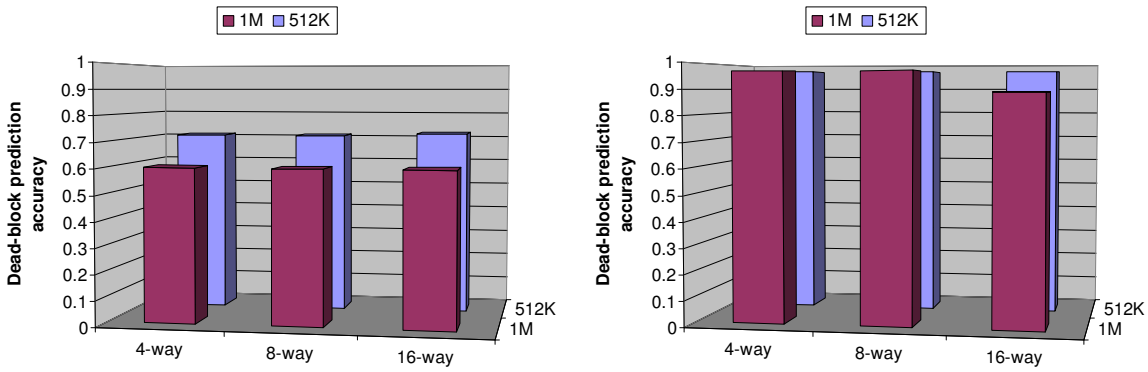
Table 2.10: Overhead of different dead-block predictors for a 1MB L2 cache

Application	RefTrace		RefCount		RefCount+	
	coverage	accuracy	coverage	accuracy	coverage	accuracy
swim	0.61	0.75	0.94	0.99	0.96	1.00
mgrid	0.69	0.80	0.74	0.92	0.85	0.98
applu	0.67	0.80	0.82	0.95	0.88	0.98
gcc	0.23	0.20	0.40	0.26	0.47	0.86
art	0.91	0.97	0.89	0.89	0.92	1.00
mcf	0.51	0.72	0.61	0.91	0.73	0.95
ammp	0.58	0.54	0.52	0.39	0.51	0.72
lucas	0.73	0.68	0.95	0.93	0.98	1.00
parser	0.18	0.21	0.15	0.11	0.23	0.56
perlbmk	0.88	0.69	0.80	0.88	0.85	0.92
gap	0.38	0.46	0.98	0.99	0.98	1.00
sphinx	0.28	0.54	0.40	0.30	0.37	0.79
corner_turn	0.41	0.40	0.55	0.38	0.40	0.85
stream	0.78	0.76	0.98	1.00	0.99	1.00
GeoMean	0.51	0.55	0.63	0.60	0.66	0.89

Table 2.11: Coverage and accuracy of L2 DBPs (Single-threaded workloads)

Prediction Accuracy and Coverage

Table 2.11 shows the coverage and accuracy of the three dead-block predictors for single-threaded benchmarks. At the L2 cache, the two counting-based predictors (RefCount, RefCount+) both outperform the trace-based predictor (RefTrace), corroborating the findings in [39] that counting-based predictors work better than trace-based predictors at the L2 cache. Of the two counting-based predictors (Re-



(a) RefCount

(b) RefCount+

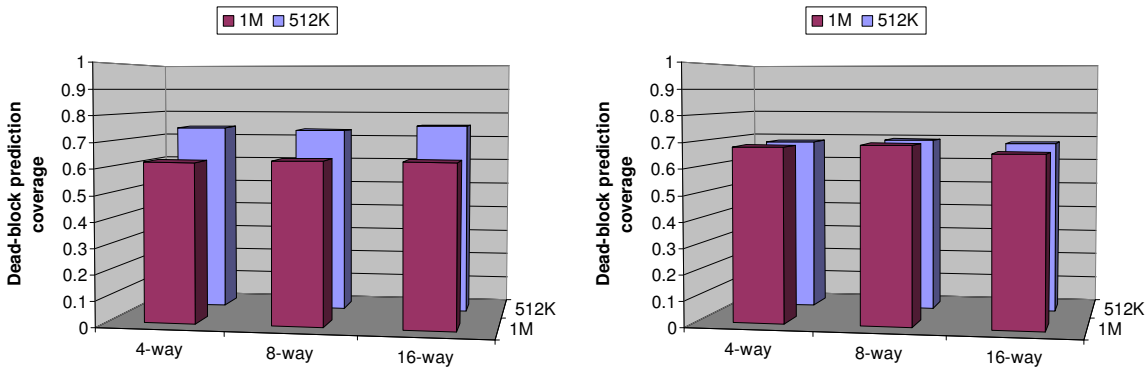
Figure 2.15: Dead-block prediction accuracy of RefCount and RefCount+ with different cache configs

fCount, RefCount+), RefCount+ has significantly higher accuracy (89% vs. 60%) and slightly higher coverage because of its ability to handle reference count variations better, as discussed earlier. And according to Table 2.10, RefCount+ also incurs the smallest overhead among the three, making it the most appealing choice for the L2 cache.

Figures 2.12 and 2.13 show the dead-block prediction accuracy and coverage of the RefCount predictor and the RefCount+ predictor with different cache sizes and associativities. For all configurations, RefCount+ matches RefCount in prediction coverage but has significantly higher prediction accuracy.

For RefCount, the prediction accuracy and coverage remain unchanged as the associativity increases. However, both the prediction accuracy and coverage drop more than 10% when the L2 cache size increases from 512KB to 1MB.

For RefCount+, the prediction accuracy and coverage are mostly the same across all cache sizes and associativites, except that there is drop in prediction accuracy of about 8% and a slight drop in prediction coverage for the 16-way, 1M



(a) RefCount

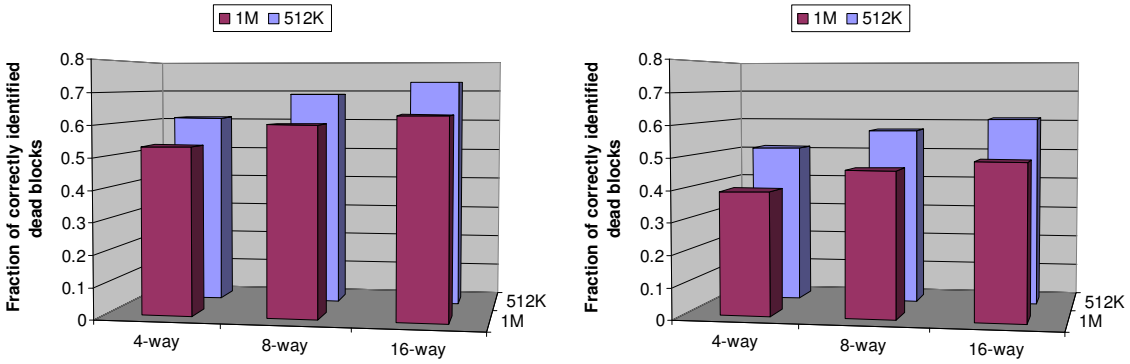
(b) RefCount+

Figure 2.16: Dead-block prediction coverage of RefCount and RefCount+ with different cache configs

L2 cache configuration.

Figure 2.17 shows the fraction of the dead blocks that has been correctly identified by RefCount+ at any moment. Figure 2.17(a) shows the average fraction of the dead blocks that have been correctly identified as dead at any given cycle whereas Figure 2.17(b) shows the average fraction of all the blocks that have been correctly identified as dead. As the associativity of the cache increases, the fraction of the dead blocks that have been correctly identified as dead at any given moment increases because dead-block prediction are made earlier. The fraction of dead blocks that has been correctly identified as dead is higher with a 512KB L2 cache because RefCount+ has higher prediction accuracy and coverage with a 512KB L2 cache, as shown in Figures 2.15 and 2.16.

Table 2.12 shows the prediction coverage and accuracy of the three dead-block predictors for multi-threaded benchmarks. One thing to notice here is that for the two counting-based predictors (RefCount, RefCount+), the prediction coverage and accuracy are much lower compared to those for the single-threaded workloads.



(a) Out of all dead blocks

(b) Out of all blocks in the cache

Figure 2.17: Fraction of correctly identified dead blocks by RefCount+ at any given time

This effect is caused by cache invalidations due to the coherence protocol, which makes dead-block predictions harder. Although RefCount+ still has the highest accuracy and significantly outperforms RefCount, its coverage is only about 27%. Another interesting result is that RefTrace has the highest coverage of the three predictors. This could also have been caused by the cache coherence protocol: the L2 caches in a CMP see more accesses (for example, write misses and upgrade requests) from the L1 which would otherwise be filtered by the L1 cache in a single processor. While RefTrace has the highest prediction coverage, its prediction accuracy is relatively low (68%). These results suggest that these three dead-block predictors may not be benefitial for multiprocessors.

A comparison of the results for the L1 cache and the L2 cache also indicate that dead block prediction at the L1 data cache has higher accuracy and coverage, for two reasons. First, the filtering effect of the L1 cache makes dead block prediction at the L2 harder. Second, the higher associativity of the L2 cache also makes dead-block prediction harder. This is true for both single-threaded and multi-threaded

Application	RefTrace		RefCount		RefCount+	
	coverage	accuracy	coverage	accuracy	coverage	accuracy
SPECweb	0.59	0.78	0.35	0.44	0.22	0.86
SPECjbb	0.46	0.58	0.32	0.40	0.23	0.67
TPC-W	0.45	0.74	0.31	0.38	0.27	0.89
barnes	0.32	0.69	0.20	0.09	0.19	0.81
FFT	0.38	0.58	0.10	0.29	0.37	0.57
lu	0.62	0.69	0.52	0.10	0.37	0.92
ocean	0.50	0.83	0.33	0.55	0.34	0.92
radix	0.42	0.59	0.11	0.38	0.23	0.54
GeoMean	0.46	0.68	0.25	0.28	0.27	0.76

Table 2.12: Coverage and accuracy of L2 DBPs (Multi-threaded workloads)

benchmarks.

2.6 Summary

In this chapter, we investigated how to identify dead blocks early through better dead-block prediction schemes. Good dead-block predictors are a first step towards better cache efficiency.

The three metrics for dead-block prediction are prediction accuracy, coverage, and timeliness. However, it is not possible to achieve high accuracy, coverage, and timeliness at the same time. Dead-block predictors proposed by prior work have been confined to making predictions immediately after a block is referenced. In this chapter, we quantified the tradeoff between prediction timeliness, accuracy, and coverage and showed that delaying prediction until a block just moves out of the MRU position gives the best tradeoff among the three metrics.

Accesses to L1 and L2 caches have different characteristics: accesses to L1 caches are bursty with abundant intra-block locality and can be easily affected by

data and control flow dependences whereas accesses to the L2 cache are filtered by the L1 cache, have little intra-block locality, and are less affected by data and control flow dependences. Because of these differences, a dead-block predictor should maintain different state in each block to make better dead-block predictions at the L1 and L2 caches.

For the L1 cache, we propose the concept of cache bursts. A cache burst combines the contiguous group of references a block receives while in the MRU position of its cache set into one entity and can thus hide the irregularity of individual references caused by data and control dependences. Dead-block predictors at the L1 cache should maintain state about cache bursts, not individual references, to make predictions because cache bursts are more predictable than individual references. We propose a burst-counting predictor and a burst-trace predictor that update the burst count/trace only when a block moves into the MRU position and make predictions only when a block moves out of the MRU position. Compared to reference-based predictors, the new burst-based predictors can correctly identify more dead blocks while making fewer predictions. The best burst-based predictor, BurstTrace, can identify 96% of the dead blocks in a 64KB, 2-way set-associative L1 D-cache with a 96% accuracy. Besides the better prediction accuracy and coverage, burst-based predictors also cause lower power overhead because they update the burst count/trace and access the history table less frequently.

For the L2 cache, a dead-block predictor should maintain state about reference counts to make predictions because of the filtering effect by the L1 cache. To cope with reference count variations, we optimize an existing counting-based predictor by maintaining more up-to-date history information to increase prediction accuracy and filtering out sporadic smaller reference counts to increase prediction

coverage. The improved predictor can identify 66% of the dead blocks in a 1MB, 16-way set-associative L2 cache with a 89% accuracy. For multiprocessors, however, none of the existing dead-block predictors work well and more future research is needed to better cope with the invalidations caused by the coherence protocol.

Chapter 3

Improving Cache Efficiency by Eliminating Dead Blocks Early

In this chapter, we investigate how the dead-block predictors discussed in Chapter 2 can be used to improve the efficiency of L1 data caches and L2 caches. There are several distinct ways to use dead-block prediction to improve cache efficiency. A conservative approach, including replacement optimization and cache bypassing, only evicts dead blocks early to give other blocks more opportunities to get reused. A more aggressive approach prefetches new blocks into dead blocks to reduce future demand misses.

Several proposals from prior work exist on using dead-block prediction to improve cache efficiency. However, these approaches consider only one particular dead-block prediction scheme and use it at only one cache level for one optimization. For example, Lai et al. used RefTrace and Hu et al. used a time-based dead-block predictor to trigger prefetches into dead blocks at the L1 data cache [29, 48], while Kharbutli et al. used RefCount to improve the LRU replacement algorithm at the

L2 cache [39].

We investigate how to use dead-block prediction for evicting dead blocks early and prefetching into dead blocks at both the L1 and L2 caches. We study two optimizations that evict dead blocks early. The first optimization, replacement optimization, always places a missing block into the cache but chooses a block predicted dead over the LRU block for replacement on a cache miss. The second optimization, cache bypassing, does not place a missing block into the cache if the block is not likely to be reused after being cached. We also study prefetching into dead blocks at different cache levels but control the aggressiveness of the prefetching by considering the tradeoffs between prefetching pollution and coverage, which is affected by the cache capacity and the miss penalty of the cache.

3.1 Evicting Dead Blocks Early

LRU and pseudo-LRU replacement algorithms are widely used in microprocessor caches. With LRU or pseudo-LRU replacement, blocks with poor locality can stay in the cache too long and cause blocks with good locality to be replaced. To address this problem, a dead block should be evicted from the cache before it reaches the LRU position. However, if these blocks do not receive additional references, evicting dead blocks early does not improve performance.

3.1.1 Replacement Optimization

In replacement optimization, dead-block prediction is used along with the LRU replacement algorithm to choose the right block to evict on a cache miss. Instead of always choosing the LRU block to evict, if a block in a non-LRU position of the cache set where the miss occurred has already been predicted dead, the dead block

is chosen for replacement. If no dead blocks are found in the set, the LRU block is replaced. Choosing a dead block not in the LRU position for replacement gives other blocks that are located lower on the LRU stack more time to be referenced again. If a program references many blocks with poor temporal/spatial locality, this optimization can identify these blocks and reduce the time these blocks are kept in the cache. Of course, if the blocks that are located lower on the LRU stack than the dead block will not be referenced in the near future, replacing the dead block early does not bring any performance gain.

Since replacement optimization tries to replace dead blocks before they reach the LRU position, only dead-block prediction schemes that identify dead blocks early enough can be used for this optimization. A dead-block prediction scheme that predicts blocks dead when they reach the LRU position will not be useful for this optimization.

3.1.2 Bypassing

On a cache miss, most cache replacement algorithms (including the oracular optimal replacement algorithm) always insert the block causing the miss into the cache. However, if the missing block will not get additional references, inserting it into the cache does not bring any benefit and can even displace blocks which will be referenced again.

Programs that exhibit poor locality or have a working set larger than the capacity of the cache can sweep the cache with little or no reuse of the blocks. Table 3.1 shows that in some benchmarks like *art*, *perlbmk*, and *ammp*, a large fraction of the cache blocks are never reused. On average, 32% of the blocks brought into the L1 cache and 40% of the blocks brought into the L2 cache for the single-

Application	DL1 zero-reuse blocks	L2 zero-reuse blocks
swim	0.00	0.11
mgrid	0.03	0.31
applu	0.14	0.58
gcc	0.47	0.27
art	0.17	0.91
mcf	0.49	0.51
ammp	0.41	0.77
lucas	0.00	0.53
parser	0.32	0.15
perlbmk	0.48	0.81
gap	0.01	0.01
sphinx	0.27	0.53
corner_turn	0.76	0.01
stream	0.40	0.58
vpenta	0.78	0.00
GeoMean	0.32	0.40

Table 3.1: Fraction of zero-reuse blocks out of all the blocks brought into the cache

threaded benchmarks listed in Table 3.1 are zero-reuse blocks. Writing these blocks into the cache causes three problems. First, it pollutes the cache by evicting useful blocks. Second, it can displace dirty blocks in the cache and generate unnecessary writebacks, which consumes bandwidth and increases the pressure on caches at lower levels or main memory. Last, it increases power consumption.

A more aggressive form of early dead-block eviction, cache bypassing, targets these zero-reuse blocks. Cache bypassing uses dead-block prediction to identify these zero-reuse blocks: on a cache miss, a prediction about whether the block causing the miss is made. If the missing block is predicted dead, it is not be written into the cache. Cache bypassing can be considered as a special form of replacement optimization because the missing block, along with the blocks already in the same set, is considered for eviction. Similar to the cache replacement optimization, bypassing

gives other blocks in the set more time to be referenced again.

For a dead block prediction scheme to be applicable to cache bypassing, it must be able to make dead-block predictions immediately after any cache access. For this reason, burst-based predictors can not be used to implement bypassing because they make predictions only after a block moves out the MRU position.

3.1.3 Results

To evaluate the effectiveness of using dead-block prediction for cache replacement optimization and bypassing, we simulate the single-threaded benchmarks using the *sim-alpha* simulator as described in section 2.5.

Figure 3.1 shows the speedup of several schemes that use dead-block prediction to evict dead blocks early. The "RefCount:Replace" scheme is the same as the counting-based replacement algorithm described in [39]. It uses the RefCount dead-block predictor to improve the LRU replacement algorithm: on a cache miss, it first tries to find a block that is predicted dead; if no such block exists, it chooses the LRU block for eviction. The "RefCount+:Replace" scheme is similar except that it uses the RefCount dead-block predictor, which has higher dead-block prediction accuracy and coverage. The "RefCount+:Bypass" scheme uses RefCount+ just for bypassing: if a missing block is predicted dead, it will not be written into the cache. The "RefCount+:Bypass+Replace" scheme uses RefCount+ for both bypassing and replacement: on a cache miss, it first tries to find a block that is predicted dead for replacement; if no such block exists, it bypasses the missing block if it is predicted dead; otherwise, the LRU block is chosen for eviction.

Figure 3.1 indicates that the four schemes achieve similar performance improvements on most of the benchmarks and the average speedup is approximately

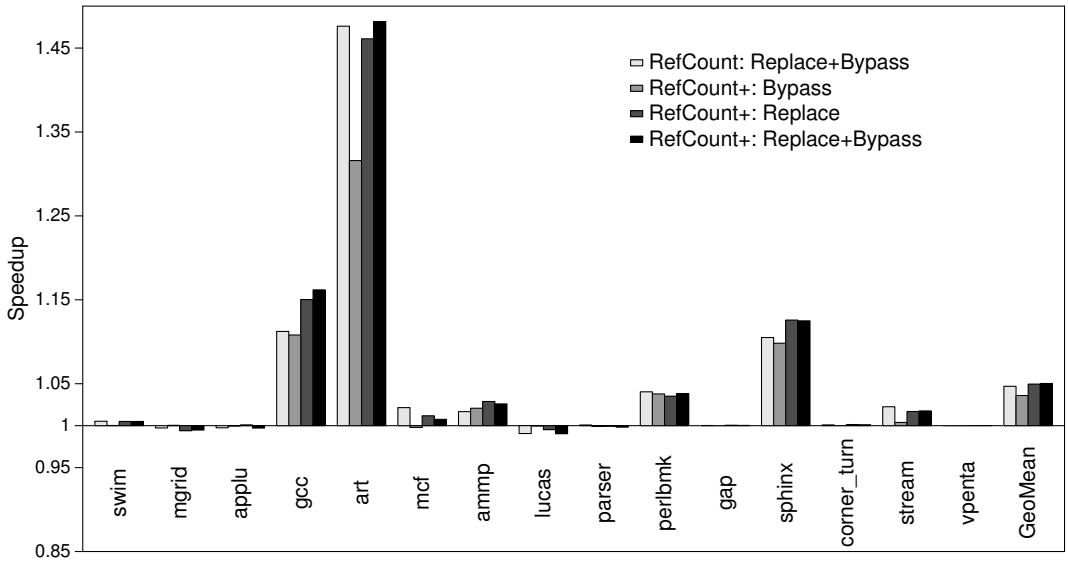


Figure 3.1: Speedups of using dead-block prediction for replacement/bypassing with a 1MB L2 cache

5%. The four benchmarks (*gcc*, *art*, *perlbmk*, and *sphinx*) that benefit most from the replacement optimization and bypassing all show significant performance improvements when simulated with a larger cache without the replacement optimization or bypassing, suggesting that these two optimizations are mainly reducing the number of capacity misses. Although several other benchmarks like *mcf*, *swim*, *mgrid* also suffer a lot of capacity misses, they do not benefit much from replacement optimization or bypassing because their working set is much larger.

Table 3.2 shows the corresponding improvement in L2 efficiency by using the RefCount+ predictor for replacement optimization and bypassing. For some benchmarks like *stream*, although there is a big increase in L2 efficiency, the performance improvement is much smaller.

The results also indicate that the benefits of using dead block prediction

Application	Baseline L2 efficiency	L2 efficiency by evicting dead blocks early
swim	0.06	0.07
mgrid	0.18	0.22
applu	0.03	0.05
gcc	0.34	0.56
art	0.12	0.78
mcf	0.05	0.16
ammp	0.05	0.10
lucas	0.01	0.01
parser	0.32	0.33
perlbmk	0.17	0.21
gap	0.07	0.07
sphinx	0.34	0.58
corner_turn	0.04	0.04
stream	0.03	0.53
vpenta	0.80	0.80
Mean	0.17	0.30

Table 3.2: Improvement in L2 cache efficiency by evicting dead blocks early through replacement optimization & bypassing

for bypassing and replacement are mostly overlapped: if a program benefits from bypassing, it also benefits similarly from the replacement optimization. And doing bypassing and replacement optimization at the same time does not bring much additional performance gain.

However, applying dead-block prediction for replacement optimization and bypassing at the L1 cache show little performance improvement, for several reasons.

First, to reduce the number of cache misses, evicting dead blocks early through replacement optimization and bypassing relies on the blocks that are given more time to be referenced to get additional references. Otherwise, just evicting the dead blocks early does not benefit performance. With the low associativity of the L1 cache, the probability of the other blocks in the same set being accessed in the near future is lower.

Second, compared to L2 misses, L1 misses have much shorter penalty. So the difference in performance between a hit and miss in the L1 cache is much smaller than the difference in performance between a hit and miss in the L2 cache. Using dead-block prediction at the L1 cache mainly reduces the number of L1 misses that would hit in the L2 cache. Therefore, the performance improvement is much smaller than the corresponding performance improvement at the L2 cache. Furthermore, an out-of-order superscalar processor can tolerate L1 misses that hit in the L2 cache much better than L2 misses.

Third, using dead-block prediction for replacement and bypassing is most effective when the working set of the program is larger than the capacity of the cache but does not exceed the cache capacity too much. For example, if a program has a working set larger than the capacity of the cache, most of the blocks will not be reused, even if the program can show good temporal locality when running

with a larger cache. Bypassing some blocks in this case can cause part of the working set to get reused. However, if the working set is much larger than the cache capacity, bypassing is unlikely to bring noticeable performance improvement because the program is still severely limited by the large number of capacity misses. Because most of the benchmarks in Figure 3.1 have a working set much larger than the capacity of the L1 D-cache, the performance benefit of replacement optimization and bypassing at the L1 cache is small.

On the other hand, even if bypassing does not bring any noticeable performance improvement, it can still save a large portion of the dynamic power consumption of the cache if a program has a lot of zero-reuse blocks.

3.1.4 Discussion

Coupled with the cache burst concept discussed in Chapter 2, more effective cache-bypassing schemes can be conceived. Like prior work [33, 34, 96] on cache bypassing, this dissertation has only considered bypassing blocks that will be accessed only once when brought into the cache. This strategy limits the success of cache bypassing at the L1 cache because many blocks are accessed more than once in a burst in the L1 cache. One example is media streaming applications. In such applications, usually several words in a block are referenced in a short interval in the L1 cache, after which the block is not referenced again. For these applications, cache bypassing can be performed more aggressively. Instead of just bypassing zero-reuse blocks, blocks that die after exactly one burst can also be bypassed, even though the burst may consist of several references. This new bypassing strategy requires a small buffer to store bypassed blocks temporarily. The benefit is reduced pollution in the cache and potentially fewer misses. By bypassing the one-burst blocks, the other blocks

will have shorter reuse distance and potentially higher hit rate. Another policy would be to write a one-burst block into the cache only if there was a dead block there; otherwise the one-burst block is placed into the bypass buffer. We leave such bypassing strategies for future work.

3.2 Prefetching into Dead Blocks

One limitation of early dead-block eviction is that successful bypassing or early replacement of dead blocks does not always improve performance. It can also leave dead blocks in the cache because it only evicts dead blocks on cache misses. For programs that do not benefit from early dead-block eviction, more aggressive techniques must be used to reduce the number of cache misses, which can be achieved by prefetching into dead blocks.

3.2.1 Synergy between Dead-block Prediction and Prefetching

While prefetching can be performed without dead-block prediction, using dead-block prediction to trigger prefetches has two benefits.

First, dead blocks provide ideal space to store prefetched blocks without causing pollution. When applied to different levels of caches, this property can be utilized differently to find the right tradeoff between prefetching coverage and pollution. If increased prefetching coverage is more important, dead-block prediction can trigger additional prefetches. If reducing pollution is more important, dead-block prediction can trigger prefetches only when there is space to accommodate the prefetched blocks, i.e., after some blocks in the cache become dead.

Second, the long dead time gives sufficient slack for the prefetched blocks to arrive at the cache before they are referenced by the program. There is no point in

prefetching new blocks into a cache set if all the current blocks in set are still live. The time when a block dies is the earliest time that a new block can be brought into the same set without causing pollution. Because the average dead time is long, if prefetches are initiated not long after blocks die, the prefetched blocks are likely to arrive at the cache in time without causing any pollution.

One issue ignored by prior work that uses dead-block prediction for prefetching is how to track prefetched blocks so that the dead-block predictor can predict when these blocks become dead. The prefetch engine can bring many blocks into the cache and these prefetched blocks are not associated with any instruction in a program. Since all the dead-block predictors we study in this work use the PC to make predictions, prefetched blocks will not be predicted dead, preventing further prefetches from being triggered. To address this problem, an extra bit, *pc_valid*, is added to each block to differentiate prefetched blocks from blocks that are caused by demand misses. For prefetched blocks, the *pc_valid* bit is initially set to zero. When a prefetched block is accessed for the first time, its *pc_valid* bit is set to one and the PC of the current instruction is used to update the hashed PC stored along with the block.

3.2.2 Baseline Prefetch Engine

We use an existing prefetching scheme, tag correlating prefetching (TCP) [30] as the baseline prefetch engine. TCP is a correlating prefetcher [12] that was proposed to reduce the penalty of L1 misses by placing prefetched data in the L2 cache to avoid polluting the L1 cache. With dead-block prediction at the L1 cache, prefetched data can be directly placed into the L1 cache. Figure 3.2 shows how TCP works. Each set maintains the two most recent tags that caused misses to the set. On a miss, a

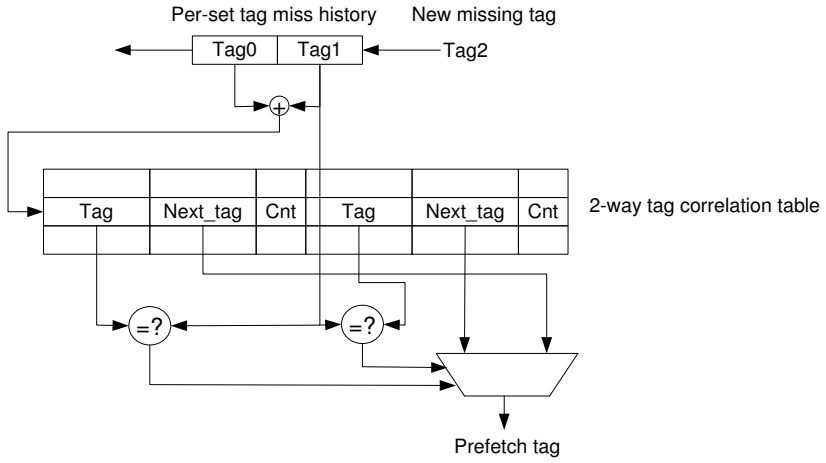


Figure 3.2: Baseline Tag Correlating Prefetch Engine

hash of the two tags in the miss history of the accessed set is used as index into the correlation table. If a match is found, the predicted tag is used with the index of the set to form a prefetch address. The correlation table is updated on every cache miss.

Like all correlation-based prefetchers, TCP can learn arbitrary repetitive access patterns. TCP also exploits the property that the same sequence of tags are often accessed in different sets, which is called constructive aliasing of tag correlation. Constructive aliasing enables TCP to learn access patterns more quickly because it takes shorter time to train the correlation predictor for a given sequence pattern when the pattern repeats in different sets. By just recording tag correlation, instead of full address correlation, TCP requires a much smaller correlation table because the same tag correlation record represents multiple full address correlations.

The size of the correlation table is as follows: when applied to the L1 cache, the correlation table has 1024 sets; when applied to the L2 cache, the correlation table has 8192 sets. Both the L1 and L2 correlation tables are 2-way set-associative.

Each entry in the table has two tags, one tag for matching and the other one is tag of the block to be prefetched. There is also a two-bit saturating counter in each entry to measure the stability of the correlation of the tags. Altogether, each entry takes about 5 bytes. Hence, the L1 correlation table is approximately 10KB and the L2 correlation table is approximately 80KB.

3.2.3 Reducing Pollution in L1 Prefetching

When prefetching into the L1 cache, extra care must be taken to avoid pollution because of the the small capacity of the L1 cache. Pollution can come from two sources: incorrectly predicted addresses that will not be referenced by the program and correct prefetches that arrive too early and evict live blocks in the cache.

One way to avoid pollution is to use a prefetch buffer parallel to the L1 data cache [35]. Such a buffer, however, may increase the critical path of L1 cache accesses and needs to be searched every time the L1 cache is accessed. A prefetch buffer also requires extra storage overhead. If the number of entries in the prefetch buffer is small, blocks prefetched earlier can be overwritten by blocks prefetched later, limiting the effectiveness of prefetching.

Another way to reduce pollution is to place the prefetched block into the LRU position of the cache set [54]. This policy is most effective when the associativity is high. Furthermore, placing a prefetched block into the LRU position also increases the probability that the block may be evicted before it is actually referenced by the program.

We make use of the dead-block information to avoid pollution by placing prefetched blocks into the space of the dead blocks. Instead of triggering prefetches on misses, prefetches are triggered only when dead blocks are detected

and prefetched blocks are stored in the space of the dead blocks. The overhead of this approach is the addition of a dead-block predictor, which is small as shown in table 2.5 and its operation is not on the critical path of the L1 cache access.

Using dead-block prediction to trigger prefetches at the L1 data cache was first proposed by Lai et al. in a scheme called dead block correlating prefetching (DBCP) [48]. DBCP uses RefTrace for dead-block prediction and full block address correlation for prefetch address prediction. The dead-block prediction and prefetch address prediction are closely coupled: the same predictor structure predicts both if a block has died and the address of the next block to prefetch. Using full block address correlation requires a large table: the DBCP scheme evaluated in [48] uses a 2MB history table for a 32KB directly-mapped L1 data cache, which is not practical to implement. Coupling dead-block prediction closely with prefetch address prediction makes it impossible to optimize the dead-block prediction and address prediction independently of each other. In DBCP, a prefetch address prediction is always made at the same time with a dead-block prediction, using the same history information. However, for best results, the dead-block predicitor and the prefetch address predictor may need to keep track of different history information. Furthermore, they do not need to make predictions at the same time.

We build upon the DBCP work and address its limitation by decoupling dead-block prediction from prefetch address prediction. A decoupled design makes it easy to optimize each component independently. By using tag correlation instead of full block address correlation, we reduce the size of the correlation table from 2MB to 10KB. This reduction in table size comes from two sources. First, tag correlation exploits the property that the same tag tends to appear in multiple sets of the cache in many applications. Therefore, the same entry of tag correlation can represent

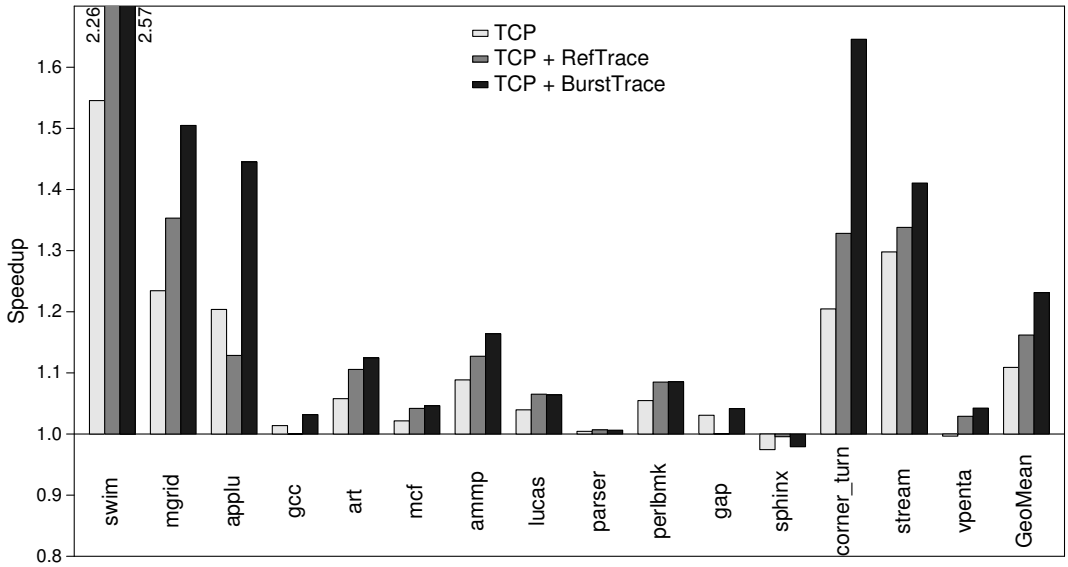


Figure 3.3: Speedups of L1 prefetching schemes with a 64KB L1 cache

multiple entries of full block address correlation for different sets. Compared to full-address correlation, this more storage-efficient representation of address correlation reduces the number of entries in the correlation table. Second, each entry of tag correlation requires fewer bits because it only records the tag part of a block address, instead of the full block address. For dead-block prediction, besides the RefTrace scheme used in DBCP, we also evaluate the BurstTrace scheme, which has been shown to have higher dead-block prediction accuracy and coverage than RefTrace.

Figure 3.3 compares the speedups of three L1 prefetching schemes. The baseline TCP prefetches on L1 misses. To avoid pollution, it places prefetched blocks into the LRU position [54]. The second scheme uses the RefTrace dead-block predictor with the baseline prefetch engine. It prefetches when dead blocks are found and places prefetched blocks into the space of dead blocks. This scheme resembles the DBCP scheme because it uses the same dead-block predictor but differs from

Application	Baseline DL1 efficiency	DL1 efficiency by prefetching into dead blocks
swim	0.02	0.36
mgrid	0.08	0.24
applu	0.07	0.23
gcc	0.05	0.10
art	0.01	0.10
mcf	0.04	0.07
ammp	0.08	0.14
lucas	0.01	0.06
parser	0.33	0.33
perlbmk	0.40	0.40
gap	0.07	0.12
sphinx	0.09	0.10
corner_turn	0.02	0.12
stream	0.01	0.21
vpenta	0.01	0.01
Mean	0.08	0.17

Table 3.3: Improvement in DL1 cache efficiency by prefetching into dead blocks

DBCP in the prefetch engine. The third scheme is similar to the second one except that it uses the BurstTrace dead-block predictor, which works best at the L1 cache. Figure 3.3 shows that using dead-block prediction improves prefetching performance for almost all the applications. It also shows that BurstTrace outperforms RefTrace when used with the baseline prefetching engine because of its higher prediction accuracy and coverage. On average, the baseline prefetching engine improves IPC by 11%, adding RefTrace improves IPC by 16%, and adding BurstTrace improves IPC by 23%. The 7% performance improvment of BurstTrace over RefTrace also comes with power reductions in dead-block prediction because of the lazy prediction strategy of BurstTrace.

Table 3.3 shows the improvements in DL1 efficiency using the BurstTrace predictor with the TCP prefetch engine. On average, the DL1 efficiency doubles

from 8% to 17%.

3.2.4 Increasing Coverage in L2 Prefetching

Dead block prediction can also be used to trigger prefetches into L2 caches, which has not been studied by prior work. Applying dead-block prediction to L2 prefetching differs from L1 prefetching in several ways. First, the L2 cache is more tolerant of pollution but L2 misses are much more expensive. Therefore L2 prefetching should be more aggressive. Second, dead-block prediction at the L2 cache has much lower coverage (66%) than at the L1 (96%). This means one third of the dead blocks are not identified by dead-block prediction and triggering prefetches only when dead blocks are identified will miss many opportunities to prefetch. Therefore, besides issuing prefetches when dead blocks are identified in the L2 cache, additional prefetches are issued when the L2 cache misses, to cover the otherwise missed opportunities of those dead blocks that are not identified by dead-block prediction.

Figure 3.4 shows the speedup of two L2 prefetching schemes: the baseline TCP, which prefetches on L2 misses, and the baseline TCP augmented with RefCount+, which prefetches on both demand misses and dead block detections. The figure shows using RefCount to trigger more prefetches improves IPC by 23% compared to the IPC improvement of 10% by the baseline TCP.

Table 3.4 shows the improvements in L2 efficiency using the BurstCount predictor with the TCP prefetch engine. Compared to using dead-block prediction for replacement optimization and bypassing, prefetching into dead blocks brings slightly smaller improvements in terms of L2 efficiency but larger improvements in performance.

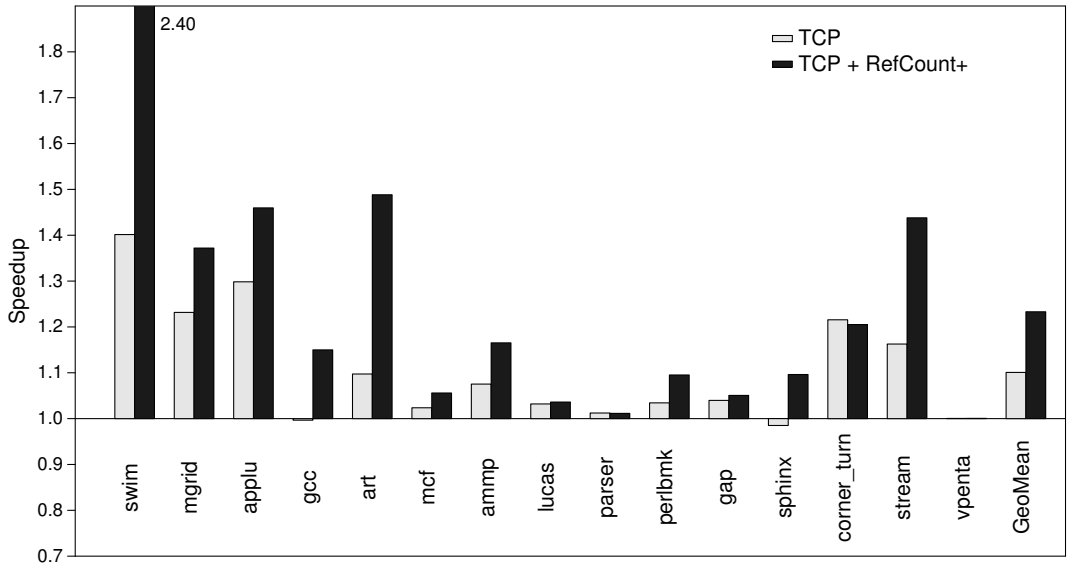


Figure 3.4: Speedup of L2 prefetching with a 1MB L2 cache

Application	Baseline L2 efficiency	L2 efficiency by prefetching into dead blocks
swim	0.06	0.10
mgrid	0.18	0.23
applu	0.03	0.07
gcc	0.34	0.55
art	0.12	0.69
mcf	0.05	0.14
ammp	0.05	0.08
lucas	0.01	0.04
parser	0.32	0.33
perlbmk	0.17	0.21
gap	0.07	0.09
sphinx	0.34	0.52
corner_turn	0.04	0.05
stream	0.03	0.16
vpenta	0.80	0.80
Mean	0.17	0.27

Table 3.4: Improvement in L2 cache efficiency by prefetching into dead blocks

3.3 Related Work

Prior work has used dead-block prediction for optimizing the cache efficiency and other aspects of the cache, such as power consumption and coherence protocols.

Prefetching: Lai et al. [48] and Hu et al. [29] used dead-block prediction to trigger prefetches into dead blocks in the L1 data cache. They found triggering prefetches on dead-block predictions improves the timeliness of prefetching compared to triggering prefetches on cache misses. In both work, dead-block prediction is tightly coupled with prefetch address prediction and can only be used for prefetching. Ferdman and Falsafi later extended the work in [48] to store correlation patterns off-chip and stream them on-chip as needed [17], which makes it possible to perform correlation-prefetching with large correlation tables.

Replacement: Kharbutli and Solihin [40] used dead-block prediction to improve the LRU algorithm by replacing dead blocks first, and also for bypassing the cache. Other approaches optimize LRU replacement without dead-block prediction: Wong and Baer modified the LRU algorithm by replacing blocks with no temporal locality first [100], Kampe et al. proposed an Self-Correcting LRU algorithm [36] to correct LRU replacement mistakes, whereas Qureshi et al. proposed to adaptively place missing blocks into the LRU instead of the MRU position when the working set is larger than the capacity of the cache [70]. Besides these hardware approaches, Wang et al. proposed to use compiler analysis to identify cache blocks that will not be reused and pass this information to the hardware to improve cache replacement decisions [98].

Bypassing: Prior work has also used bypassing [20, 32, 33, 78, 96] to improve cache efficiency. Tyson et al. proposed bypassing based on the hit rate of the missing load/store instruction [96]. Johnson et al. proposed bypassing based on the reference

frequency of the data being referenced [33] but put bypassed blocks in a separate buffer parallel to the cache. Jalminger and Stenström proposed bypassing based on the reuse distance of the missing block [32]. González et al. proposed to bypass L1 data cache blocks with low temporal locality [20]. McFarling applied bypassing to the instruction cache [61].

Power reduction: Dead-block prediction has also been used to reduce leakage by dynamically turning off dead blocks. Kaxiras et al. used dead-block prediction to turn off blocks in the L1 D-cache [37]. Abella et al. proposed to turn off blocks in the L2 cache dynamically [4]. Both schemes predict how many cycles have to pass before a block can be turned off without affecting performance. Dead-block prediction can also be used in drowsy caches [8, 18], to decide which blocks should switch to the drowsy state.

Coherence protocol optimization: Cache coherence protocols can also benefit from dead-block prediction. Lebeck and Wood proposed dynamic self-invalidation [49] to reduce the overhead of the cache coherence protocol by invalidating some of the shared cache blocks early. The shared cache blocks to be invalidated early are identified by the cache directory and conveyed to the cache controller. Lai and Falsafi later proposed a last-touch predictor [47] that uses PC traces to predict when shared cache blocks should be invalidated. Somogyi et al. studied using PC-traces to identify last stores to cache blocks [90].

Improving Byte-level Efficiency: Besides the block-level inefficiency, caches also suffer byte-level inefficiency. Byte-level inefficiency can be addressed by sub-blocking [55] or storing only the parts of a block that will be referenced by the program [69, 71]. These techniques aim to improve the byte-level efficiency of the cache and are orthogonal to the techniques we have discussed to improve the

block-level efficiency.

3.4 Summary

The efficacy of the cache is determined by the amount of useful data it stores, not the capacity of the cache. In this chapter, we use dead-block predictors discussed in Chapter 2 to increase the efficacy of the cache through replacement optimization, cache bypassing, and prefetching.

All three approaches try to eliminate dead blocks early but differ in when and how dead blocks are eliminated. Both replacement optimization and bypassing eliminate dead blocks only on demand misses; replacement optimization evicts dead blocks already in the cache while bypassing evicts dead blocks causing the misses. Both can miss opportunities by leaving dead blocks in the cache. Prefetching into dead blocks aims to eliminate dead blocks whenever they are identified. As a result, prefetching is able to reduce more cache misses and achieve greater performance improvement. On average, replacement optimization or bypassing improves performance by 5% while prefetching into dead blocks brings a 12% performance improvement over the baseline prefetching scheme for the L1 cache and a 13% performance improvement over the baseline prefetching scheme for the L2 cache.

These results indicate that it is possible to increase cache efficiency by storing useful data in the space of dead blocks. At the same time, even after these optimizations, the average cache efficiency is still low (17% for the L1 and 27% for the L2). The remaining sources of inefficiency include: dead blocks identified too late, live blocks which are incorrectly identified as dead blocks, dead blocks not identified by the dead-block predictor, the time spent waiting for correctly prefetched blocks to arrive, and useless prefetches.

However, because of the intrinsic tradeoff in dead-block prediction accuracy, coverage, and timeliness, it may be hard to achieve high average cache efficiencies in the range of 60% or higher. On one hand, accuracy drops as timeliness improves and vice versa. Furthermore, it takes time to replace a dead block from the time the block is identified dead to the time the new block arrives. This means some portion of the cache will inevitably be occupied by dead blocks. On the other hand, cache efficiency by itself is not the ultimate goal to optimize for. It is possible to have caches with relatively low efficiency but also low miss rate. For example, in a prefetching scheme where prefetched blocks always arrive just before they are needed by the program, a large portion of the cache can be occupied by dead blocks and the miss rate can still be very low.

Chapter 4

Instruction Cache Design for EDGE Architectures

The last two chapters discussed techniques to improve the efficiency of the L1 data cache and the L2 cache. In the next two chapters, we study how to improve the efficiency of the L1 instruction cache. In particular, we study the instruction cache for systems that support EDGE ISAs.

The EDGE (Explicit Data Graph Execution) architecture addresses the scalability problem faced by superscalar processors. As the delay of global on-chip wires increases relative to the transistor switching speed, the large structures such as the register file, the instruction scheduler, and the bypass network used by traditional superscalar processors do not scale well and it has been increasingly hard to achieve improved performance on single-threaded applications. EDGE architectures strive to sustain continued performance improvements of single-threaded applications by exploiting concurrency and tolerating wire delay through both the hardware and the software.

EDGE architectures employ block-atomic execution and dataflow-like direct operand communication among instructions within the same block. EDGE architectures are implemented in distributed microarchitectures to exploit concurrency and tolerate communication latency. This chapter describes my role in the design and implementation of the distributed instruction cache in the TRIPS prototype [81, 82], the first instantiation of an EDGE architecture. It also describes the instruction cache in TFlex [43], a next-generation EDGE microarchitecture that shares the same ISA as TRIPS but contains additional capabilities provided by the microarchitecture. The next chapter discusses the efficiency problem of the current instruction cache design in TFlex and proposes techniques to increase the I-cache efficiency.

4.1 EDGE Architectures

The inherent high cost and power inefficiency when scaling up large structures such as the register file, the instruction scheduler, and the bypass network in superscalar processors have forced mainstream microprocessors out of primarily relying on CMOS scaling and increased clock frequency to sustain performance improvements [9, 24, 25, 27, 65]. Instead, performance improvement must now rely more on exploiting concurrency. At the same time, the delay of on-chip global wires has been growing relative to the delay of gates [26, 58], allowing a signal to be able to reach only a smaller fraction of the chip. As a result, future microarchitectures must be distributed to better tolerate the communication delay.

EDGE architectures [11] are proposed to aggressively exploit concurrency and tolerate communication latency in future wire-delay dominant VLSI technologies. EDGE architectures have two defining features: *block-atomic execution* and

direct instruction communication.

Block-atomic execution: The EDGE architecture aggregates a group of instructions into a single entity called a *block*. All the instructions within a block execute atomically: from the programmer's point of view, either all the instructions within a block commit or none of them commit. A block can be conceptually thought of as a very long instruction with many operations. Unlike VLIW processors, individual instructions within a block execute dynamically to tolerate latencies unknown at compile time. By aggregating multiple instructions into one block, the hardware can execute the instructions within a block in data-flow fashion with high efficiency. When mapped onto a distributed microarchitecture with sufficient execution bandwidth, the data-flow execution model within a block and the speculation across multiple blocks also facilitate the execution of a large number of instructions concurrently. Another benefit of the block-atomic execution model is that it naturally facilitates high-bandwidth instruction fetch, which has been a challenge for superscalar processors [23, 108].

Direct instruction communication: Within the same block, the EDGE architecture encodes the dependency among instructions through direct instruction communication. The ISA provides support for specifying the consumers of an instruction directly, instead of specifying them indirectly using source register names. This enables execution of instructions within a block in a dataflow fashion. Once an instruction receives all its operands from the producer instructions, it can execute and forward its output to its consumer instructions. Direct instruction communication also simplifies the hardware because it does not need to perform functions such as register renaming to dynamically compute the data dependency across instruc-

tions within the same block. Memory dependences, however, must still be expressed using a shared namespace and the ordering information about the memory instructions within the same block is encoded in the block. Instructions from different blocks still communicate through the registers and memory.

EDGE architectures reduce the complexity and cost of building a microprocessor capable of maintaining a large instruction window by transferring part of the complexity to the compiler. To achieve high performance and power efficiency, the compiler must be able to make good decisions on block formation [57, 85] and instruction scheduling [15] to increase concurrency and tolerate communication latency among instructions within a block.

EDGE architectures are designed with distributed microarchitectures in mind. Distributed microarchitectures make it possible to make use of a large number of independent execution resources to achieve high instruction-level parallelism. To tolerate the communication latency associated with distributed microarchitectures, the hardware maintains a large instruction window and schedules instructions in the instruction window out of order. The compiler helps mitigate the communication latency by placing producer-consumer instructions close to each other.

4.2 The TRIPS Prototype

The TRIPS prototype is the first instantiation of an EDGE architecture. It uses a new EDGE ISA and is implemented in a distributed, tiled microarchitecture. It has a set of well-defined on-chip networks to communicate across different tiles. It has a compiler that compiles large C and Fortran programs. The hardware uses speculation to maintain a 1024-entry instruction window and does dynamic, out-of-order execution of the instructions. The TRIPS prototype has been implemented in silicon

following an ASIC methodology and demonstrated to work without any hardware bugs found so far. This section briefly describes the ISA and the microarchitecture of the TRIPS prototype. The next section will focus on the instruction cache, which is one part of my role in the design and implementation of the TRIPS prototype chip.

4.2.1 The TRIPS ISA

As an EDGE architecture, the TRIPS ISA [60] aggregates up to 128 regular instructions into one block and encodes the data dependences among the instructions in the same block into each instruction. The motivation for choosing up to 128 instructions within a block was to give the compiler writers a large space to try aggressive optimization techniques [6, 86] and get a sense of the performance limit of the capabilities of this architecture. The ISA was defined with a distributed grid processor microarchitecture [67] in mind. Besides block-atomic execution and direct instruction communication, the TRIPS ISA also has the following differences from conventional RISC architectures.

Read/Write instructions: Read instructions and write instructions are special instructions that specify the live register inputs and outputs of a block. Read instructions specify the input registers for a block and the instructions within the block that consume those values. Similarly, the write instructions specify live registers produced by a block. By isolating register accesses using explicit read and write instructions, all other instructions strictly produce and consume temporary values that are valid within a block and never access the global registers.

Target encoding and fanout: Instructions do not encode their source operands; they encode only their consumers. The TRIPS compiler assigns labels for all instructions in a block, and the hardware interprets these labels to map instructions to appropriate locations in the hardware.

However, if an instruction produces a value that is needed by many consumers, encoding limitations prevent the TRIPS ISA from specifying all of the consumers in the producer instruction. In such cases, the ISA inserts additional `mov` instructions called *fanout* instructions to forward the results to every consumer.

Expressing control flow: A TRIPS block is a single-entry, multiple-exit region of instructions with no internal transfer of control flow. Instructions within a TRIPS block do not contain any control dependences. The only dependences are true data dependences and dependences enforced via loads and stores to data memory. A block can contain multiple branches but only one of the branches must be taken at runtime. A taken branch transfers control to a succeeding block, not to another instruction within the same block.

4.2.2 The TRIPS Microarchitecture

The TRIPS prototype chip consists of two processors located next to an array of non-uniform cache access (NUCA [42]) L2 cache banks. Figure 4.1 provides an organizational overview of the TRIPS chip. Each processor has 80 KB of L1 instruction cache and 32 KB of L1 data cache and is capable of issuing 16 out-of-order instructions per cycle from up to 1024 in-flight instructions. A processor can utilize all the hardware resources for one thread or up to four simultaneously multi-threaded (SMT) threads. The two processors share the L2 cache. The sixteen L2 cache banks are connected through an On-Chip-Network (OCN [21]) and together provide

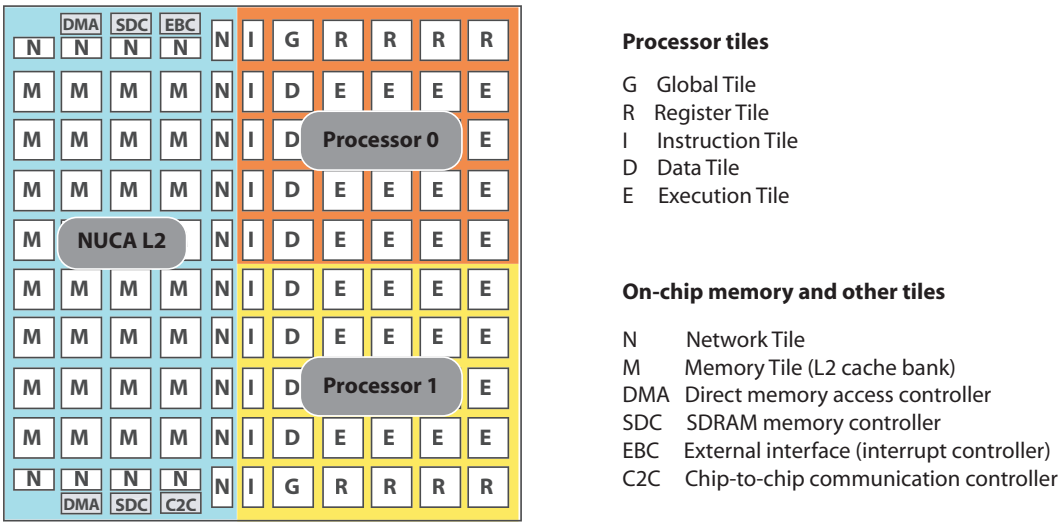


Figure 4.1: TRIPS chip overview (Figure from [66]).

a 1 MB L2 cache in the prototype chip.

The Processor

Each processor core is implemented using 30 tiles that belong to five unique types: one global control tile (GT), 16 execution tiles (ET), four data tiles (DT), four register tiles (RT), and five instruction tiles (IT). These tiles are connected via multiple networks to pass control and data around.

- **Global control tile (GT):** The GT sequences the overall execution of a program. It controls the fetch, execution, commit, next-block prediction and misspeculation recovery of the processor. Both the next-block predictor and the tag part of the instruction cache are located in the GT.

- **Execution tile (ET):** Each ET consists of an integer and floating point unit, a 64-entry reservation station, and a standard single-issue execution pipeline.

The regular instructions within a block are mapped to the ETs, with up to eight instructions per ET. Up to eight blocks can be in-flight in any cycle.

- **Register tile (RT):** Each RT contains a portion of the architecture and physical register file. The RT receives register read instructions and send the values of the registers to the regular instructions that are mapped to the ETs. It also receives the register write instructions and value produced by the regular instructions and commit the register writes to the architectural register.

- **Data tile (DT):** Each DT consists of an L1 data cache bank, cache miss handling logic, load/store queues, and a 1-bit dependence predictor to predict the dependences among in-flight memory loads and stores. Each DT receives the memory ordering information within each block. The total L1 data cache is 32KB, 2-way set associative with 64-byte cache lines interleaved among the four DTs.

- **Instruction tile (IT):** The five ITs comprise the L1 instruction cache of the processor. The top IT stores register read/write instructions for the RTs and the other four ITs store regular instructions for one row of ETs each. Each IT is 16KB with 16-byte cache lines. The whole L1 instruction cache is 2-way set associative.

- **The control and data networks:** The five types of tiles in the processor core are connected by multiple control and data networks. All the networks are hop by hop and do not use global, long wires. These networks include the on-chip operand network (OPN [22]) for forwarding data values, the global dispatch network (GDN) and global refill network (GRN) for fetching from the

instruction cache and refilling blocks into the instruction cache, and various other networks for sending commands and collecting status.

The Secondary Memory System

The TRIPS prototype chip contains a 4-way, 1 MB, on-chip L2-cache, implemented using 16 memory tiles (MTs) as shown in Figure 4.1. Each MT contains a 64 KB data bank, which may be configured as a cache bank or as a byte-addressable scratch-pad memory. The MT requires three cycles from receiving a request to producing the first reply packet. The network tiles (NTs) surrounding the MTs translate memory addresses to determine where the data for a particular address may be found. The NTs and MTs are clients on another network called the On-Chip Network (OCN), which is a 4×10, two-dimensional, worm-hole routed network [21]. The OCN also interfaces with each of the ITs on the edge of the TRIPS processors to provide high-bandwidth L2 cache access.

4.2.3 My Contributions

Development of the TRIPS prototype was a multi-year effort that involved contributions from many people. From the initial kickoff of the TRIPS prototype project in mid-2003 to the bringup testing of the prorotype chip at the end of 2006, as a member of the prototype hardware team, I made the following contributions to the TRIPS prototype:

- **Design, implementation, & verificationf of the instruction tile:** As the owner of the instruction tile, I designed the internal structure of the instruction tile, including choosing the capacity of the instruction cache. I wrote the Verilog code for the instruction tile, performed the timing and area closure, and

verified its correctness by constructing both manual test cases and automatic randomized self-checking testbenches.

- **Development of the cycle-accurate microarchitecture simulator:** I wrote the instruction cache and the uncore part (the model that emulates the behavior of the on-chip L2 cache) of the microarchitecture simulator, which has been used extensively by both the hardware and software team for performance evaluation & validation, functional validation, and evaluation of compiler optimizations. I also wrote a subset of the assembly micro-benchmarks to test targeted features of the microarchitecture simulator.

- **Pre-silicon verification:** Besides the verification of the instruction tile, I also took part in the verification of the L2 cache bank, the coverage analysis of the TRIPS prototype processor core, the verification of the chip-level Verilog, and the verification of the netlist after synthesis. Together with Paul Gratz, I completed the verification of the L2 cache bank using both manually constructed corner test cases and automatic randomized self-checking testbenches. We uncovered more than ten bugs in the Verilog code of the L2 cache bank. The verification of the L2 cache bank was a trailblaze effort of the verification of the whole TRIPS project. As part of this effort, I wrote the first randomized self-checking testbench of the TRIPS project. I was also responsible for the coverage analysis of the TRIPS prototype processor core, in which I collected statistics to decide which features of the Verilog code need to go through more tests so that we can be confident the processor core has been tested thoroughly. As part of the chip-level verification of the Verilog, I wrote a subset of the diagnostic tests that tested the functionality of the L2 cache and the OCN. Lastly, I was in charge of the verification of the full-chip

netlist. I developed the testbech for the full-chip netlist which was used in the extensive regression tests before tapeout. I also took part in the debugging of bugs found during the post-synthesis verification.

- **Chip bringup testing:** After the tapeout, I was part of the chip bringup team that wrote targeted tests to verify the correctness of the actual chip. I was responsible for the testing and debugging of the L2 cache and the OCN in this effort.

4.3 Instruction Cache in the TRIPS Prototype

The instruction cache in the TRIPS prototype is different from the instruction cache in conventional superscalar processors in several ways. First, the TRIPS instruction cache stores instructions in the unit of blocks, which are much larger than the unit of caching in normal instruction caches. Second, the distributed nature of the instruction cache itself and the TRIPS microarchitecture necessitates distributed protocols to perform instruction fetches and instruction refills.

4.3.1 Storing TRIPS Blocks

In the TRIPS prototype, a block is the basic unit of instruction fetches and refills. A block contains the regular instructions and the register read and write instructions. It also contains some meta information about the block such as the ordering information about the loads and stores within the block.

The format of the TRIPS block, as shown in Figure 4.2, matches the layout of the tiles in the TRIPS processor core. A header chunk stores the register read and write instructions needed by the register tiles and the meta information about the block needed by the global control tile. Each of the up to four body chunks stores

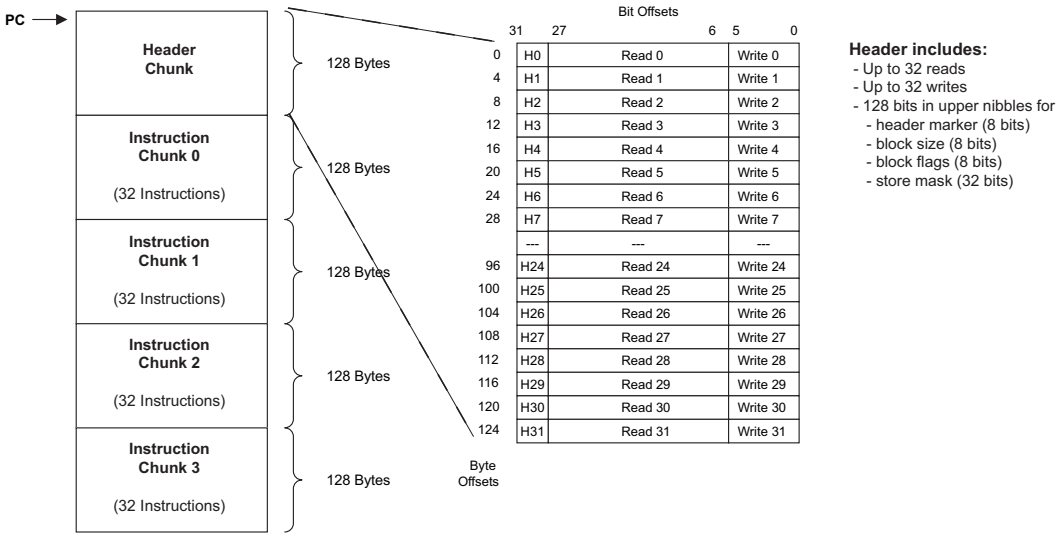


Figure 4.2: Block format in the TRIPS prototype ISA (from [82])

the regular instructions that are needed by each row of the execution tiles. Both the header chunk and body chunk are of fixed size of 128 bytes each, i.e., 32 register read/write instructions for the header chunk and 32 regular instructions for each of the body chunks. If there are not enough useful instructions, the unused space is padded with NOPs. To save space in main memory and the L2 cache, a block can have between 1 and 4 body chunks of regular instructions. However, when stored in the instruction cache, a block always occupies the same space of one header chunk and four body chunks. So from the instruction cache's point of view, the size of a block is fixed at 640 bytes. This design decision simplifies the complexity of the hardware but results in inefficiency in the utilization of the instruction cache.

Correspondingly, the instruction cache of the TRIPS prototype is distributed into five tiles, one tile to store the header chunk and one tile for each of the four body chunks of a block.

4.3.2 Distributed Protocols in the Instruction Cache

Another unique feature of the instruction cache in TRIPS is that it is distributed into multiple tiles: the tags of the instruction cache are stored in the global control tile and the data part of the instruction cache is further divided into five instruction tiles. Decoupling the tags from the data in the instruction cache enables the capability to read the tags independently of the data in the instruction cache, which can be beneficial as we will see in the next chapter. At the same time, both the instruction fetch and the instruction refill involve distributed protocols.

Instruction Fetch

Because of the large number of instructions in a TRIPS block, fetching a block from the instruction cache to the execution tiles and register tiles takes multiple cycles. Figure 4.3 shows the fetch pipeline from the global control tile's point of view. After the GT gets the address of the block to be fetched from the next-block predictor, it looks up the I-cache tags to detect if the block resides in the instruction cache. If so, it generates a fetch command, which includes the way of the block in the instruction cache and its index in the corresponding way. From the time that the GT generates the fetch command for a block in cycle 5, it takes a total of eight cycles to read the instructions in each instruction tile. Every cycle, 16 bytes of instructions (four regular instructions from each body chunk and four read and write instructions from the header chunk) are read from each of the five ITs, providing a total fetch bandwidth of 80 bytes per cycle.

Once the GT sends out the fetch command, the command propagates via the global dispatch network, which goes through every IT to every RT and ET, one tile per cycle, delivering the corresponding instructions to each tile. Figure 4.4 shows

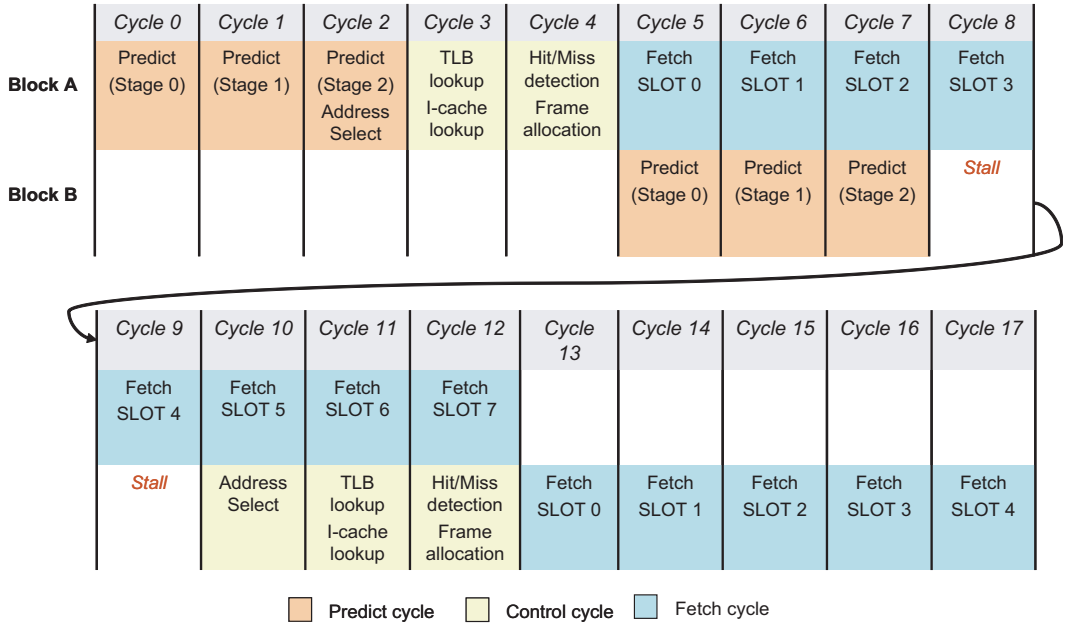


Figure 4.3: Fetch pipeline (Figure from [66]).

the relative time of arrival of the instructions to each RT and ET from the time the command is sent out by the GT. The width of the global dispatch network is 128 bits. Assuming that the block dispatch command is issued by the GT in cycle X, the closest ET (upper-left corner) receives its first instruction for that block in cycle X+4, and continues receiving one instruction per cycle until it receives its last instruction for the block in cycle X+11. The farthest ET (`ET15`) receives its first instruction for the block in cycle X+10, and its last in X+17.

While the latency to complete a distributed fetch operation is relatively large (18 cycles), multiple block fetches can be pipelined, so that at steady-state peak operation, each ET receives one fetched instruction per cycle with no fetch bubbles in between blocks. Figure 4.3 shows how the fetches for two blocks are pipelined.

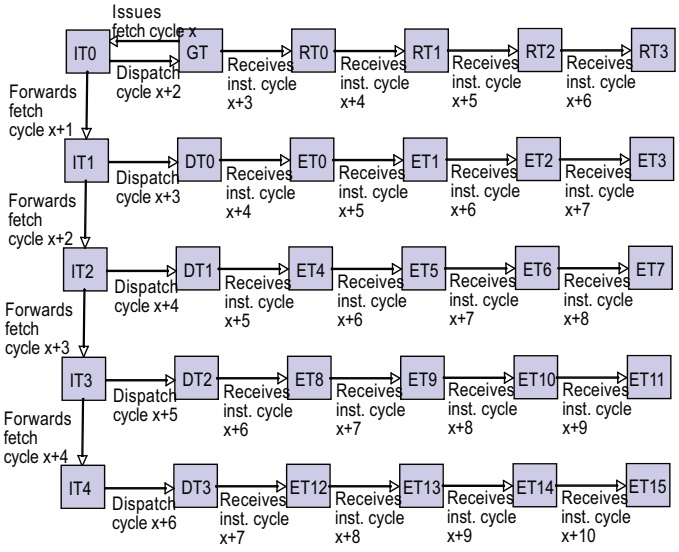


Figure 4.4: Timing of block fetch and instruction distribution. The figure depicts the delivery time of the first instruction at each ET/RT. Each tile continues to receive a new instruction each cycle for the next seven cycles.

Handling Instruction Cache Misses

The distributed organization of the instruction cache and the L2 cache, along with the large block size, makes refilling blocks when there is an instruction cache miss quite expensive. Figure 4.6 shows the networks involved with refilling a missing block from the L2 cache. The GT first sends out the physical address of the missing block on the global refill network (GRN). After each IT receives the address, it calculates the address of its chunk of instructions and sends out two independent read requests to the secondary on-chip network (OCN). Each request will bring 64 bytes of instructions, in five OCN packets, into the IT. When an IT completes its fill operation, it sends a notification signal upwards to its top neighbor using the global status network (GSN). The IT sends such a notification signal only if it has already received a similar signal from its bottom neighbor. The top-most IT notifies

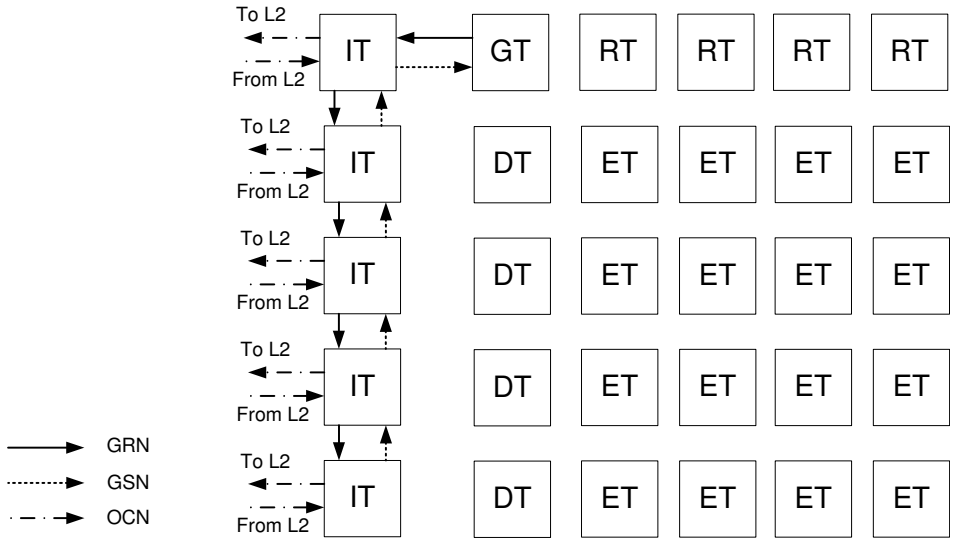


Figure 4.5: Networks involved in handling instruction cache misses.

the GT. Thus the refill completion signal daisy-chains all the way from the bottom-most IT to the GT, and the GT eventually receives one notification that marks the completion of the entire refill operation. Figure 4.5 shows the networks involved when a block is refilled from the L2 cache into the instruction cache.

Figure 4.6 shows the timing of the different events during the execution of the refill protocol. The I-cache miss handling involves communication among multiple tiles. First, the GT needs to send out the address of the missing block to all instruction tiles, which takes five cycles for the last IT to receive the message. Each IT then sends out two requests to the L2 cache and each request will bring in 64 bytes of an 128-byte chunk of the missing block. Each refill request is routed through the on-chip-network of the L2 system and can take up to 13 cycles to arrive at the destination L2 cache bank. For each request, the L2 cache bank sends 64 bytes of instructions back in five packets to the requesting IT through the on-chip-network. When all the ITs have received the the replies back from the L2 cache,

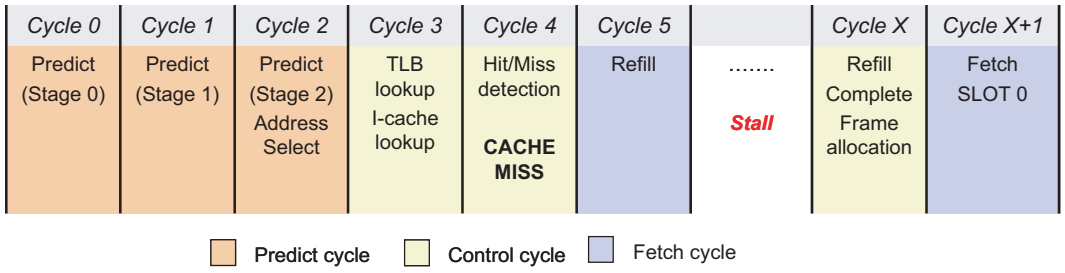


Figure 4.6: Refill pipeline (Figure from [66]).

the completion signal is propogated back to the GT. Because of the overhead of this distributed protocol, the time the GT must stall to wait for the completion of the refill can be quite long: the process can take more than 30 cycles if the whole block resides in the L2 cache and much longer if parts of the block have to be read from main memory.

The blocks refilled from the L2 cache are first stored in a refill buffer and written into the instruction cache only when they are fetched. This design has two advantages over writing the refilled blocks into the instruction cache directly. First, a small refill buffer reduces the pollution effect caused by mispredictions from the next-block predictor: the next-block predictor can produce illegal addresses that do not correspond to any block and writing these blocks into the instruction cache causes pollution. Second, with a refill buffer, the instruction cache array can perform the fetch and refill operation with just one read/write port without having any port conflicts. Only the small refill buffer needs to be dual-ported to avoid port conflicts when a fetch and a refill occur in the same cycle.

Although the I-cache refill protocol supports up to four outstanding refills in any cycle, for each hardware thread, only one I-cache refill can be inflight at any time. Therefore, when the TRIPS processor is running in single-threaded mode, the

instruction cache can be considered as blocking [45, 46]. This decision simplifies the hardware complexity but exposes the full I-cache miss penalty and can be a major issue for programs experiencing high I-cache miss rates.

4.3.3 Interaction with the Next-block Predictor

As shown by Figure 4.3, the next-block prediction [73] operations are tightly coupled with the instruction fetch operations. The next-block predictor only needs three cycles to predict the address of the next block while the fetch of a block takes eight cycles. As a result, the next-block predictor stalls for five cycles to synchronize with instruction fetch. An alternate design could have completely decoupled the prediction pipeline from the fetch pipeline using a *Fetch Target Buffer* [75]. By decoupling next-block prediction from instruction fetch, multiple refills can be initiated well ahead of a fetch, thus prefetching several blocks into the I-cache. Such a design, however, incurs additional hardware complexity and was not used in the TRIPS prototype. We will discuss this approach in more detail in the next chapter.

4.3.4 TRIPS Specific Features

The TRIPS instruction cache also has the following features that are specific to the TRIPS prototype and may not be found in the instruction cache of other EDGE architectures.

Interaction with the L2 Cache and the L1 Data Cache

The instruction tiles are the only places where the TRIPS processor core interacts with the rest of the system. Each IT has an on-chip-network (OCN) port to communicate with other components connected to the OCN. The OCN port is 128s bit

wide in each direction and is shared by the the IT and DT on the same row. Therefore, each IT must arbitrate the L2 traffic due to the data cache and the instruction cache on the same row, which goes through the same OCN port.

Implementation

The TRIPS instruction cache is implemented using an ASIC methodology. The instruction cache array is implemented using a SRAM array macro of 128-bit width and 1024-entry depth, with a total capacity of 16KB. The SRAM array has one read/write port and can therefore perform a read or write operation in one cycle. The area of each IT is $1mm^2$ and all the 10 ITs on the two TRIPS processor cores account for 2.9% of the total chip area.

The TRIPS prototype chip was implemented in the IBM Cu-11, 130 nm ASIC process. It consists of more than 170 million transistors in a chip area of 18.30 mm by 18.37 mm. Figure 4.7 shows the die photograph of the full TRIPS prototype chip.

4.3.5 Comparison with I-cache in Superscalar Processors

Compared to the instruction cache in superscalar processors, the I-cache in the TRIPS prototype chip has higher I-cache miss rates, because of the larger code size as a result of the transformations by the TRIPS compiler to expose more instruction-level parallelism, and the inefficient utilization of the I-cache capacity due to the fixed-sized block design.

The evaluation of the TRIPS prototype chip confirms that the I-cache in the TRIPS prototype chip experiences higher I-cache miss rates than the I-cache in superscalar processors [19]. Table 4.1 shows some of the results from the TRIPS prototype evaluation study. The table compares the number of instruction cache

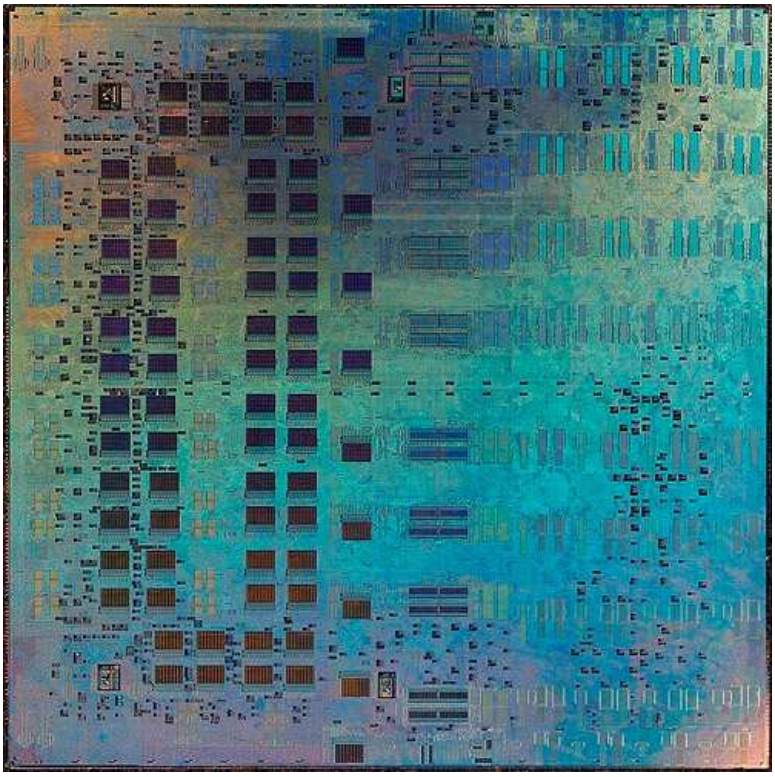


Figure 4.7: TRIPS die photo

Application	Core 2 I-cache misses	TRIPS I-cache misses	TRIPS/ Core 2 Ratio
bzip2	0	0	
crafty	2.55	15.8	6.2
gcc	0	0.3	
gzip	0	0	
mcf	0	0	
parser	0	1.1	
perlbmk	0.01	3.2	320
twolf	0	9.2	
vortex	0.48	8.1	16.9
vpr	0	0	
applu	0	0	
apsi	0.2	3.3	16.5
art	0	0	
equake	0	0	
mesa	0.01	7.9	790
mgrid	0	0	
swim	0.01	0	
wupwise	0.01	0	

Table 4.1: The TRIPS prototype vs. Core 2 Duo: number of I-cache misses per 1000 useful instructions (results from [19] by Gebhart et al.)

misses of the TRIPS prototype to that of the Core 2 Duo for some SPEC 2000 benchmarks. These numbers are collected through hardware performance counters and normalized to the number of I-cache misses per 1000 useful instructions on each processor. For example, the *crafty*, the Core 2 Duo experiences 2.55 I-cache misses every 1000 instructions while the TRIPS prototype experiences 15.8 instruction misses every 1000 instructions, which translates into an even smaller number of TRIPS blocks. The column labeled "TRIPS/Core 2 Ratio" is obtained by dividing the number of TRIPS I-cache misses by the number of Core 2 Duo I-cache misses. Even though each processor in the TRIPS prototype chips has 80KB of instruction cache whereas each processor in Core 2 Duo has only 32KB of instruction cache, the TRIPS prototype experiences significantly more I-cache misses than the Core 2 Duo does. Some of this gap in I-cache miss rates between the TRIPS prototype and Core 2 Duo can be narrowed by instruction prefetching because Core 2 Duo performs hardware I-cache prefetching while the TRIPS prototype does not do any I-cache prefetching. We will discuss instruction prefetching for EDGE architectures in the next chapter.

4.4 Instruction Cache in TFlex

As the first effort to demonstrate the feasibility and evaluate the potential of EDGE architectures, the TRIPS prototype made some simplifications in terms of both the architecture and the microarchitecture. It also always spreads the instructions in a block to all the tiles to achieve high concurrency, although some applications may have very limited concurrency and are better suited to a smaller number of tiles. The TFlex [41, 43] microarchitecture was proposed to address the limitations [79] of the TRIPS prototype. As a microarchitecture sharing the same ISA as TRIPS,

TFlex bears many similarities to TRIPS. This section briefly describes TFlex and the instruction cache in TFlex.

4.4.1 TFlex: A Composable Lightweight Processor Microarchitecture

TFlex consists of a large number of identical, fine-grained processor cores, which can be dynamically aggregated to form larger, more powerful single-threaded processors on the fly. Thus, the number and size of the processors can adjust to provide the configuration that best suits the software needs at any given time. The same software thread can run transparently—without modifications to the binary—on one core, two cores, and up to as many as 32 cores.

Like TRIPS, TFlex also uses a tiled microarchitecture. However, each tile is a full-fledged processor core that has a block-management logic performing the function of the GT in TRIPS, an instruction cache, a data cache, and functional units. The cores communicate through protocols similar to those used in the TRIPS prototype. The main difference is that there is no fixed, centralized control like the GT in TFlex. Instead, depending on the address of the block being executed, one core will be chosen as the owner of the block, initiating the fetch of the block and committing the block (the GT functions in the TRIPS prototype). If the next block to be executed has a different owner core, control signals need to be transferred to the new owner. More detailed information about how the mechanisms to support composability can be found in [43]. Next we describe how we modified the TRIPS instruction cache to support TFlex.

4.4.2 Extending the Instruction Cache in TRIPS to TFlex

To extend the TRIPS instruction cache to TFlex, the instruction tiles used in TRIPS that store instructions for each row of the execution tiles or register tiles have to be fully distributed to each core, as shown in Figure 4.8 on 16 cores.

As discussed earlier in this chapter, the TRIPS instruction cache has one IT for storing the header chunks and four ITs for storing the body chunks. In a TFlex processor with 16 cores, the IT that stores the header chunks are evenly divided into 16 smaller header caches and each core will have a header cache $1/16^{th}$ of the size of the total header cache capacity. The total capacity of the four ITs that store the body chunks are combined and evenly divided into 16 body caches, one body cache per core. The tag of each block is co-located with the header chunk. Therefore, only the header caches have tags and the body caches are tagless. A block is stored in the instruction cache this way: the header cache of the owner core stores the whole header chunk of the block and all the cores store an equal portion of the regular instructions in the block. As a result, all the instructions that are mapped to a core are stored locally in its instruction cache.

The I-cache design discussed above spreads the instructions of a block evenly to all the cores and therefore has the smallest tag overhead in the instruction cache because only one tag is needed for each block. It is possible to come up with other I-cache designs that allow instructions within a block to be stored more flexibly in different cores. For example, the instructions of a block can be stored entirely in the I-cache of one core instead of the I-cache of all the cores. Such I-cache designs are more desirable if the microarchitecture supports a dynamic block mapping policy [80]. To support flexible storage of the blocks in the I-cache, the I-cache must be managed at a finer granularity and the tag overhead can increase significantly. In

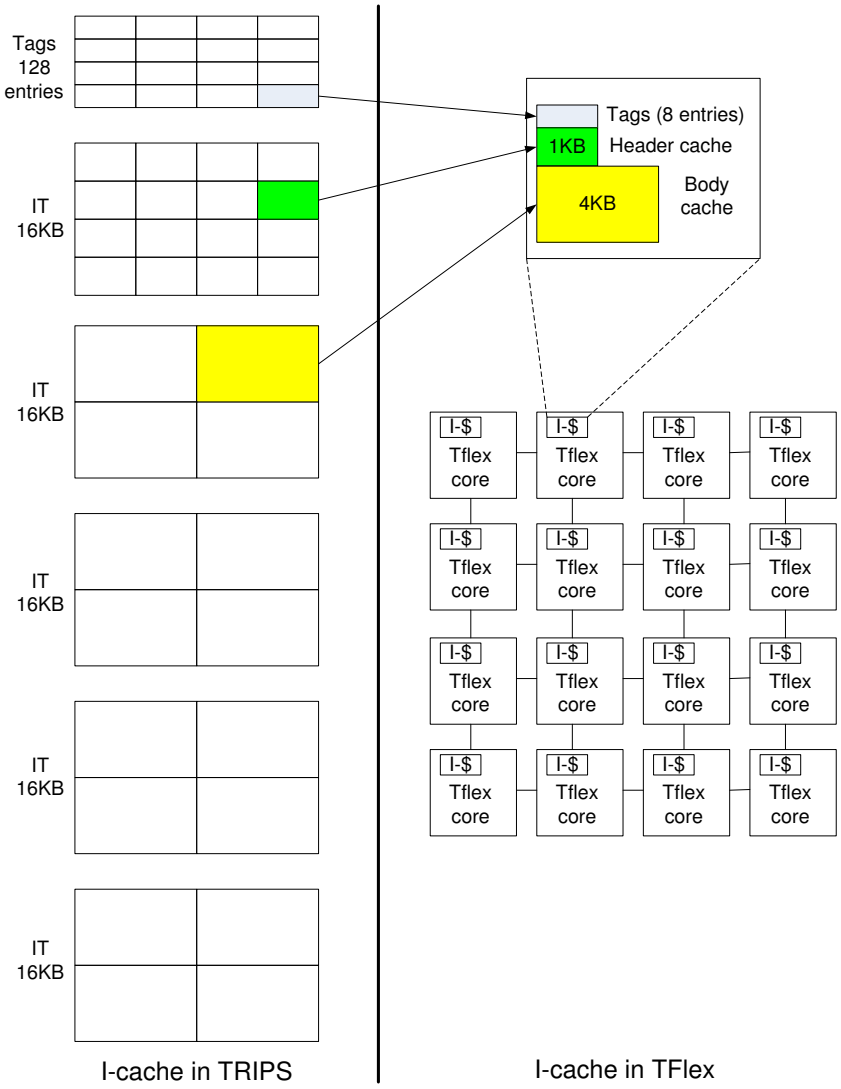


Figure 4.8: Extending the TRIPS I-cache to TFlex

this dissertation, we only consider the I-cache design that supports the symmetric mapping of instructions to all the cores and leave the I-cache designs that support more flexible mapping of instructions for future work.

Instruction fetch: During instruction fetch, the owner core looks up its local header cache tags to decide if the block resides in the instruction cache. If so, it broadcasts the fetch command to all the cores. The fetch command includes information such as in which way of the set the block is stored in the instruction cache and the index of the block in the selected way. Each core then computes the index into its local body cache based on the index received from the owner core and the local core's id.

Instruction refill: If the owner tile finds that the block to be fetched is not in the instruction cache, it broadcasts a refill command to all the cores. At the same time, the owner core sends out the request to load the header chunk of the missing block from the L2 cache. After receiving the refill command, each core computes the address of the portion of the instructions that it needs to load from the L2 cache based on the starting address of the missing block and its own id. When the reply comes back from the L2 cache, each core must send the refill completion signal back to the owner tile. Since only the owner core has the tag of a block, the protocol must ensure that the tag, the header chunk, and the pieces of the body chunk stored in each core are consistent. This I-cache design has the property that either a whole block is resident in the I-cache or none of the instructions of a block is resident in the I-cache. An alternative approach would be to allow a block to be partially resident in the I-cache. When a block that is partially resident in the I-cache needs to be fetched, only the portion of the block that is missing from the

I-cache is refilled. Such a design causes more tag overhead but can manage the I-cache at a finer granularity. We leave such I-cache designs for future work.

Support for composability: As a composable processor, the number of cores that participate in the execution of a program can change. When the number of cores that participate in the execution changes through reconfiguration, the instruction cache must also be reconfigured. The reconfiguration involves changing the mapping function from the block address to the owner cores and also how the regular instructions are mapped to each core. When the number of cores changes, the owner core still stores the whole header chunk of a block. However, the portion of the regular instructions that are stored in each core changes. If the number of cores doubles, the body cache of each core only needs to store half of the instructions for each block and can therefore store twice the number of blocks. The collective header cache across all the cores can also store twice the number of blocks because of the doubling of the cores. As a result, the total capacity of the instruction cache scales linearly with the number of cores.

4.5 Summary

In this chapter, we discussed issues related to the design and implementation of the instruction cache for EDGE architectures. In particular, we presented the design and implementation of the instruction cache in the TRIPS prototype, the first instantiation of an EDGE architecuture implemented in silicon. We also discussed how the TRIPS instruction cache can be extended to TFlex, a composable lightweight processor using the TRIPS ISA.

EDGE architectures feature block-atomic execution and direct operand com-

munication between instructions within the same block. EDGE architectures are best suited for distributed microarchitectural implementations. These features all have direct implications on how the instruction cache in EDGE architectures should be designed and implemented.

Block-atomic execution requires the instruction cache to store instructions in the unit of blocks instead of regular cache lines of instructions. Because of the large size of the blocks, the fetch of a block takes multiple cycles, which affects how the fetch pipeline interacts with the next-block predictor. The penalty of instruction cache misses are also higher because more bytes have to be loaded from the L2 cache.

The distributed microarchitecture used in EDGE architectures provide high instruction fetch bandwidth necessary to keep the execution resources well utilized when the blocks to be fetched reside in the instruction cache. However, it also introduces extra latency when instruction cache miss occurs. The longer instruction cache miss penalty, along with the higher instruction cache miss rate, can be a bottleneck for the performance of an EDGE architecture processor. We discuss how to address these issues in the next chapter.

Chapter 5

Increasing the Instruction-Cache Efficiency in EDGE Architectures

EDGE architectures use a distributed microarchitecture with a large number of functional units and a large instruction window to achieve high instruction level parallelism. To efficiently utilize these execution resources, the instruction cache must be able to feed instructions to the execution units with high bandwidth. The instruction-cache design presented in the previous chapter provides sufficiently high instruction-fetch bandwidth when the fetched blocks reside in the I-cache.

However, features of EDGE architectures such as block-atomic execution and direct operand communication among instructions within the same block, along with the distributed protocols used in the microarchitecture, put more pressure on the instruction cache. The penalty due to I-cache misses in EDGE architectures can be higher than the penalty in superscalar processors, as is the case in the TRIPS

prototype chip. Furthermore, the instruction cache in EDGE architectures will typically incur higher miss rates than the I-cache in superscalar processors does. As a result, the instruction cache in EDGE architectures can be a performance bottleneck for some applications, which has been confirmed by the performance evaluation [19] of the TRIPS prototype chip.

This chapter investigates techniques to mitigate the effect of I-cache misses in EDGE architectures. These techniques are applicable to all microarchitectures implementing the EDGE ISA with little modification. In this dissertation, however, we evaluate these techniques in the context of TFlex.

5.1 I-Cache Issues in EDGE Architectures

Compared to the I-cache in superscalar processors, the I-cache in EDGE architectures have higher I-cache miss rates. Depending on the design, the I-cache in EDGE architectures can also have longer miss penalty.

5.1.1 Higher Miss Rates

The first issue is the relatively higher I-cache miss rate in EDGE architectures compared to superscalar processors. The higher I-cache miss rate is caused by three factors.

First, the static code size of programs for EDGE architectures is usually larger than the code size of programs for superscalar processors. Compilers for EDGE architectures use transformations such as loop unrolling and tail duplication to extract more instruction-level parallelism. These transformations, however, can dramatically increase the static code size. Furthermore, the intra-block dataflow execution model used by EDGE architectures also contributes to the increase in the

static code size. The dataflow execution model removes the data communication through reads and writes to shared registers. Instead, it passes operands around with explicit data movement instructions. In the current microarchitecture instantiation of the TRIPS ISA, if an operand is needed by several consumer instructions, the compiler inserts several data movement instructions just to pass the operand to the consumer instructions. In contrast, in a superscalar processor, the operand just needs to be written into a register by the producer instruction and can then be directly used by all consumer instructions. The evaluation of the TRIPS prototype shows that the data movement instructions used in TRIPS programs account for about 20% of the total instructions [19].

Another contributor to the higher I-cache miss rate in EDGE architectures is the NOPs used to fill under-full blocks. In TRIPS and TFlex, a block can have up to 128 regular instructions, 32 register read instructions, and 32 register write instructions. If the compiler can not find enough useful instructions to fill these slots, it will fill the unused slots with NOP instructions. In the instruction cache, a block always occupy the amount of space that can accommodate all these instructions. In other words, the NOPs used to fill the unused slots in a block are stored in the instruction cache. Storing NOPs in the instruction cache effectively reduces the size of the I-cache and leads to more I-cache misses. The evaluation of the TRIPS prototype shows that NOPs account for 50% of the slots in a block [19], which means that half of the instruction cache is wasted.

5.1.2 Longer Miss Penalty

Besides the higher I-cache miss rates, current instantiations of EDGE architectures also have longer I-cache miss penalty compared to superscalar processors. The size

of a block is much larger than the size of a cache line in a superscalar processor; in TRIPS and TFlex, the size of a block is 640 bytes whereas a cache line in a superscalar processor is usually 64 bytes. Refilling this many of bytes from the L2 cache and main memory takes longer time.

The distributed protocols used in the TRIPS microarchitectures adds overhead to the I-cache penalty. The refill command needs to be propagated to multiple tiles over multiple cycles. After each tile has refilled its part of the block, it needs to send the completion signal back to the tile which initiated the refill, which also takes multiple cycles. Because of this overhead, even if the missing block is found in the L2 cache, it takes tens of cycles to refill the block into the I-cache. This longer I-cache miss penalty causes bubbles in the execution pipeline. Unlike data cache misses, however, it is hard to tolerate I-cache misses and the processor has to stall when I-cache miss occurs.

5.1.3 Potential for Improvement

Table 5.1 shows the I-cache hit rates of the SPEC 2000 benchmarks that can run on the current TFlex simulator. The results are for a 16-core TFlex processor with each core having a 1KB header cache and a 4KB body cache. The total capacity of the instruction cache is therefore 80KB, the same as the instruction cache in the TRIPS prototype. The instruction cache is 4-way set associative. The latency of both the header cache and body cache in each core is 1 cycle. Other parameters of the simulation can be found in Table 5.3.

Two I-cache hit rates are shown in Table 5.1. The column "Overall hit rate" shows the I-cache hit rates of all the blocks that are fetched, regardless of whether a fetched block was committed or not. The column "Correctly speculated hit rate"

Application	Overall hit rate	Correctly speculated hit rate
wupwise	0.86	1.00
swim	1.00	1.00
mgrid	1.00	1.00
applu	1.00	1.00
mesa	0.78	**0.89**
art	1.00	1.00
equake	1.00	1.00
ammp	0.98	1.00
sixtrack	0.98	0.99
apsi	0.81	**0.78**
gzip	0.93	1.00
vpr	1.00	1.00
gcc	0.99	0.99
mcf	1.00	1.00
crafty	0.68	**0.68**
parser	0.96	1.00
perlbmk	0.67	**0.85**
vortex	0.69	**0.78**
bzip2	1.00	1.00
twolf	0.74	**0.76**
GeoMean	0.89	0.93

Table 5.1: Hit rates of a 80KB, 4-way instruction cache with 16 cores (5KB per core)

Application	I-cache efficiency
wupwise	0.05
swim	0.07
mgrid	0.11
applu	0.18
mesa	**0.69**
art	0.66
equake	0.17
ammp	0.36
sixtrack	0.44
apsi	**0.31**
gzip	0.39
vpr	0.44
gcc	0.52
mcf	0.60
crafty	**0.40**
parser	0.65
perlbmk	**0.57**
vortex	**0.34**
bzip2	0.05
twolf	**0.55**
GeoMean	0.29

Table 5.2: Efficiency of a 80KB, 4-way instruction cache with 16 cores (5KB per core)

is the hit rate of the committed blocks only. In most cases (except for *apsi*, the hit rate of the committed blocks is higher than the hit rate of all the blocks that are fetched. Because the hit rate of the committed blocks correlates more directly to the performance of the processor, we use this metric instead of the overall hit rate to in the remainder of this chapter.

Table 5.2 shows the block-level efficiency of instruction cache. Compared to the efficiency of the data caches, the instruction cache has higher efficiency. Several benchmarks (*wupwise*, *swim*, *mgrid*, *applu*, and *bzip*) have low I-cache efficiency but almost perfect I-cache hit rate because the number of blocks in the working set

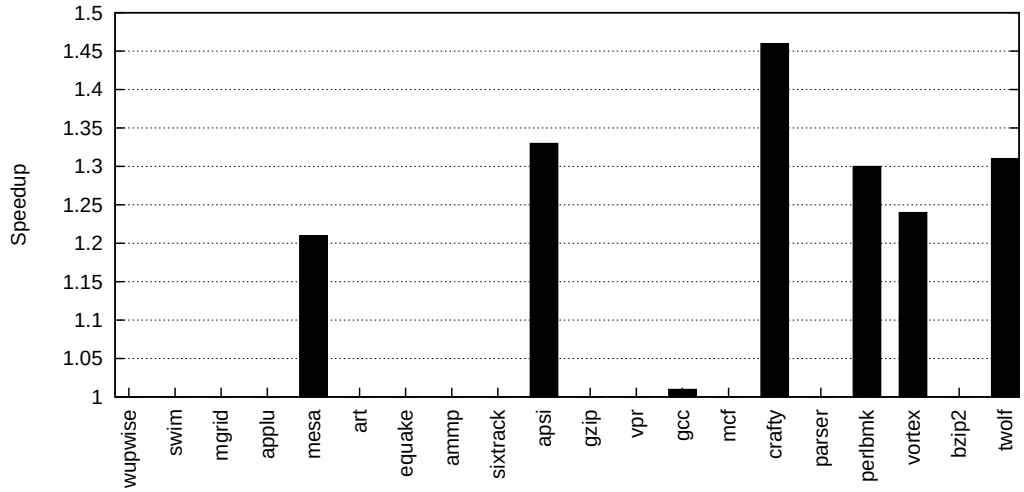


Figure 5.1: Speedups achieved with a perfect instruction cache

of these benchmarks are small so they use a fraction of the I-cache capacity. In contrast, the I-cache efficiency of *mesa*, *apsi*, *crafty*, *perlbmk*, *vortex*, and *twolf* are higher but they suffer much more frequent I-cache misses as shown in Table 5.1. In the remainder of this chapter, we focus our attention to these six benchmarks with high I-cache miss rates.

Figure 5.1 shows the performance gap between the realistic instruction cache and a perfect I-cache. As expected, the six benchmarks with high I-cache miss rates show a performance gap of at least 20% and up to 45% between the realistic I-cache configuration and a perfect I-cache. On the other hand, the other benchmarks with low I-cache miss rates show little speedup with a perfect I-cache.

5.2 Increasing the I-cache Efficiency Through Prefetching

A proven technique to increase the efficiency of the instruction cache is prefetching. Instruction prefetching can be performed via either hardware [28, 76, 87, 89, 91, 92] or software [56]. In this dissertation, we only focus on hardware prefetching schemes. Prior hardware instruction prefetching schemes can be classified into two categories: sequential prefetching schemes and non-sequential prefetching schemes.

Sequential prefetching aims at reducing the number of instruction cache misses caused by non-control-transfer instructions. The simplest form of sequential prefetching is next-line prefetching. In this scheme, whenever a cache line is fetched, the hardware tries to prefetch the next cache line [89]. This scheme can significantly increase the amount of traffic between the instruction cache and the L2 cache. To address this problem, two variations of this "next-line always" scheme exist: next-line on miss and next-line tagged. The "next-line on miss" scheme issues a prefetch for the next line only if the cache line currently being fetched results in a I-cache miss whereas the "next-line tagged" issues a prefetch for the next line only if the cache line currently being fetched results in a I-cache miss or is tagged (which means the line was prefetched into the I-cache earlier) [88].

Next-line prefetching can only hide short I-cache miss latencies. If the latency of the L2 cache is long or if the next line has to be prefetched from main memory, next-line prefetching issues the prefetches too late. A more aggressive sequential prefetching scheme, next N-line prefetching [87], prefetches the next N lines immediately following the cache line currently being fetched. Besides the ability to initiate the prefetches in a more timely manner, this scheme can also prevents non-sequential misses caused by control transfers where the target falls within the next

N lines.

Sequential prefetching, however, is generally not effective at eliminating I-cache misses resulting from control transfers to distant cache lines. A previous study shows that it is insufficient to just eliminate sequential instruction cache misses only [91] because some applications have a high proportion of function calls to small functions or frequent changes in control flow and can therefore cause frequent non-sequential I-cache misses. To eliminate non-sequential cache misses, two approaches have been proposed: those that use a dedicated table to retain information about the sequence of cache lines previously fetched by the program [28, 89] and those that rely on the branch predictor to run ahead of the instruction fetch [50, 76, 77] to derive the prefetch addresses. In this dissertation, we study the second approach, which relies on the next-block predictor (the counterpart of the branch predictor in a superscalar processor) to run ahead of the instruction fetch to provide candidate prefetch addresses. But before we discuss how this prefetching scheme works in EDGE architectures, we first describe how it works in superscalar processors.

5.2.1 Instruction Prefetching Driven by Look-ahead Branch Prediction in Superscalar Processors

Instruction prefetching driven by look-ahead branch prediction was first proposed by Reinman et al. in [76]. The idea is quite straightforward: a fetch target buffer (FTB) is used to decouple the branch predictor from the instruction fetch so that the branch predictor can work ahead of the instruction fetch [75]. The branch predictor stores the addresses of future cache lines that will be needed by the program into the FTB and the instruction fetch hardware consumes these addresses from the FTB. Since the branch predictor runs ahead of the instruction fetch, a prefetch engine can

examine the addresses already in the FTB to decide if the cache lines need to be prefetched before they are needed by the program.

Each entry in the FTB is a target. A target specifies a fetch block, which is a contiguous region of instructions that will be executed by the program in the future. A fetch block consists of instructions that are either non-branch instructions or branch instructions that have never been taken before. A fetch block ends either because it has reached the maximum allowed size, or because the next instruction is a branch instruction that has been taken earlier during the program execution. A target specifies the starting address a fetch block. It also specifies the size of a fetch block; otherwise the instruction fetch hardware and the prefetch engine do not know where to stop fetching/prefetching. The branch predictor needs to provide the size of each target. The input to the branch predictor is the starting address of a fetch block. The outputs from the branch predictor include the starting address of the next fetch block and the size of the current fetch block. In [76], the size of a fetch block can be at most three cache lines.

5.2.2 Instruction Prefetching Driven by Look-ahead Next-block Prediction in EDGE Architectures

Instruction prefetching driven by look-ahead next-block prediction matches particularly well with EDGE architectures for several reasons.

First, because of the large size of the blocks in EDGE architectures, the time it takes to fetch a block is longer than the time it takes to predict the address of the next block [66]. For example, in the TRIPS prototype, on an instruction cache hit, it takes eight cycles to fetch a block from the instruction cache but only six cycles to predict the address of the next block and update the branch predictor

(three cycles for the prediction and three cycles for the update). In TFlex, the branch prediction and the predictor update can be overlapped, which gives more slack for the branch predictor to run ahead. On an instruction cache miss, the gap between the instruction fetch and the next-block prediction is even larger because of the longer I-cache miss penalty. This means the next-block predictor can easily run ahead of the instruction fetch and the prefetch engine can have more time to initiate the prefetches when necessary.

Second, the distributed I-cache design used in EDGE architectures facilitates filtration of the prefetch requests. Before a prefetch for a block in the FTB is issued, the prefetch engine needs to probe the tags of the instruction cache to see if the block already resides in the instruction cache. If so, no prefetch is needed. In both TRIPS and TFlex, the tags of the instruction cache are decoupled from the body caches that store the instructions. Furthermore, the instruction fetch process only needs to access the tags in the first cycle of the fetch. After that, the fetch process can proceed without accessing the tags. This gives sufficient tag-probing bandwidth to the prefetch engine. In contrast, instruction prefetching driven by look-ahead branch prediction can cause significant contention on the tags of the instruction cache that either delays the prefetches or warrants a dual-ported instruction cache.

Third, in superscalar processors, a basic block can have only up to two possible successors: the fall-through block and the branch target block. Therefore, even the simpler sequential prefetching works well for many applications. In EDGE architectures, however, a block usually have more than two possible successors because of the use of predication. As a result, conventional sequential prefetching schemes do not work as well. Furthermore, the size of a block is much larger than the size of cache line in superscalar processors. If a wrong block is prefetched into

the instruction cache, it causes much more pollution than a wrong cache line that is prefetched into the instruction cache of a superscalar processor. Therefore, EDGE architectures must be more prudent in choosing the prefetch addresses and should rely on the next-block predictor to generate the prefetch addresses, rather than simply prefetch the next block.

5.2.3 Implementing Instruction Prefetching with Look-ahead Next-block Prediction in a Distributed Microarchitecture

The distributed microarchitectures used in EDGE architectures, however, also pose challenges to the implementation of fetch target buffer. Two questions need to be answered. First, how should a fetch target buffer be implemented in a distributed microarchitecture? Second, how should the look-ahead distance of the next-block predictor from instruction fetch be maintained?

The front end of a superscalar processor is centralized so implementing the fetch target buffer is trivial: it is simply implemented as a FIFO. Such a FIFO also works for TRIPS because the next-block predictor and the tags of the instruction cache are both centralized in the global control tile. But how can such a buffer be implemented in a distributed microarchitecture like TFlex where different entries of the fetch target buffer can be located on different cores? One way is to use pointers to construct a linked list. But linked lists implemented in hardware are rarely used because of their complexity.

A better approach is to make use of the block management mechanism, which supports speculation through next-block prediction and already exists in the distributed microarchitecture. The block management mechanism keeps track of the blocks that are inflight. For every inflight block, each core keeps track of the

status of the block using an entry in a circular buffer. Only the oldest block is non-speculative while all the other blocks are speculative. In the current TFlex design, where next-block prediction is coupled with instruction fetch (like the fetch pipeline in TRIPS described in 4.3.2) and no look-ahead instruction prefetching is performed, a new entry for a block is allocated only when the current block has completed fetching. To decouple the next-block prediction from instruction fetch, a new entry for a block is allocated whenever the address of the next block has been produced by the next-block predictor and arrives at the next owner core. At this moment, the new owner core can use the address to probe the I-cache tags to decide if a prefetch is necessary. If so, it sends the prefetch message to all the other cores to initiate a prefetch. The fetch of the new block, however, does not start until the current block has completed fetching. The owner core of the new block knows when the current block has completed fetching because it also participates in the fetching of the current block. Figure 5.2 shows how a distributed Fetch Target Buffer can be implemented on top of the existing block management mechanism in TFlex.

One consequence of relying on the existing block management mechanism to perform look-ahead instruction prefetching is that the look-ahead distance is constrained by the instruction window in terms of the number of inflight blocks. It is possible to prefetch beyond the instruction window. Prefetching beyond the instruction window requires extra buffer space to keep track of the addresses of the blocks that have been produced by the branch predictor.

The second question is how to maintain a reasonable look-ahead distance of the next-block prediction from the instruction fetch. Maintaining a reasonable look-ahead distance is important because it does not make sense to let the next-block predictor run too far ahead of the instruction fetch because as the next-block

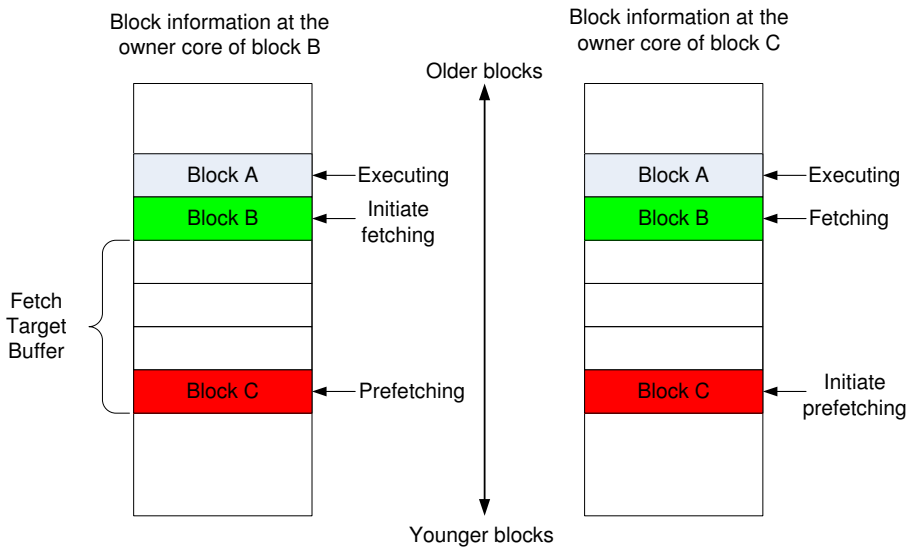


Figure 5.2: Implementing a distributed Fetch Target Buffer on top of the existing block-management mechanism in TFlex

predictor runs further ahead, the prediction accuracy drops and the prefetches are more likely to cause pollution to the instruction cache. A overly small look-ahead distance is also not desirable because the prefetches may be issued too late.

This issue is again trivial in a superscalar processor because the branch predictor can just stall when the number of entries in the FTB FIFO reaches some threshold and start predicting again once the number of entries in the FTB FIFO drops below the threshold. In a distributed microarchitecture like TFlex, however, the information about the FTB is scattered across different cores. For example, the next-block prediction may be performed on a different core than the one that is controlling the fetch of the current block and there may be other blocks that have been predicted and are located on other cores. To prevent the next-block prediction from running too far ahead of the instruction fetch, each core needs to keep track of how many blocks the next-block predictor is running ahead of the instruction

fetch. This again is achieved with the block management mechanism: when a core receives a block address produced by the next-block predictor, it allocates a new entry for the block, and informs all other cores to allocate a new entry with the same index for the block. Therefore, all the cores are synchronized on how far the next-block predictor is running ahead of the instruction fetch. Whenever the owner core of the youngest block observes an event that can change the distance between the next-block prediction and the instruction fetch, in this case the event can be the fetching of a new block or the receipt of a block address from the next-block predictor, it needs to check whether the next-block predictor should be started again or if it should be stalled. If the owner core of a block receives the block address from the next-block predictor, it checks the distance between the new block and the block that is being fetched. If the distance equals the threshold, the next-block prediction is stalled. On the other hand, if the owner core of the youngest block starts fetching a new block, it checks if the next-block predictor can be started again in case it has already been stalled. If the look-ahead distance is below the threshold, the next-block predictor is allowed to continue to predict.

Figure 5.2 shows an example of a Fetch Target Buffer with a run-ahead distance of four. In this example, when the owner core core of block B received the address of block B, it found that the look-ahead distance is zero so it predicted the address of the next block and send the new address to the next owner core. Upon receiving the address, the next owner core found that the look-ahead distance was still below the threshold of four so it continued to predict the next block address. This process continued until the owner core of block C received the address of block C. At this moment, the look-ahead distance has reached the threshold so the branch prediction is stalled. Later, when the owner core of block C detects that block B

has been fetched and the look-ahead distance is below the threshold again, it will resume the branch prediction.

5.3 Increasing the I-cache Efficiency by Storing Variable-sized Blocks in the I-cache

Another way to increase the I-cache efficiency for EDGE architectures is to design I-caches that can store variable-sized blocks. Until now, we have discussed I-cache designs that always allocate a fixed-sized space in the instruction cache for each block, regardless of how many useful instructions the block actually contains. While this fixed-sized block approach helps simplify the hardware complexity, it wastes a significant fraction of the I-cache capacity. For example, the evaluation of the TRIPS prototype shows that only 50% of I-cache capacity is used to store useful instructions [19], while the other 50% is wasted due to NOPs.

To be able to store variable-sized blocks in the instruction cache, the block format must be changed. The ISA of the TRIPS prototype already has support for variable-sized blocks. However, the TRIPS block format only allows variable-sized blocks to be stored in the L2 cache and main memory. To store variable-sized blocks in the instruction cache, a new block format is needed. In the remainder of this section, we discuss such a block format in the context of TFlex, although the idea works in the context of TRIPS, too.

TFlex inherits the block format from TRIPS, which has a separate row of register tiles and has a header chunk that contains the register read/write instructions. However, the TFlex microarchitecture is fully distributed and has no separate register tiles; the registers are evenly distributed across all the cores. Therefore, the TRIPS block format does not match the TFlex microarchitecture well. To match

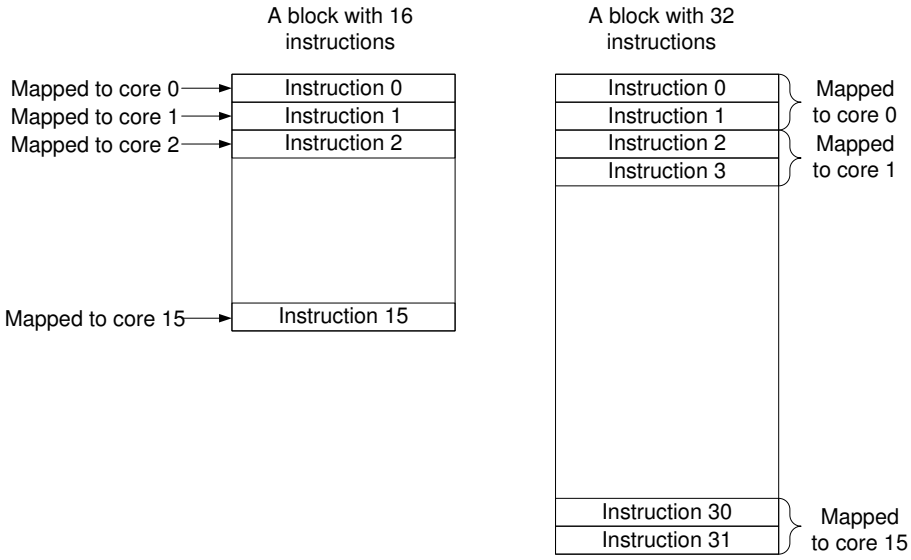


Figure 5.3: A new variable-sized block format for TFlex

the fully distributed microarchitecture of TFlex, the new block format should get rid of the header chunk. The header chunk can be removed by either treating the register read/write instructions as regular instructions or by getting rid of the register read/write instructions from the ISA altogether. The meta information about a block, such as the memory ordering information and block size, which was located in the header chunk, can be stored at fixed locations in the new block format. Figure 5.3 shows two blocks of different sizes.

With this new block format, it is convenient to encode blocks of different sizes. The harder question is how to store these variable-sized blocks in the instruction cache. We discuss a scheme that can store variable-sized blocks in the instruction cache with a hardware complexity that is close to the complexity of an instruction cache that only stores fixed-sized blocks.

This technique is called block compaction as shown in figure 5.4. The idea is that the hardware still manages the instruction cache as if all the blocks are full-sized

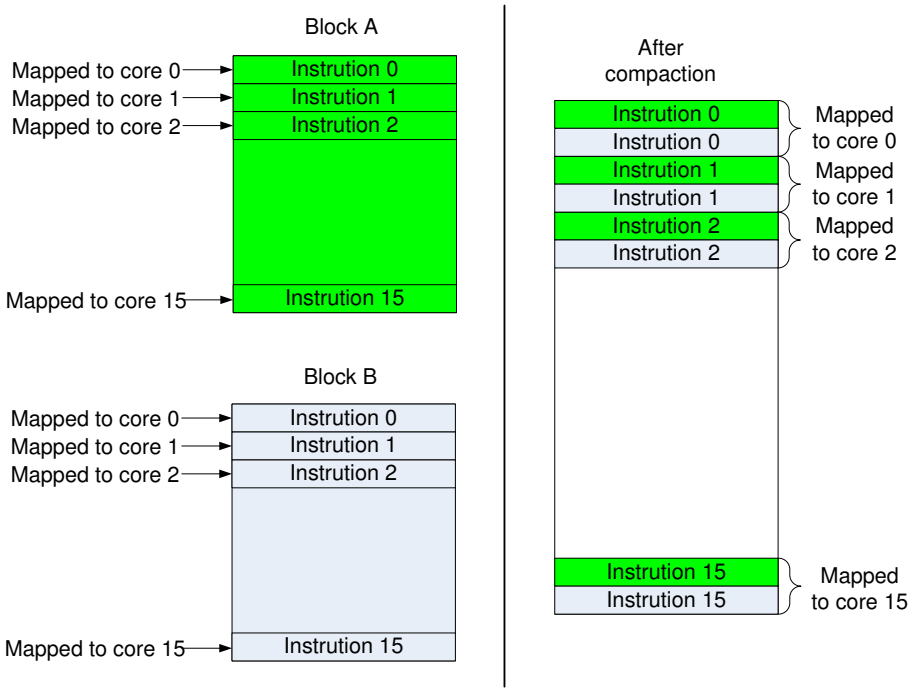


Figure 5.4: Compacting two small blocks into a larger block

blocks. However, several small blocks that are contiguous in the static code can be compacted into one block and can therefore share the space of one full-sized block in the instruction cache. This is to some extent similar to how several instructions share the same cache line in superscalar processors. The hardware in a superscalar processor manages the instruction cache at the unit of cache lines and one cache line contains multiple instructions. Here we can think of a full-sized block as a cache line. The TFlex hardware manages the instruction cache at the unit of full-sized blocks and the space of one full-sized block can contain several smaller blocks.

The complication of block compaction is that the position of an instruction within a full-sized block determines on which core the instruction will be mapped to execute. Therefore, the several small blocks that are compacted into one block can not be concatenated together; to ensure the correct instruction mapping, the

instructions from these small blocks must be properly *interleaved*. The compaction of the static code is performed by the compiler. During the compaction process, the compiler decides which contiguous small blocks should be compacted and properly interleaves the instructions from these blocks.

Although the several blocks that are compacted together share the same space of a full-sized block in the instruction cache, they are still independent blocks. That is, they are fetched, executed, and committed independently and therefore need to have different addresses. The starting address of a block by itself can no longer be used as the address of a block because several small blocks may share the same starting address after block compaction. Instead, the new block address should be composed by concatenating the starting address of the full-sized block with the offset of first instruction of an individual block from the starting address of the full-sized block. This is again performed by the compiler and encoded into the binary executable.

The hardware complexity of the block compaction scheme is almost the same as the complexity of the fixed-sized block design. When there is an I-cache miss, the hardware only needs to refill fixed-sized blocks from the L2 cache without worrying about size of the missing block and the complication that the missing block and the block that needs to be replaced may be of different sizes. The price for this low complexity in the hardware, however, is the changes that are required to the compiler. Because block compaction requires changes to both the ISA (block format) and the compiler, we do not simulate the performance benefits of this technique in full detail. Instead, we only estimate its potential benefits as discussed in the next section.

Parameter	Configuration
Instruction Supply	Partitioned 5KB I-cache with 1KB header cache & 4KB body cache, 1-cycle hit latency, Local/Gshare Tournament predictor (8K bits, 3-cycle latency) with speculative updates; Local: 512(L1) + 1024(L2), Global: 4096, Choice: 4096, RAS: 128, BTB: 2048.
Execution	Out-of-order execution, RAM structured 128-entry issue window, dual-issue (up to two INT and one FP). 128 architectural registers
Data Supply	Partitioned 44-entry LSQ bank, Partitioned 8KB D-cache (2-cycle hit, 2-way associative, 1-read port and 1-write port). 4MB S-NUCA L2 cache [42] (8-way associative, LRU, the L2 hit latencies vary from 5 cycles to 27 cycles depending on memory addresses) Average (unloaded) main memory. latency is 150 cycles
Interconnection Network	Each router uses round-robin arbitration. There are four buffers in each direction per router. The hop latency is 1 cycle.

Table 5.3: Microarchitectural parameters for a single TFlex core

5.4 Results

In this section, we evaluate the performance benefit of instruction prefetching driven by look-ahead next-block prediction. In particular, we compare the performance improvements when the distance that the next-block prediction runs ahead of the instruction fetch is varied. We also estimate the performance benefit of storing variable-sized blocks in the instruction cache through block compaction.

5.4.1 Methodology

The performance evaluation is done using an execution-driven simulator that can simulate both the TRIPS and the TFlex microarchitectures. We use the simulator to simulate the TFlex microarchitecture. Table 5.3 lists the microarchitectural

parameters. For all the experiments, we simulate a TFlex processor with 16 cores.

We simulate 10 integer benchmarks and 10 floating-point benchmarks from SPEC 2000. Each benchmark is run for a single SimPoint [84] region with the reference input. As discussed earlier, we only report results for the six benchmarks that have high I-cache miss rates and can potentially benefit from I-cache optimizations.

5.4.2 Prefetching Results

We first evaluate the performance of instruction prefetching driven by look-ahead next-block prediction.

Prefetching Results with Realistic Next-block Prediction

Figure 5.5 shows the speedups achieved by look-ahead instruction prefetching when the next-block predictor runs at different distances ahead of the instruction fetch. For example, "FTB depth=1" indicates that the next-block predictor can only run at most one block ahead of the instruction fetch. Figure 5.5 also shows the speedups of a perfect I-cache. All the speedups are calculated over the baseline I-cache with no prefetching.

All the six benchmarks benefit from the look-ahead instruction prefetching. As the run-ahead distance increases, the speedups first increase because a larger run-ahead distance improves the timeliness of the prefetching. If the next-block predictor runs only one or two blocks ahead of the instruction fetch, the prefetched blocks may not be able to arrive in time. However, as the next-block predictor runs further ahead, the speedups drop because of the drop in the next-block prediction accuracy. The further the next-block predictor runs ahead of the instruction fetch, the lower the prediction accuracy becomes. Therefore, even though the timeliness of

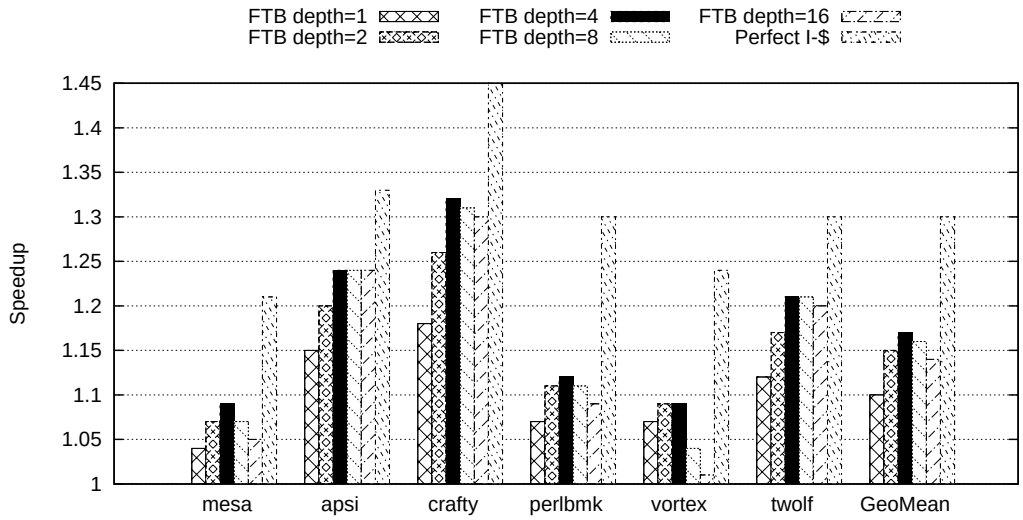


Figure 5.5: Speedups achieved by look-ahead prefetching with different FTB depths

the prefetching improves, the prefetching accuracy drops and the pollution caused by prefetching increases. At some point, the increase in pollution outweighs the improvement in timeliness and the performance begins to drop. Interestingly, for all the six benchmarks, the best performance is achieved when the next-block predictor runs four blocks ahead of the instruction fetch. On average, look-ahead instruction prefetching with a run-ahead distance of four gives a speedup of approximately 17% compared to the speedup of 30% by a perfect I-cache.

With instruction prefetching, the committed blocks can be classified into three categories: those that hit in the instruction cache, those that miss in the instruction cache, and those that hit in the MSHR of the instruction cache, which means the prefetched blocks have not arrived when the processor attempted to fetch these blocks. Figures 5.6, 5.7, and 5.8 show the fraction of these three categories of blocks.

Figure 5.6 shows the I-cache hit rate of the committed blocks when the run-

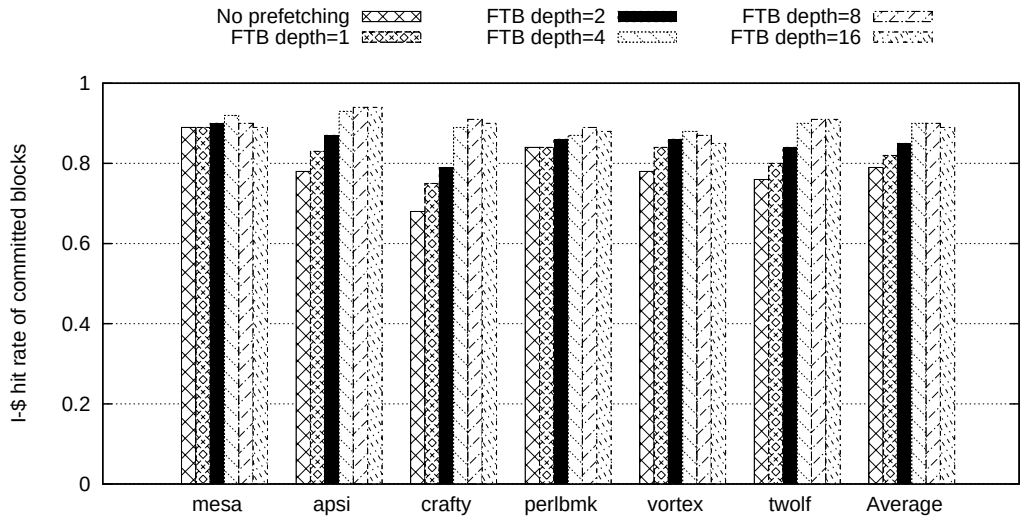


Figure 5.6: I-cache hit rate of committed blocks of look-ahead prefetching with different FTB depths

ahead distance changes. Compared with no prefetching, the look-ahead instruction prefetching improves the I-cache hit rate in all cases. For most benchmarks, the highest hit-rate is achieved when the next-block predictor runs four or eight blocks ahead of the instruction fetch.

Figure 5.7 shows the I-cache miss rate of the committed blocks when the run-ahead distance changes. Compared with no prefetching, the look-ahead instruction prefetching reduces the I-cache miss rate in all cases. However, as the run-ahead distance increases, the I-cache miss rate slightly increases due to the increased pollution.

Figure 5.8 shows the I-cache MSHR hit rate of the committed blocks when the run-ahead distance changes. These blocks are correctly predicted and prefetched into the instruction cache. However, the prefetches are initiated not early enough. When the processor tries to fetch such blocks, the prefetching has not completed yet. Even so, prefetching these blocks still helps improve performance because it

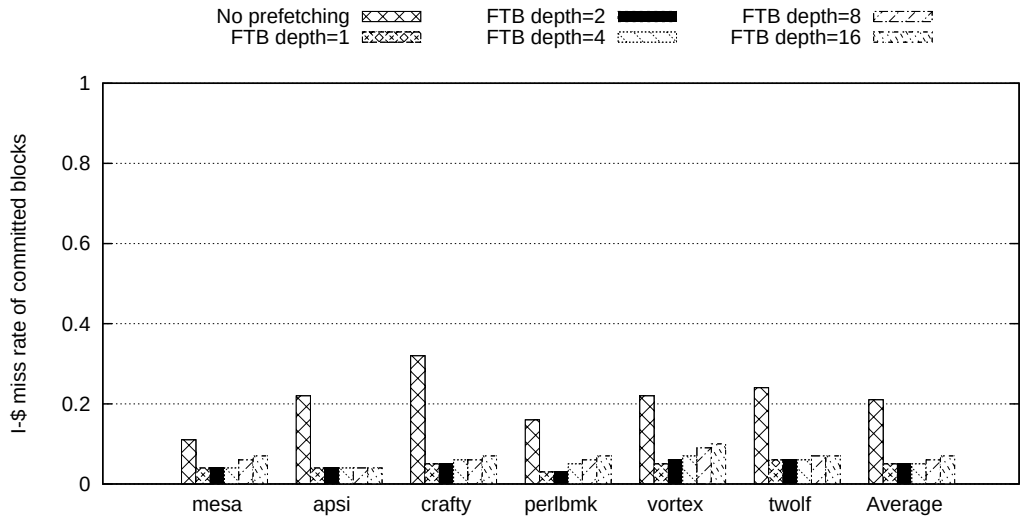


Figure 5.7: I-cache miss rate of committed blocks of look-ahead prefetching with different FTB depths

partially hides the penalty of the instruction cache misses. As the run-ahead distance increases, the MSHR hit rate drops in almost all cases because the prefetch engine is able to initiate the prefetches earlier. The only exception is *mesa*, the MSHR hit rate slightly increases when the run-ahead distance changes from 8 to 16.

Figure 5.9 shows the amount of traffic between the instruction cache and the L2 with look-ahead instruction prefetching. The traffic shown is relative to the traffic with no prefetching; a relative traffic of 1 means the traffic does not increase. As the run-ahead distance increases, the traffic between the instruction cache and the L2 increases as well because more useless prefetches are initiated. With a run-ahead distance of 4, the traffic increases by approximately 34% on average.

Table 5.4 shows the I-cache efficiency achieved by look-ahead instruction prefetching. Surprisingly, look-ahead instruction prefetching rarely improves the I-cache efficiency. In most cases, the I-cache efficiency actually drops as the prefetch engine becomes more aggressive and runs further ahead of the instruction fetch.

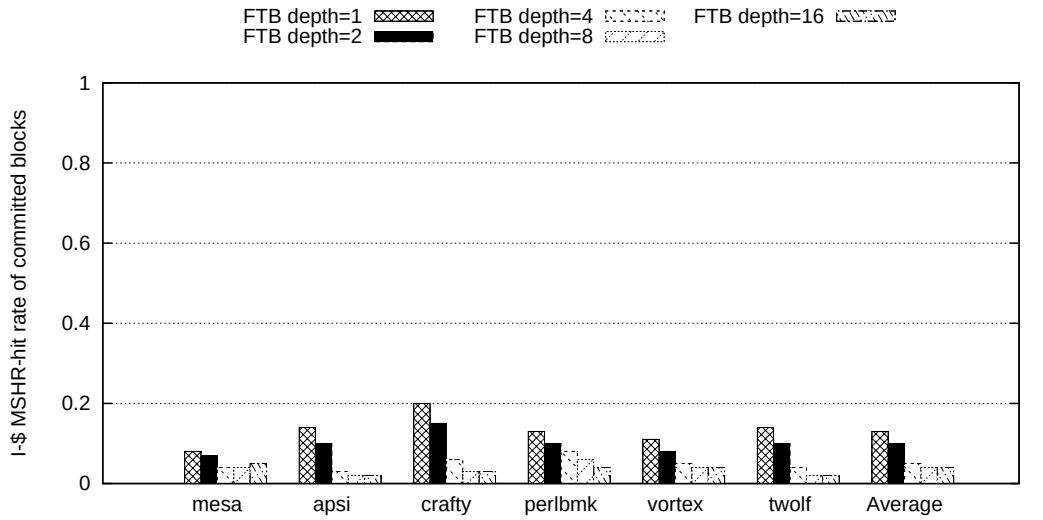


Figure 5.8: I-cache MSHR hit rate of committed blocks of look-ahead prefetching with different FTB depths

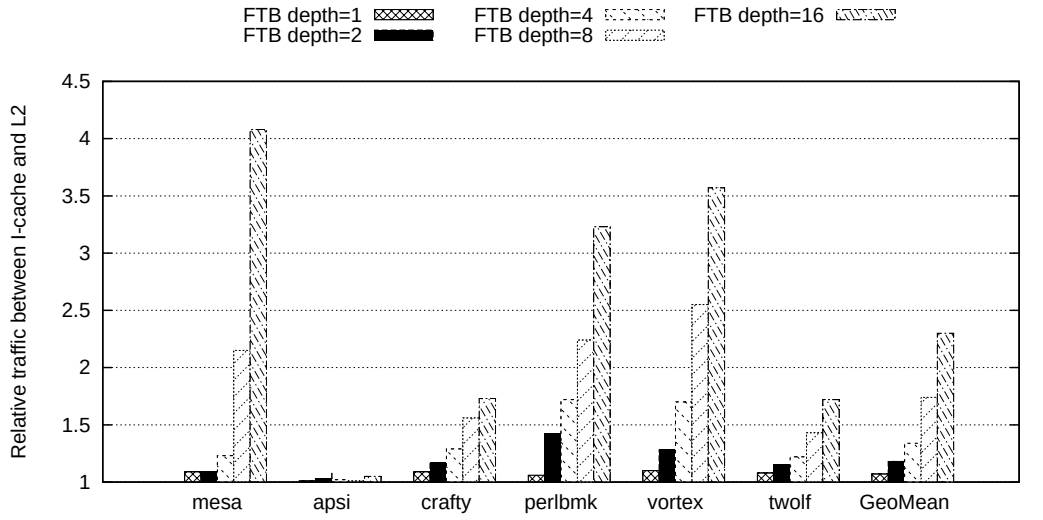


Figure 5.9: Relative traffic between I-cache and L2 when doing look-ahead prefetching with different FTB depths

Application	I-cache efficiency					
	No prefetching	FTB=1	FTB=2	FTB=4	FTB=8	FTB=16
mesa	0.69	0.68	0.68	0.68	0.59	0.45
apsi	0.32	0.32	0.32	0.32	0.32	0.32
crafty	0.40	0.40	0.40	0.38	0.35	0.33
perlbmk	0.57	0.57	0.59	0.55	0.47	0.37
vortex	0.35	0.34	0.33	0.31	0.25	0.19
twolf	0.56	0.55	0.54	0.54	0.48	0.41
GeoMean	0.46	0.46	0.46	0.44	0.39	0.34

Table 5.4: Efficiency of the I-cache when doing look-ahead prefetching with different FTB depths

The drop in I-cache efficiency happens because the look-ahead prefetching scheme usually prefetches a block not long before the block is needed by the program. As a result, even if the prefetch proves useful, the interval between the time when the prefetched block arrives at the instruction cache and the time when the block is fetched is short. According to Equation 1.1, this interval between the arrival of the block and the fetching of the block is a weight in the calculation of the cache efficiency. Therefore, although the prefetch turns out to increase the cache efficiency, the increase is small. On the other hand, if the prefetched block evicts a block that will be referenced later, the decrease in cache efficiency due the the evicted block can be much larger because the evicted block could have been stayed in the cache for a long time.

While look-ahead instruction prefetching does not improve the I-cache efficiency per se, we should keep in mind that these efficiency numbers as defined by Equation 1.1 are not the ultimate goal of instruction prefetching. The ultimate goal is to improve the performance. In that sense, look-ahead instruction prefetching does prove to be effective.

Prefetching Results with Perfect Next-block Prediction

To evaluate the full potential of look-ahead instruction prefetching, we also evaluate it with a perfect next-block predictor.

Figure 5.10 shows the speedups achieved by look-ahead instruction prefetching with a perfect next-block predictor. The baseline processor in Figure 5.10 also assumes a perfect branch predictor but does not perform prefetching. The perfect I-cache configuration also assumes a perfect next-block predictor. The best run-ahead distance is still 4 for most of the six benchmarks. Even with perfect next-block prediction, letting the next-block predictor run too far ahead of the instruction fetch can still cause pollution because the blocks prefetched later can potentially evict the blocks prefetched earlier before they have the chance to be used. Compared with the experiments on a realistic next-block predictor, the performance gap between look-ahead prefetching and a perfect I-cache is much smaller: a perfect I-cache achieves a speedup of 40% on average and the best look-ahead instruction prefetching achieves a speedup of 35% on average.

Figure 5.11 shows the I-cache hit rate of the committed blocks achieved by look-ahead instruction prefetching with a perfect next-block predictor. Unlike the results in Figure 5.6, the I-cache hit rate keeps increasing, or at least does not drop (except for *apsi*), as the run-ahead distance increases.

Figure 5.12 shows the I-cache miss rate of the committed blocks when doing look-ahead instruction prefetching with a perfect next-block predictor. Interestingly, the I-cache miss rate slightly increases as the run-ahead distance increases. The I-cache miss rate increases because some blocks prefetched earlier can be replaced by blocks prefetched later, before they get the opportunity to be used.

Figure 5.12 shows the I-cache MSHR hit rate of the committed blocks when

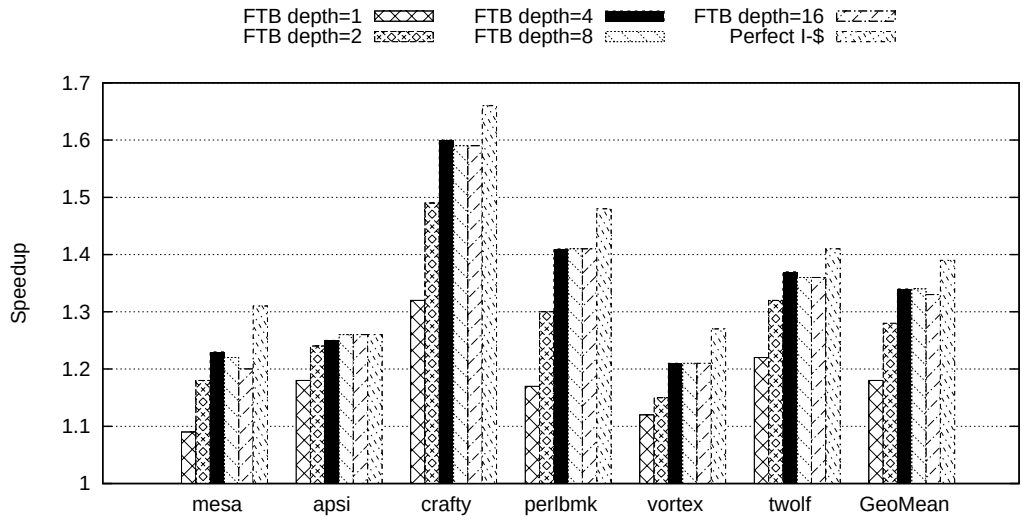


Figure 5.10: Speedups achieved by look-ahead prefetching with different FTB depths and a perfect next-block predictor

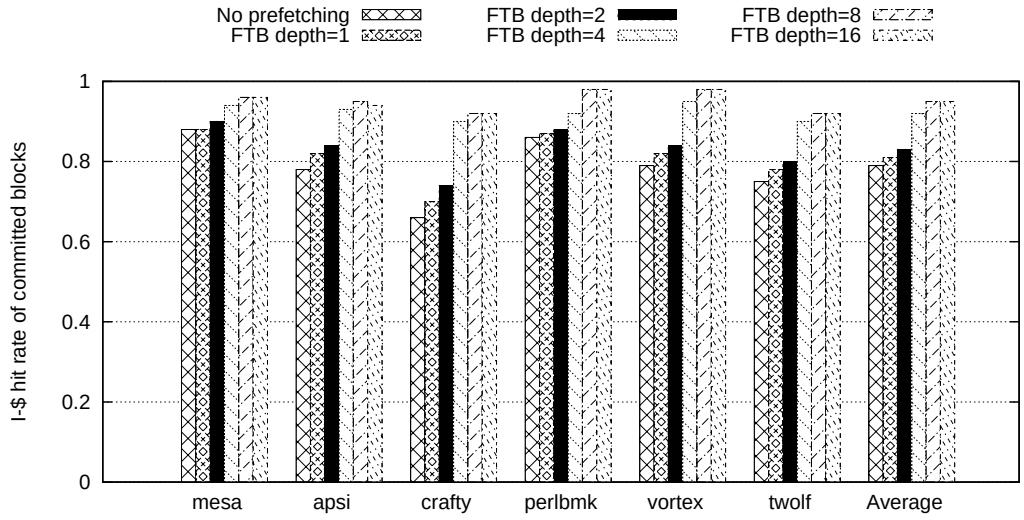


Figure 5.11: I-cache hit rate of committed blocks of look-ahead prefetching with different FTB depths and a perfect next-block predictor

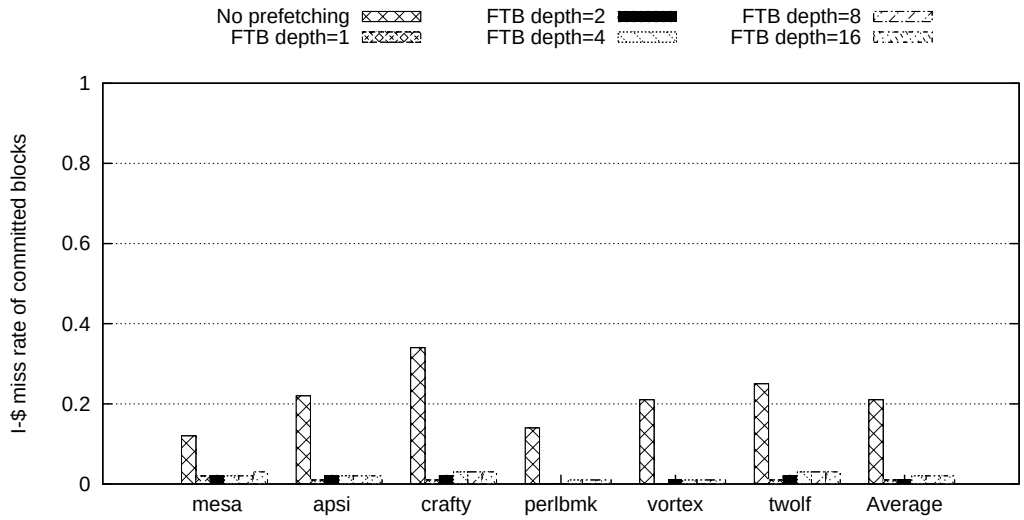


Figure 5.12: I-cache miss rate of committed blocks of look-ahead prefetching with different FTB depths and a perfect next-block predictor

doing look-ahead instruction prefetching with a perfect next-block predictor. As the run-ahead distance increases, the timeliness of the prefetching improves and the MSHR hit rate drops. Furthermore, increasing the run-ahead distance from 2 to 4 significantly improves the timeliness of prefetching. Last, even with a run-ahead distance of 16, some prefetched blocks still do not arrive in time. While further increasing the look-ahead distance may reduce the number of late prefetches, it is unlikely to bring any further performance improvement.

Table 5.5 shows the I-cache efficiency achieved by look-ahead instruction prefetching with a perfect next-block predictor. Even with a perfect next-block predictor, the I-cache efficiency sees little improvement. The difference from Table 5.4 is that here the efficiency does not drop as the run-ahead distance increases.

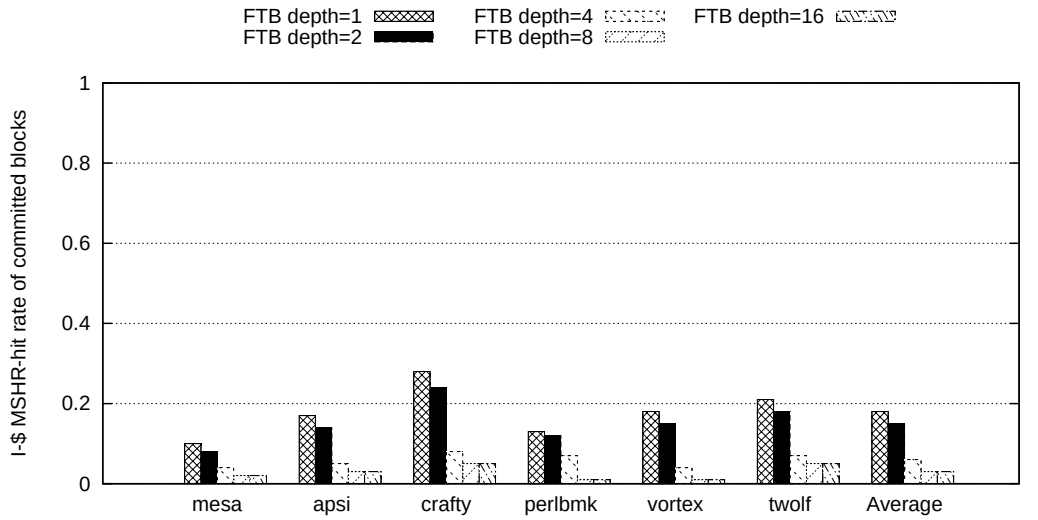


Figure 5.13: I-cache MSHR hit rate of committed blocks of look-ahead prefetching with different FTB depths and a perfect next-block predictor

Application	I-cache efficiency					
	No prefetching	FTB=1	FTB=2	FTB=4	FTB=8	FTB=16
mesa	0.69	0.69	0.69	0.69	0.69	0.69
apsi	0.31	0.31	0.31	0.31	0.31	0.31
crafty	0.40	0.41	0.41	0.42	0.42	0.42
perlbmk	0.62	0.62	0.62	0.62	0.63	0.63
vortex	0.31	0.29	0.27	0.27	0.27	0.27
twolf	0.55	0.56	0.56	0.56	0.57	0.57
GeoMean	0.46	0.45	0.45	0.45	0.45	0.45

Table 5.5: Efficiency of the I-cache when doing look-ahead prefetching with different FTB depths and a perfect next-block predictor

5.4.3 Block Compaction Results

We also evaluate the potential performance benefit of the block compaction technique discussed earlier. Because this technique requires changes to both the ISA and the compiler, we do not simulate this technique in full detail. Instead, we try to estimate its potential benefit by using the default TFlex configuration but with a double-sized instruction cache. The rationale is that the evaluation of the TRIPS prototype [82] shows that on average half of the capacity of the instruction cache in TRIPS is wasted due to NOPs. With block compaction, we can potentially store twice the number of blocks without increasing the I-cache capacity.

Figure 5.14 shows the speedups if the I-cache capacity is doubled. The I-cache hit latency is kept the same as the smaller I-cache because the goal of this study is to approximate the effects of block compaction, which effectively increases the capacity of the I-cache with a smaller cache size. For several benchmarks like *mesa* and *perlbmk*, doubling the I-cache capacity achieves speedups very close to those achieved by a perfect I-cache. Overall, doubling the I-cache capacity proves to be effective for all six benchmarks; the average speedup is approximately 18%, compared to the 30% speedup by a perfect I-cache and the 17% speedup by look-ahead instruction prefetching.

Figure 5.15 shows the I-cache hit rate of the committed blocks with a double-sized I-cache. The hit rate increases across all six benchmarks. In *mesa* and *perlbmk*, the hit rate comes very close to 1, indicating the new I-cache capacity is already close to the size of the working set of these two benchmarks. This also explains why doubling the I-cache capacity achieves speedups close to the speedups of a perfect I-cache for these two benchmarks. On the other hand, for *apsi* and *crafty*, even with a double-sized I-cache, the I-cache miss rate is still relatively high.

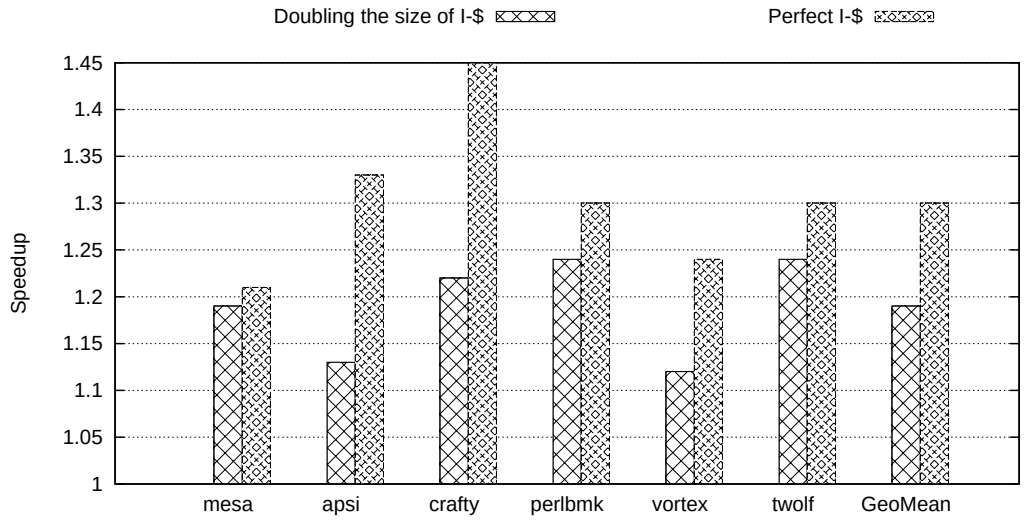


Figure 5.14: Speedups achieved by doubling the size of the instruction cache (10KB per core)

5.4.4 Combining Prefetching with Block Compaction

Until now, we have evaluated instruction prefetching and block compaction separately. In fact, these two techniques compliment each other and can be used together.

Furthermore, the I-cache miss rate for SPEC benchmarks is relatively low. Some larger applications like server applications have very large instruction footprints and can cause much higher I-cache miss rates even for superscalar processors. For these applications, just increasing the I-cache capacity or using block compaction will be insufficient.

Figure 5.16 shows the potential speedup if look-ahead instruction prefetching is used together with block compaction. The results are obtained by simulating look-ahead instruction prefetching with a double-sized I-cache. The baseline is a double-sized I-cache with no prefetching. For the two benchmarks (*apsi*, *crafty*) that still have high I-cache miss rates with a double-sized I-cache, doing look-ahead

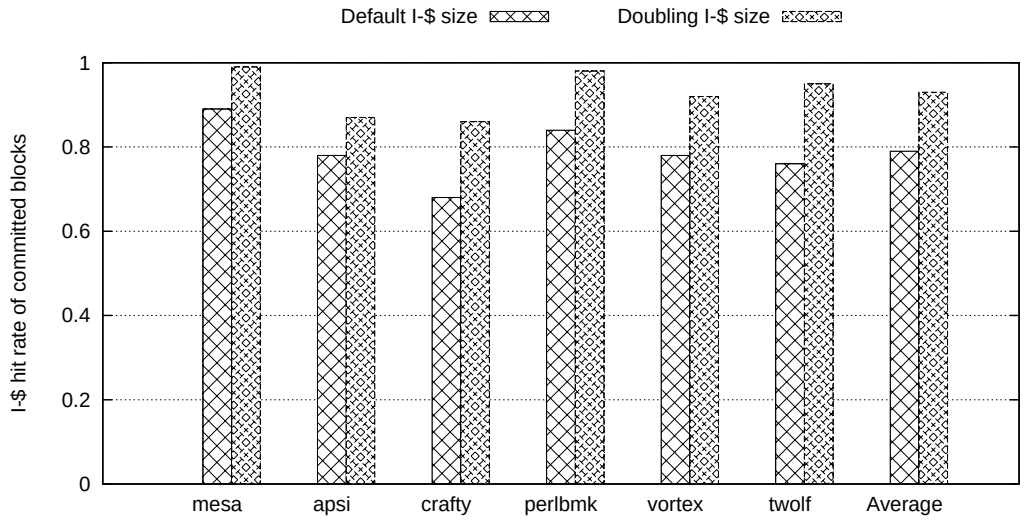


Figure 5.15: I-cache hit rate of committed blocks when doubling the size of the instruction cache (10KB per core)

instruction prefetching brings another 10% speedup approximately.

5.5 Related Work

Instruction prefetching: For superscalar processors, sequential prefetching is fairly straightforward. Most recent studies focus on non-sequential prefetching. To improve the timeliness of non-sequential prefetching, Luk and Mowry proposed to use the compiler to insert software prefetches [56]. Aamodt et al. [2, 3] proposed to perform instruction prefetching by using a helper thread to pre-execute future regions of instructions. Veidenbaum et al. [97] used a multi-level branch predictor [106] to trigger prefetches of instructions several branches away from the current instruction whereas Srinivasan et al. [92] proposed a slightly different scheme to achieve the same goal. Chen et al. [13] proposed Branch Prediction Based Prefetching, which applies the look-ahead data prefetching technique [7] to instruction prefetch-

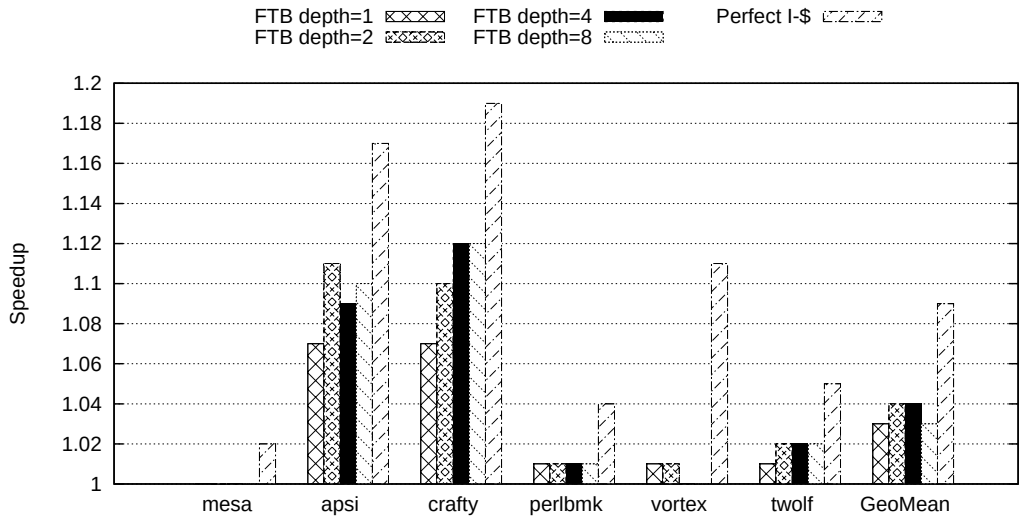


Figure 5.16: Speedups when doubling the size of the instruction cache and doing look-ahead prefetching with different FTB depths

ing. They use a separate branch predictor, which runs ahead of normal instruction fetch, using a LA-PC, to trigger prefetches.

To reduce the pollution caused by instruction prefetching, Xia and Torrellas proposed a scheme where software explicitly marks the end of a sequential sequence so that the prefetch engine does not prefetch beyond the last instruction in the sequence [104]. Luk and Mowry [56] proposed to retain information in the L2 cache about whether a block was used when previously prefetched into the instruction cache. When the prefetch engine is instructed to prefetch these blocks again, the prefetch request is just dropped.

To improve the coverage of instruction prefetching, Pierce and Mudge have proposed Wrong-Path prefetching [68], which prefetches both directions of a branch.

Instruction compression: VLIW processors faced similar code bloat problems that EDGE architectures face. Various schemes have been proposed to reduce the

pressure on the I-cache by instruction compression. Conte et al. [14] proposed a horizontal compression scheme by using Head and Tail bits within an operation to mark the beginning and end of an instruction. Every operation also contains a Pause field to indicate the number of empty instructions following this instruction, thereby accomplishing vertical compression. The Cydra 5 [74], PICO VLIW [5] and the Intel Itanium [95] employ multiple instruction templates to achieve horizontal compression. Zhong et al. [107] proposed to perform instruction compression for a multi-cluster VLIW processor with distributed control by allowing each cluster to have its own PC. More aggressive compression schemes also have been proposed to compress the space occupied by non-NOP instructions [44, 52, 105].

5.6 Summary

In this chapter, we discussed techniques to improve the I-cache efficiency for EDGE architectures. Because of their wide issue widths and large instruction windows, EDGE architectures require sustainable high-bandwidth instruction fetch mechanisms. However, the block-atomic execution model and the distributed microarchitecture pose several challenges to the instruction cache design.

Compared to RISC architectures, programs running on EDGE architectures can experience more I-cache misses for two reasons. First, the code size of EDGE programs are larger than the code size of RISC programs for reasons such as compiler optimizations (loop unrolling, predication), data movement instructions, and NOPs. Second, current microarchitecture instantiations of EDGE architectures allocate fixed-sized space in the instruction cache to all blocks and utilize the instruction cache inefficiently. Furthermore, current microarchitecture instantiations of EDGE architectures also have longer I-cache miss penalty due to the large block size and

the overhead of the distributed protocols.

We proposed two techniques to reduce the I-cache miss rate in EDGE architectures, look-ahead instruction prefetching and variable-sized blocks.

Look-ahead instruction prefetching relies on the next-block predictor, which already exists in the microarchitecture, to run ahead of the instruction fetch and provide predictions about what blocks will be needed by the program in the future. A prefetch engine then prefetches these blocks into the I-cache if necessary. We use the block management mechanism that already exists in the hardware to decouple the next-block predictor from the instruction fetch and control how far the next-block predictor runs ahead of the instruction fetch in the distributed microarchitecture. We evaluate look-ahead instruction prefetching with different run-ahead distances. The results show that a run-ahead distance of 4 provides the best tradeoff between prefetching timeliness and prefetching accuracy. On average, look-ahead instruction prefetching achieves a speedup of 17% on the benchmarks that show high I-cache miss rates, compared to a speedup of 30% by a perfect I-cache.

To reduce the space wasted by storing NOPs in the I-cache, we also discuss how to store variable-sized blocks in the I-cache. The technique we propose, block compaction, compacts several small blocks into the space of one full-sized block so that they can share the space of one block in the I-cache. From the hardware point of view, the blocks in the I-cache still look like fixed-sized blocks so this scheme has low hardware complexity. It does require changes to the ISA and the compiler, though. For this reason, we do not simulate this scheme in full detail but only estimates its potential benefit by simulating a fixed-sized block design with double-sized I-cache. On average, block compaction can potentially bring a speedup of 18% on the benchmarks that have high I-cache miss rates.

Look-ahead instruction prefetching and variable-sized blocks are two techniques that are complimentary to each other and could be used together. For benchmarks that have a large code footprint, combining these two techniques brings further performance improvements.

Chapter 6

Conclusions

Because of the large speed gap between the microprocessor core and the main memory, caches are widely used in today's microprocessors. As the microprocessor core becomes more powerful and the number of cores increases, larger caches are needed to provide enough data and bandwidth in order to keep the cores well utilized. Prior studies [10, 94] suggest that only a small portion of the caches are utilized to store information that will be referenced again, indicating that there exists great potential to improve the utilization of the cache capacity by storing more useful information in the cache.

In this dissertation, we studied hardware techniques to improve the data cache efficiency in general and the instruction cache efficiency in EDGE architectures. Because the differences in the access behavior of these two kinds of caches, we explored different techniques in these two studies.

6.1 Summary

To increase cache efficiency, the hardware must store more live information in the cache without increasing the capacity of the cache. We use different strategies to improve the efficiency of data caches in general and instruction caches in EDGE architectures.

For the instruction cache, the branch predictor provides quite accurate information about what future instructions will be needed by the program and can therefore be used to bring useful instructions into the I-cache. We explore how to prefetch future instructions needed by the program by letting the branch predictor run ahead of the instruction fetch in a distributed microarchitecture instantiation of an EDGE architecture. We also consider techniques that are specifically targeted at EDGE architectures to reduce the space needed to store instructions in the instruction cache.

For the data cache and the L2 cache, we investigate how to identify dead blocks in the cache with higher accuracy and coverage and how to use the dead-block information for better cache replacement policies, bypassing zero-reuse blocks, and more effective prefetching.

6.1.1 Improving Data Cache Efficiency

In the first half of this dissertation, we explored techniques to improve data cache efficiency by identifying and eliminating dead blocks early.

Identifying dead blocks early is achieved by dead-block prediction. The three metrics for dead-block prediction are prediction accuracy, coverage, and timeliness. However, it is not possible to achieve the best prediction accuracy, coverage, and timeliness at the same time. Dead-block predictors proposed by prior work make

predictions immediately after a block is referenced. While this approach identifies dead blocks as early as possible, it sacrifices prediction accuracy and coverage because a block just accessed may be accessed again soon. There is a tradeoff between the timeliness and accuracy/coverage of dead-block prediction. The earlier the prediction is made, the more useful it is. On the other hand, the later the prediction is made, the less likely it is to mispredict. We quantified the tradeoff between prediction timeliness, accuracy, and coverage and showed that delaying prediction until a block just moves out of the MRU position gives the best tradeoff among the three metrics.

Accesses to L1 and L2 caches have different characteristics and these differences should considered when designing dead-block predictors for different cache levels.

Accesses to L1 caches are bursty with abundant intra-block locality and can be easily affected by data and control flow dependences and data alignment variations. To tolerate the irregularity in the individual references to a block in the L1 cache, we propose the concept of cache bursts. A cache burst combines the contiguous group of references a block received while in the MRU position of its cache set into one entity and can thus hide the irregularity of individual references caused by data and control dependences. Dead-block predictors at the L1 cache should make predictions based on cache bursts, not individual references, because cache bursts are more predictable than individual references. Cache bursts can be applied to counting-based, trace-based, or time-based predictors. In this dissertation, we evaluate a burst counting predictor and a burst trace predictor. Compared to reference-based predictors, the new burst-based predictors can correctly identify more dead blocks while making fewer predictions. The best burst-based predictor,

BurstTrace, can identify 96% of the dead blocks in a 64KB, 2-way set-associative L1 D-cache with a 96% accuracy. For a 64KB, four-way L1 cache, the prediction accuracy and coverage are 92% and 91% respectively. At any moment, the average fraction of the dead blocks that has been correctly detected for a two-way or four-way L1 cache is approximately 49% or 67% respectively. Besides the better prediction accuracy and coverage, burst-based predictors also have lower power consumption because they update the burst count/trace and access the history table less frequently.

Accesses to the L2 cache are filtered by the L1 cache, have little intra-block locality, and are less affected by data and control flow dependences. Because of the loss of information due to the filtering by the L1, dead-block predictors for the L2 cache should be counting-based predictors that keep track of the individual references. To cope with reference-count variations, we proposed several optimizations to an existing counting-based predictor. These optimizations increase the prediction accuracy by maintaining more up-to-date history information and increase the prediction coverage by filtering out the sporadic smaller reference counts. The improved predictor can identify 66% of the dead blocks in a 1MB, 16-way set-associative L2 cache with a 89% accuracy. At any moment, 63% of the dead blocks in such an L2 cache, on average, has been correctly identified by the dead-block predictor

Dead-block prediction by itself does not increase cache efficiency. To get better cache efficiency and higher performance, the dead blocks identified must be eliminated from the cache early. We evaluate three techniques to eliminate dead blocks early: replacement optimization, cache bypassing, and prefetching into dead blocks. Replacement optimization evicts blocks that become dead after several reuses, before they reach the LRU position. Cache bypassing identifies blocks that

cause cache misses but will not be reused if they are written into the cache and do not store these blocks in the cache. Prefetching into dead blocks replaces dead blocks with prefetched blocks which are likely to be referenced in the future.

All three approaches try to eliminate dead blocks early but differ in when and how dead blocks are eliminated. Both replacement optimization and bypassing eliminate dead blocks only on demand misses; replacement optimization evicts dead blocks already in the cache while bypassing evicts dead blocks causing the misses. Both can miss opportunities by leaving dead blocks in the cache. Prefetching into dead blocks aims to eliminate dead blocks whenever they are identified. As a result, it is able to reduce more cache misses and achieve greater performance improvement. On average, replacement optimization or bypassing improves performance by 5% while prefetching into dead blocks brings a 12% performance improvement over the baseline prefetching scheme for the L1 cache and a 13% performance improvement over the baseline prefetching scheme for the L2 cache.

6.1.2 Improving Instruction Cache Efficiency in EDGE Architectures

In the second half of this dissertation, we explored techniques to improve the instruction cache efficiency in EDGE architectures.

EDGE architectures feature a block-atomic execution model and direct communication among instructions within a block. To achieve high ILP, microarchitectures implementing EDGE ISAs have a large instruction window and a high instruction issue width. To be scalable, these microarchitectures are distributed and avoid large, centralized structures. These features all require a new instruction cache design that can sustain a high instruction fetch bandwidth.

We presented such an I-cache design for the TRIPS prototype chip, the first instantiation of an EDGE architecture. The instruction cache in the TRIPS prototype is distributed into five tiles and uses distributed protocols to communicate with each other and other components of the chip. We discussed how the TRIPS blocks are stored in the instruction cache. Because of the large block size and the distributed structure, fetching a block takes multiple cycles and instruction cache misses are expensive. As a result, the branch predictor usually has to spend time waiting for the instruction fetch to catch up.

Experiences with the TRIPS prototype reveal some limitations of the TRIPS microarchitecture. These limitations are addressed in TFlex, a new microarchitecture that shares the same ISA as TRIPS. One lesson we learned from the TRIPS prototype is that EDGE architectures can experience more instruction cache misses than RISC architectures do. This happens for two reasons. First, the code size of EDGE programs are larger than the code size of RISC programs. Second, EDGE architectures have a longer I-cache miss penalty due the large block size and the overhead of the distributed protocols.

We proposed two techniques to reduce the I-cache miss rate in EDGE architectures, look-ahead instruction prefetching and variable-sized blocks.

Look-ahead instruction prefetching relies on the next-block predictor, which already exists in the microarchitecture, to run ahead of the instruction fetch and provide predictions about what blocks will be needed by the program in the future. A prefetch engine then prefetches these blocks into the I-cache if necessary. We use the block management mechanism that already exists in the hardware to decouple the next-block predictor from the instruction fetch and control how far the next-block predictor runs ahead of the instruction fetch in the distributed microarchitecture. A

key parameter in look-ahead instruction prefetching is the run-ahead distance. Our simulation results show that a run-ahead distance of 4 provides the best tradeoff between prefetching timeliness and prefetching accuracy. On average, look-ahead instruction prefetching achieves a speedup of 17% on the benchmarks that show high I-cache miss rates, compared to a speedup of 30% by a perfect I-cache.

To reduce the space wasted by storing NOPs in the I-cache, we also discussed how to store variable-sized blocks in the I-cache. The technique we propose, block compaction, compacts several small blocks into the space of one full-sized block so that they can share the space of one block in the I-cache. This scheme has low hardware complexity because the instruction cache is still manged in the unit of fixed-sized blocks. It does require changes to the ISA and the compiler, though. For this reason, we do not simulate this scheme in full detail but only estimates its potential benefit by simulating a fixed-sized block design with double-sized I-cache. On average, block compaction can potentially bring a speedup of 18% on the benchmarks that have high I-cache miss rates.

Look-ahead instruction prefetching and variable-sized blocks are two techniques that are complimentary to each other and could be used together. For benchmarks that have a large code footprint, combining these two techniques brings further performance improvements.

6.2 Further Improving Cache Efficiency

While the techniques discussed in this dissertation are able to increase the data cache efficiency and improve the performance of both data caches and instruction caches in EDGE architecture, there is still great potential to improve the cache efficiency even further. To further improve cache efficiency, the following approaches are promising.

Improving data cache efficiency: All the dead-block predictors discussed so far have a common limitation: their future actions are affected not only by the memory references generated by the program, but also by the actions, such as replacement, bypassing, or prefetching, taken based on the predictions made by these dead-block predictors. In other words, these dead-block predictors are not perturbation resistant. Such predictors can be problematic because a wrong prediction made by the predictor can cause it to continue to make wrong predictions in the future. In theory, this can cause the dead-block predictor to always make wrong predictions. In practice, most dead-block predictors use some sort of heuristics such as saturating counters to reduce the probability of this pathological case. Better dead-block prediction accuracy and coverage can be achieved if we can find a dead-block predictor that is perturbation resistant, which means that the predictions of the predictor are only affected by the memory reference stream generated by the program.

The dead-block predictors discussed in this dissertation can be considered fine-grained predictors because they keep information about each individual block. Another approach is to keep information at a coarser granularity, for example, the whole cache. A coarse-grained predictor does not need to know the exact blocks that are dead; instead, it only predicts what fraction of the blocks are dead. This information can then be used to disable some portion of the cache. Such techniques can be more effective in some cases because sometimes there is little difference between the hit rate of a larger cache and the hit rate of a smaller cache. A coarse-grained dead-block predictor obviously has very low overhead and can be complimentary to the fine-grained predictors.

Because of the fundamental tradeoff between dead-block prediction timeliness and accuracy, hardware predictors by themselves may be insufficient. For best dead-

block prediction results, hybrid predictors should be considered. A hybrid predictor has a hardware predictor and also provides mechanisms for the software to pass hints about the live and dead status of cache blocks. When making a prediction, the predictor uses the hints from the software if they are available. Otherwise, the hardware predictor makes a prediction.

This dissertation proposed new approaches to identify the dead space in the data cache and L2 cache. These new approaches are able to identify a larger portion of the dead space with higher accuracy. However, the effectiveness of these new dead-block prediction schemes are constrained by how the identified dead space can be effectively utilized. Existing techniques such as replacement optimizations, cache bypassing, and prefetching, do not seem to be able to fully utilize the dead space identified. To better utilize the identified dead space, more effective techniques are needed. The dead-block prediction schemes proposed here could be more successful if such techniques are found.

Improving instruction cache efficiency: One limitation of the look-ahead instruction prefetching scheme is that useful blocks can be evicted from the cache by prefetched blocks or demand misses. For example, when the next-block predictor runs several blocks ahead of the instruction fetch, a block prefetched earlier may get evicted before it has the chance to be used by the program. Another example is that although a block may reside in the cache when the prefetch engine probes its existence, it may get evicted later before actually get used. To address this problem, the cache can mark some blocks as live, which is the opposite of dead-block prediction. The live blocks are very likely to be used by the program soon and should not be evicted from the cache before getting used.

The block compaction technique can be considered as a simple instruction

cache compression scheme with low hardware complexity. More aggressive compression schemes could be used to reduce the amount of space occupied by the blocks in the instruction cache. However, as the compression ratio goes up, the complexity of the instruction cache also goes up and either increase the cycle time or the number of stages in the pipeline. Instruction cache compression has been used on some VLIW processors, which faced similar code bloat problems that EDGE architectures face. More research is needed to explore whether more aggressive compression schemes would be beneficial for EDGE architectures.

Bibliography

[1] Itanium Tukwila processor. http://en.wikipedia.org/wiki/Tukwila_(processor).

[2] T. M. Aamodt, P. Chow, P. Hammarlund, H. Wang, and J. P. Shen. Hardware support for prescient instruction prefetch. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 84, Washington, DC, USA, 2004. IEEE Computer Society.

[3] T. M. Aamodt, P. Marcuello, P. Chow, A. González, P. Hammarlund, H. Wang, and J. P. Shen. A framework for modeling and optimization of prescient instruction prefetch. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 13–24, New York, NY, USA, 2003. ACM.

[4] J. Abella, A. Gonzàlez, X. Vera, and M. F. P. O'Boyle. IATAC: a smart predictor to turn-off L2 cache lines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(1):55–77, March 2005.

[5] S. Aditya, S. A. Mahlke, and B. R. Rau. Code size minimization and retargetable assembly for custom epic and VLIW instruction formats. *ACM Transactions on Design Automation and Electronic Systems*, 5(4):752–773, 2000.

[6] J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th Annual Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

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

[8] M. Bhadauria, S. A. McKee, K. Singh, and G. Tyson. A precisely tunable drowsy cache management mechanism. In *Proceedings of Watson Conference on Interaction between Architecture, Circuits, and Compilers*, 2006.

[9] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4), 1999.

[10] D. Burger, J. R. Goodman, and A. Kägi. The declining effectiveness of dynamic caching for general-purpose microprocessors. Technical Report 1261, Computer Sciences Department, University of Wisconsin, Madison, 1995.

[11] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, and W. Yoder. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[12] 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.

[13] I.-C. Chen, C.-C. Lee, and T. N. Mudge. Instruction prefetching using branch prediction information. In *Proceedings of the 1997 International Conference on Computer Design*, page 593, September 1997.

[14] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye. Instruction fetch mechanisms for VLIW architectures with compressed encodings. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 201–211, Washington, DC, USA, 1996. IEEE Computer Society.

[15] K. E. Coons, X. Chen, S. K. Kushwaha, D. Burger, and K. S. McKinley. A spatial path scheduling algorithm for EDGE architectures. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–140, October 2006.

[16] R. Desikan, D. C. Burger, S. W. Keckler, and T. M. Austin. Sim-alpha: a validated, execution-driven alpha 21264 simulator. Technical Report TR-01-23, Computer Sciences Department, University of Texas at Austin, 2001.

[17] M. Ferdman and B. Falsafi. Last-touch correlated data streaming. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 105–115, 2007.

[18] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 148–157, May 2002.

[19] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley. An evaluation of the TRIPS computer system. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2009.

[20] A. González, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proceedings of the 9th International Conference on Supercomputing*, 1995.

[21] P. Gratz, C. Kim, R. McDonald, S. W. Keckler, and D. Burger. Implementation and Evaluation of On-Chip Network Architectures. In *Proceedings of the 2006 IEEE International Conference on Computer Design*, October 2006.

[22] P. Gratz, K. Sankaralingam, H. Hanson, P. Shivakumar, R. McDonald, S. W. Keckler, and D. Burger. Implementation and Evaluation of Dynamically Routed Processor Operand Network. In *To Appear in the 1st ACM/IEEE International Symposium on Networks-on-Chip*, May 2007.

[23] E. Hao, P.-Y. Chang, M. Evers, and Y. N. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 191–200, Washington, DC, USA, 1996. IEEE Computer Society.

[24] A. Hartstein and T. R. Puzak. The optimium pipeline depth for a microprocessor. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 7–13, May 2002.

[25] A. Hartstein and T. R. Puzak. Optimum power/performance pipeline depth. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 117–128, 2003.

[26] R. Ho, K. W. Mai, and M. A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[27] M. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and

P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 14–24, May 2002.

[28] W.-C. Hsu and J. E. Smith. A performance study of instruction cache prefetching methods. *IEEE Transactions on Computers*, 47(5):497–508, 1998.

[29] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

[30] Z. Hu, M. Martonosi, and S. Kaxiras. TCP: Tag correlating prefetchers. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, 2003.

[31] J. Huh. Hardware techniques to reduce communication costs in multiprocessors, May 2006. Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin.

[32] J. Jalminger and P. P. Stenström. A novel approach to cache block reuse prediction. In *Proceedings of the 2003 International Conference on Parallel Processing*, 2003.

[33] T. L. Johnson, D. A. Connors, M. C. Merten, and W. W. Hwu. Run-time cache bypassing. *IEEE Transactions on Computers*, 48(12):1338–1354, 1999.

[34] T. L. Johnson and W. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[35] N. P. 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*, 1990.

[36] M. Kampe, P. Stenström, and M. Dubois. Self-correcting LRU replacement policies. In *Proceedings of the 1st conference on Computing frontiers*, 2004.

[37] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th International Symposium on Computer Architecture*, 2001.

[38] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[39] M. Kharbutli and Y. Solihin. Counter-based cache replacement algorithms. In *Proceedings of the International Conference on Computer Design*, October 2005.

[40] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4):433–447, 2008.

[41] C. Kim. A technology-scalable composable architecture, August 2007. Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin.

[42] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, October 2002.

[43] C. Kim, S. Sethumadhavan, M. S. Govindan, N. R. ganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 381–394, Chicago, Illinois, USA, 2007. IEEE Computer Society.

[44] M. Kozuch and A. Wolfe. Compression of embedded system programs. In *Proceedings of the 1994 IEEE International Conference on Computer Design*, pages 270–277, Washington, DC, USA, 1994. IEEE Computer Society.

[45] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 81–87, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[46] D. Kroft. Retrospective: lockup-free instruction fetch/prefetch cache organization. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 20–21, New York, NY, USA, 1998. ACM.

[47] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 139–148, 2000.

[48] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 144–154, 2001.

[49] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48–59, 1995.

[50] C.-C. Lee and T. N. Mudge. Instruction prefetching using branch prediction

information. In *ICCD '97: Proceedings of the 1997 International Conference on Computer Design (ICCD '97)*, page 593, Washington, DC, USA, 1997. IEEE Computer Society.

[51] K.-F. Lee, H.-W. Hon, and R. Reddy. An overview of the sphinx speech recognition system. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 38(1):35–44, 1990.

[52] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Instruction selection using binate covering for code size optimization. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 393–399, Washington, DC, USA, 1995. IEEE Computer Society.

[53] W.-F. Lin and S. K. Reinhardt. Predicting last-touch references under optimal replacement. Technical Report CSE-TR-447-02, Department of Electrical Engineering and Computer Science, University of Michigan, 2002.

[54] W.-F. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *Proceedings of the 7th International Symposium on High Performance Architecture*, 2001.

[55] J. S. Liptay. Structural aspects of the system/360 model 85 II: The cache. *IBM Systems Journal*, 7(1):15–21, 1968.

[56] C. Luk and T. Mowry. Architectural and compiler support for effective instruction prefetching: a cooperative approach. *ACM Transactions on Computer Systems*, 19(1):71–109, 2001.

[57] B. A. Maher, A. Smith, D. Burger, and K. S. McKinley. Merging head and tail duplication for convergent hyperblock formation. In *Proceedings of the 39th*

Annual IEEE/ACM International Symposium on Microarchitecture, pages 65–76, December 2006.

[58] D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, September 1997.

[59] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995.

[60] R. McDonald, D. Burger, S. W. Keckler, K. Sankaralingam, and R. Nagarajan. TRIPS processor reference manual. Technical Report TR-05-19, Department of Computer Sciences, The University of Texas at Austin, March 2005.

[61] S. McFarling. Cache replacement with dynamic exclusion. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[62] S. A. McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, page 162, New York, NY, USA, 2004. ACM.

[63] K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[64] A. Mendelson, D. Thiébaut, and D. Pradhan. Modeling live and dead lines in cache memory systems. Technical Report TR-90-CSE-14, Department of Electrical and Computer Engineering, University of Massachusetts, 1990.

[65] T. N. Mudge. Power: A first-class architectural design constraint. *IEEE Computer*, 34(4):52–58, 2001.

[66] R. Nagarajan. Design and evaluation of a technology-scalable architecture for instruction-level parallelism, May 2007. Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin.

[67] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of GRID processor architectures. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 40–51, December 2001.

[68] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 165–175, Washington, DC, USA, 1996. IEEE Computer Society.

[69] P. Pujara and A. Aggarwal. Increasing the cache efficiency by eliminating noise. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, 2006.

[70] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr, and J. Emer. Adaptive insertion policies for high-performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[71] M. K. Qureshi, M. A. Suleman, and Y. N. Patt. Line distillation: Increasing cache capacity by filtering unused words in cache lines. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, February 2007.

[72] R. Rabbah, I. Bratt, K. Asanovic, and A. Agarwal. Versatility and VersaBench: A new metric and a benchmark suite for flexible architectures. Technical Report MIT-LCS-TM-646, MIT Techincal Memo, June 2004.

[73] N. Ranganathan. Control flow speculation for distributed architectures, Ph.D proposal, April 2007.

[74] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towie. The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *Computer*, 22(1):12–26, 28–30, 32–35, 1989.

[75] G. Reinman, T. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 234–245, May 1999.

[76] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *Proceedings of the 32nd International Symposium on Microarchitecture*, pages 16–27, November 1999.

[77] G. Reinman, B. Calder, and T. Austin. Optimizations enabled by a decoupled front-end architecture. *IEEE Transactions on Computers*, 50(4):338–355, 2001.

[78] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing resue information in data cache management. In *Proceedings of the 12th International Conference on Supercomputing*, 1998.

[79] B. Robatmili, K. Coons, D. Burger, and K. McKinley. Strategies for mapping dataflow blocks to distributed hardware. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, November 2008.

[80] B. Robatmili, K. E. Coons, D. Burger, and K. S. McKinley. Strategies for mapping dataflow blocks to distributed hardware. In *Proceedings of the 41st International Symposium on Microarchitecture*, November 2008.

[81] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[82] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *Proceedings of the 39th Internationtal Symposium on Microarchitecture*, November 2006.

[83] The national technology roadmap for semiconductors. Semiconductor Industry Association, 2001.

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

[85] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE architectures. In *Fourth International ACM/IEEE Symposium on Code Generation and Optimization (CGO)*, pages 185–195, March 2006.

[86] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Dataflow predication. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 89–100, December 2006.

[87] A. J. Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, 1978.

[88] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[89] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 588–597, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[90] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Memory coherence activity prediction in commercial workloads. In *3rd Workshop on Memory Performance Issues*, 2004.

[91] L. Spracklen, Y. Chou, and S. G. Abraham. Effective instruction prefetching in chip multiprocessors for modern commercial applications. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 225–236, 2005.

[92] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak. Branch history guided instruction prefetching. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 291, September 2001.

[93] Standard Performance Evaluation Corporation. *SPEC Newsletter*, Fairfax, VA, September 2000.

[94] P. Stenström. Chip-multiprocessing and beyond. In *Keynote at 12th International Symposium on High-Performance Computer Architecture*, 2006.

[95] W. Triebel. *Itanium Architecture for Software Developers*. Intel Press, 2000.

[96] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995.

[97] A. V. Veidenbaum, Q. Zhao, and A. Shameer. Non-sequential instruction cache prefetching for multiple–issue processors. *International Journal of High-Speed Computing*, 10(1):115–140, 1999.

[98] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2002.

[99] M. Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, 14(2):270–271, 1965.

[100] W. A. Wong and J.-L. Baer. Modified LRU policies for improving second-level cache behavior. In *Proceedings of 6th International Symposium on High Performance Computer Architecture*, 2000.

[101] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[102] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1991.

[103] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[104] C. Xia and J. Torrellas. Instruction prefetching of systems codes with layout optimized for reduced cache misses. In *Proceedings of the 23rd annual international symposium on Computer architecture*, pages 271–282, New York, NY, USA, 1996. ACM.

[105] Y. Xie, W. Wolf, and H. Lekatsas. A code decompression architecture for VLIW processors. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 66–75, Washington, DC, USA, 2001. IEEE Computer Society.

[106] T.-Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *Proceedings of the 7th international conference on Supercomputing*, pages 67–76, New York, NY, USA, 1993. ACM.

[107] H. Zhong, K. Fan, S. Mahlke, and M. Schlansker. A distributed control path architecture for VLIW processors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 197–206, September 2005.

[108] A. Zmily, C. Kozyrakis, and E. Killian. Improving instruction delivery with a block-aware ISA. In *Proceedings of EuroPar'05 Conference*, September 2005.

Vita

Haiming Liu was born in Qingdao, China in 1974. He finished high school in Qingdao No. 3 Middle School and entered the University of Science and Technology of China for college education in 1993. He received a Bachelor of Science degree in Computer Sciences from the University of Science and Technology of China in July 1998. He then entered the Institute of Computing Technology of the Chinese Academy of Sciences and received a Masters degree in Computer Sciences in July 2001. In the fall of 2001, he joined the doctoral program at the Department of Computer Sciences at the University of Texas at Austin.

Permanent Address: 31 Yancheng Rd Apt 1-201,

Qingdao, Shandong, China 266071

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.