USING GENERAL-PURPOSE PROCESSOR CORES AS PREFETCHING ENGINES IN CHIP MULTIPROCESSOR ARCHITECTURES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Ilya Konstantinovich Ganusov

August 2007

USING GENERAL-PURPOSE PROCESSOR CORES AS PREFETCHING ENGINES IN CHIP MULTIPROCESSOR ARCHITECTURES

Ilya Konstantinovich Ganusov, Ph.D.

Cornell University 2007

Scaling the performance of applications with little thread-level parallelism is one of the most serious impediments to the success of multi-core architectures. At the same time, the long latency of memory accesses represents one of the largest performance bottlenecks for individual program threads. As a result, a typical microprocessor spends a significant amount of time waiting for data to be delivered from memory instead of performing useful computation.

Fortunately, it is often possible to guess which memory data will be needed by a program thread in the near future. Various hardware and software prefetching techniques have been developed to fetch critical data before they are requested by the processor. This way prefetching can eliminate processor stalls otherwise induced by the slow response from the memory system.

The main contribution of this dissertation is the development of two techniques that utilize extra cores of a chip multiprocessor (CMP) as prefetching engines to increase the performance of single program threads. The proposed approaches effectively leverage the execution capabilities of chip multiprocessors to compute data addresses that are likely to miss in the cache and prefetch them ahead of program thread load requests.

I demonstrate the effectiveness of the proposed approaches by performing cycle-accurate simulations of a chip multiprocessor consisting of two four-way superscalar cores running the single-threaded SPEC CPU2000 benchmark suite. The proposed mechanisms provide significant performance improvements over a baseline that already includes an aggressive hardware stream prefetcher. A comparison with other multi-core prefetching mechanisms from the literature shows that the techniques proposed in this dissertation provide competitive performance, incur less energy overhead, and require considerably simpler hardware support.

BIOGRAPHICAL SKETCH

The author graduated from High School #1, Ivanovo, Russian Federation, with the Golden Medal for outstanding scholastic achievements in 1996. He enrolled at the Ivanovo State Power University in 1996, pursuing a double major in Electrical Engineering and English, and graduated with Electronics Engineer and Professional Translator degrees with Honors in 2001. Since mid 2002, the author has been a graduate student in the Computer Systems Laboratory, which is a part of the School of Electrical and Computer Engineering of Cornell University, working under the guidance of Prof. Martin Burtscher.

Dedicated to my parents

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Prof. Martin Burtscher, for his continuous guidance and enthusiastic support throughout the lifetime of this project. Thanks to my committee members, Profs. Rajit Manohar and Jose Martinez, for encouraging discussions and insightful comments.

I am grateful to all my friends and fellow computer engineers from the Computer Systems Laboratory for their help, great discussions, and a sense of humor. Special thanks to David Fang, Filipp Akopyan, Karan Singh, Vince Weaver, and Virantha Ekanayake for their assistance with all types of technical problems and challenges - at all times.

I thank Maya Haridasan, Filipp Akopyan, Nosheen Ali, Jui Bhagwat, David Fang, Carlos Tadeo Ortega Otero, Christina Peraki, Paula Petrica, Zoya Svitkina, Basit Riaz Sheikh, and Jonathan Winter for their great friendship, selfless support, and many exciting conversations.

Finally, I am forever indebted to my parents for their understanding, endless patience and encouragement when it was most needed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This chapter describes how scaling the performance of single-threaded applications represents a serious impediment to the success of chip multiprocessor (CMP) architectures. It also discusses how the slow execution speed (the latency) of load instructions can impact the performance of individual program threads and introduces data prefetching, a technique to alleviate the load latency problem. Furthermore, the contributions of this dissertation to the area of multi-core prefetching are presented.

1.1 Problem

Chip multiprocessors hold the promise of delivering performance scalability while significantly reducing the complexity relative to monolithic superscalar out-of-order cores. In fact, all major high-performance microprocessor manufacturers are already selling chips with up to eight cores. CMPs improve the performance by integrating many cores on the same die and using them to harness thread-level parallelism (TLP). At the same time, designing simple cores and then replicating them reduces complexity.

While multiple cores are immediately beneficial in multiprogrammed environments, scaling the performance of applications with little TLP is one of the most serious impediments to the success of CMP architectures. Modern automatic parallelization tools are efficient only on a small subset of data-regular applications. Manual parallelization to extract significant TLP from general-purpose applications is often a difficult, time-consuming, and therefore expensive and error prone task. Moreover, many programs exhibit limited scalability beyond a certain num-

ber of threads and are inherently incapable of utilizing the advantages of multiple cores. In light of these trends, architectural techniques that allow the use of additional cores to speed up individual threads are becoming very attractive to CMP manufacturers [38].

This dissertation investigates *simple* yet *efficient* ways to realize this goal. One way to achieve this objective is to utilize additional cores as helper engines for individual threads. These helper engines can accelerate program execution by alleviating some of the performance bottlenecks that prevent the program thread from fully utilizing a core's capabilities.For example, helper engines can reduce the number of branch mispredictions or eliminate stalls associated with cache misses. The following section demonstarted that eliminating main memory stalls has high performance improvement potential and provides a very attractive problem for helper engines to attack.

1.2 Main Memory Stalls

Over the past two decades the CPU speed has been growing faster than the speed of the memory subsystem. As a consequence, memory accesses have, in relative terms, become slower over the years. The long latency of memory accesses causes modern high-end microprocessors to deliver only a fraction of their theoretical peak performance. Whenever the processor core requests data that is located in the main memory, it has to stall for many clock cycles while the request is delivered to the memory subsystem and the data is located in the main memory and transferred back to the processor.

To reduce the access time of frequently used data, most modern microprocessor systems incorporate multiple levels of fast cache memory. The first level caches

are characterized by the smallest sizes and the fastest access times. These caches store the most recent data in case it is needed again. Subsequent cache levels are generally bigger and slower. The main memory is at the end of the memory hierarchy and has the longest access time. When a load instruction is executed, the first-level cache is queried for the desired data item. If the data item is available in the first-level cache, it is fetched to the processor and the load request is satisfied very quickly. If not, the load request gets forwarded to subsequent cache levels until the requested data is found. If the data cannot be found in any cache, the data has to be retrieved from main memory. Hence, the time it takes to execute a load instruction depends on the cache level that satisfies the load request and can vary from a few clock cycles to hundreds of cycles. In comparison, reading data values from the CPU's register file rarely takes more than a single cycle.

While caching is effective at reducing the average load latency, a wide range of applications have relatively high cache miss rates, i.e., a large fraction of load requests that is not satisfied at any level of cache hierarchy. Since load instructions

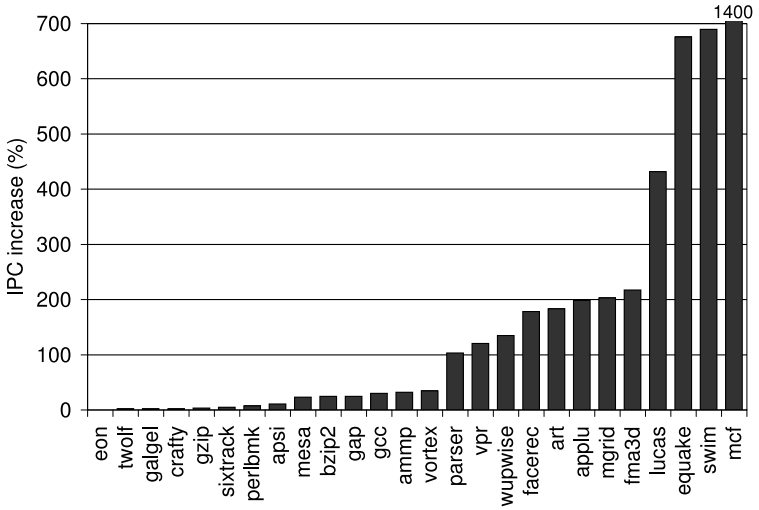

Figure 1.1: Execution time speedup for SPEC CPU2000 programs if all main memory stalls are eliminated

are the most frequently executed instructions in modern CPUs, even a small number of loads that miss in the cache can severely degrade processor performance. Figure 1.1 demonstrates the theoretical speedup of SPEC CPU2000 programs running on an Alpha 21264-like microprocessor core with 64KB first-level cache, 2MB second-level cache, and a main memory latency of 400 cycles when the negative effects of all second-level cache misses are eliminated (i.e., assuming all first-level cache misses are second-level cache hits). The data reveal that for half of the SPEC programs the high latency of memory accesses reduces the core efficiency by more than a factor of two even in the presence of a cache hierarchy with a significant capacity.

1.3 Data Prefetching

Fortunately, load instructions often reference predictable sequences of data addresses. For example, a large fraction of the load instructions that miss in the cache occur in loops that sequentially traverse the elements of large data arrays [23]. The data addresses referenced by such loads form an easily predictable sequence of incrementally increasing addresses. Such behavior, which has been demonstrated explicitly on a number of architectures, is a specific case of *value locality* [11, 28]. The predictability of load addresses can be exploited by predicting which data will be requested by the CPU ahead of the actual load requests.

Data prefetching techniques identify load addresses that miss in the cache and try to detect such predictable address patterns. Based on the identified patterns, data prefetchers predict future load addresses and issue load requests on behalf of a CPU. The data items requested by a prefetcher are pre-loaded into a certain level of cache and, when needed, can be readily consumed by a CPU. Thus, correctly

predicted prefetches can effectively eliminate stalls associated with long memory latencies and significantly increase program execution speed.

There are multiple ways to identify instructions with predictable load addresses. First, such instructions can potentially be detected via static program analysis. In this approach, the program binary or source code is analyzed prior to program execution. The limitation of this technique is a rather large number of run-time constants, which are not known at compile time. The second class of analysis techniques involves profile-driven approaches that analyze program behavior by running a set of sample inputs. While this approach provides better knowledge of the program behavior, it does not provide accurate estimation of temporal data locality that characterizes caching effects. As a result, it might result in incorrect conclusions about which load instructions are likely to miss or hit in the cache. The third class of analysis techniques represents dynamic analysis approaches which continuously measure the behavior of programs while they execute. This dynamic approach does not suffer from the limitations of static and profile-driven techniques, but it can potentially incur run-time overheads or require specific hardware support.

Due to the limitations of the static and profile-driven approaches, this dissertation focuses on developing new data prefetching algorithms utilizing dynamic techniques. As such, I propose approaches that can accelerate existing applications without requiring profiling or compiler support. Moreover, I propose to take advantage of available cores in a CMP to significantly lower the implementation complexity and hardware support usually associated with dynamic data prefetching techniques.

1.4 Contributions

The goal of this dissertation is to develop and evaluate novel prefetching techniques that can use available cores in multi-core microprocessors to prefetch data for computation threads running on active cores. My contributions toward this goal include the following:

- Future Execution based prefetching

  The development of a novel prefetching mechanism using a simple hardware value predictor to dynamically generate prefetching threads that can compute and prefetch a significant fraction of unpredictable load addresses.

- Prefetching with event-driven helper threading

  The design of a mechanism to implement prefetching algorithms based on value prediction entirely in software, improving the flexibility of prefetcher implementations and minimizing additional hardware complexity.

- Energy-efficient prefetching techniques

  Several approaches to substantially decrease the energy consumption of the proposed prefetching techniques by minimizing prefetching activity for program phases with low miss rates.

- Prefetching coverage analysis

  A comparative microarchitecture-independent analysis of the prefetch coverage provided by various prefetchers and validating the relevance of the proposed techniques in the context of future computer systems.

The results of this research have the potential to impact future generations of microprocessors in two important ways. First, they present relatively simple

ways to use extra cores of CMPs as helper engines for individual program threads. Thus, they enable multi-core architectures to accelerate applications with both little and abundant thread-level parallelism. Second, they resolve many limitations of hardware prefetching techniques previously proposed in the research literature and reduce their implementation complexity while providing competitive execution time speedups. The individual contributions are described in more detail in the following paragraphs.

A data-flow analysis of the instructions that compute load addresses revealed that many loads are caused by cache misses that execute repeatedly and whose address-generating program slices do not change (much) between consecutive executions. Based on this observation, I proposed the Future Execution prefetching technique, which dynamically generates a prefetching thread for each active core by simply sending a copy of all committed, register-writing instructions to another core. The key innovation is that on the way to the second core, a value predictor replaces each predictable instruction in the prefetching thread with a load immediate instruction, where the immediate is the predicted result that the instruction is likely to produce during its n*th* next dynamic execution. Experimental results show that executing the future execution prefetching thread on an available core of a CMP can prefetch a larger fraction of cache misses than conventional stride prefetching and can deliver a significant execution time speedup on a wide range of applications.

Next, I analyzed hardware prefetching algorithms that are based on value prediction. I found that despite a large number of interesting proposals, their introduction into commercial designs was often hampered by the considerable storage requirements for prediction tables and/or application specificity. To alleviate

these limitations, I developed an event-driven helper threading (EDHT) framework, which allows using general-purpose memory for storage and a software helper thread for executing a prefetching algorithm on an available CMP core. Experimental results show that implementations of various prefetching techniques within the EDHT framework provide performance improvements within 5% of pure hardware implementations.

I investigated a hybrid prefetching approach by running event-driven helper threads on top of a baseline with a conventional hardware stride prefetcher. I found that this approach significantly outperforms a configuration that only includes a hardware stride prefetcher. Furthermore, the results show that the EDHT approach provides competitive performance improvements over other approaches for muti-core helper threading while executing fewer instructions and requiring considerably less hardware support.

Next, I found that the proposed prefetching techniques considerably increase the energy consumption. Since energy consumption has recently became a high-priority concern even for high-performance microprocessors, I investigated techniques to reduce the energy overheads of Future Execution and Event-Driven Helper Threading. Experimental results show that the proposed techniques decrease the average energy overhead of the prefetching techniques by more than a factor of two.

Finally, I validated the general conclusions of this dissertation in the context of the new SPEC CPU2006 benchmark suite. First, I demonstrated that the high latency of memory accesses is likely to remain an important problem for CPU2006 programs. Second, I developed analysis techniques to evaluate the prefetching potential of various prefetching techniques. Interestingly, I found little difference

between various techniques for cache sizes larger than 2MB. Therefore, the differences between the analyzed prefetching approaches are more likely to originate from prefetch timeliness rather than from coverage. This result is similar to the properties observed for the SPEC CPU2000 programs and, therefore, the general conclusions of this dissertation are likely to remain relevant in the context of the newer benchmark suite release.

1.5 Summary

Scaling the performance of applications with little thread-level parallelism is one of the most serious impediments to the success of multi-core architectures. At the same time, the long latency of memory accesses represents one of the largest performance bottlenecks for individual program threads.

In this dissertation, I developed techniques that utilize extra cores of a CMP as helper engines to increase the performance of single program threads. In particular, my work focuses on using available cores to run data prefetching algorithms to mitigate the detrimental effects of the long memory latencies. The main contribution of this dissertation is the development and evaluation of two multi-core prefetching techniques that provide significant performance improvements for single-threaded applications running in a multi-core environment.

1.6 Organization

The remainder of this dissertation is organized as follows. Chapter 2 explains the impact of the load latency on modern superscalar CPUs and the operation of conventional data prefetching techniques. Chapter 3 describes the configuration of

the simulator that is used to measure the speedup numbers and discusses the application benchmark suite. Chapter 4 introduces the Future Execution technique and analyzes its impact on single-thread performance. Chapter 5 investigates the idea of Event-Driven Helper Threading and compares its performance with other dual-core helper threading techniques. Chapter 6 takes a closer look at the energy consumption of different helper threading techniques, proposes several improvements to make the techniques more energy-efficient, and evaluates their effectiveness. Chapter 7 presents a set of techniques for microarchitecture-independent evaluation of different helper-threading techniques based on trace-driven simulation and validates the general conclusions of the previous chapters in the context of the SPEC CPU2006 benchmark suite. Chapter 8 presents related work on data prefetching in uni-core and multi-core environments. Chapter 9 summarizes my work and takes a look into the future.

CHAPTER 2

BACKGROUND

This chapter provides background information on the operation of contemporary high-performance microprocessors. Furthermore, it describes data address prediction approaches and the operation of data prefetching techniques.

2.1 Conventional High-Performance Processor Architecture

Most modern high-performance microprocessors have a superscalar architecture with hardware support for dynamic out-of-order execution [35]. Since the goal of my work is to improve the performance of high-performance microprocessors, all experimental results in this dissertation are obtained by simulating the operation of such a CPU. The list of configuration parameters of the simulated CPU can be found in Chapter 3.

The term *superscalar* refers to CPU organizations that can execute multiple instructions at the same time. An *out-of-order* CPU is capable of dynamically re-arranging the order in which instructions are executed. Both superscalar and out-of-order execution techniques exploit the instruction-level parallelism (ILP) of computer programs. ILP is characterized by the existence of instructions that are not dependent on the results of each other and, consequently, can be executed in any order or in parallel.

Figure 2.1 shows a conceptual pipeline organization of such a CPU. First, instructions are fetched from the instruction cache. Then, they are decoded, renamed, and dispatched into the instruction window as long as there are buffer slots available. This dissertation assumes that these pipeline stages process instructions in program order. While there are proposals in the literature for out-of-order im-

plementations of these pipeline stages for increased performance [35], they incur significant hardware complexity and have yet to find their way into commercial microprocessor implementations.

After being dispatched into the instruction window, each instruction remains there until all of its source operands become available. The CPU's issue logic continuously scans the instruction window and searches for such instructions. If a ready instruction is found, the issue logic checks the availability of a functional unit (FU) that can execute this instruction. Once a functional unit becomes available, the instruction is sent to the assigned unit for execution. As a result, instructions in the instruction window might execute out of program order since their execution is determined exclusively by the availability of data values. Executed instructions are marked as completed. The commit stage of the CPU pipeline involves scanning instructions in the program order, identifying completed instructions, and retiring them from a CPU.

Modern superscalar CPUs can operate on multiple instructions in each stage of the pipeline. As a result, they are capable of processing several instruction per cycle (IPC) as long as there is enough ILP within the internal instruction window.

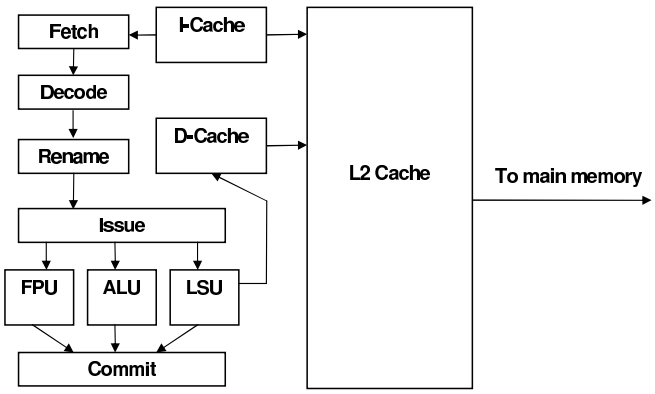

Figure 2.1: The pipeline of a high-performance microprocessor

Most program instructions that do not need to load data from memory have deterministic latencies (no more than a few cycles). Thus, they can always execute and retire fast, maintaining a high IPC rate. However, the latency of load instruction depends on the location of the requested data in the cache hierarchy. Out-of-order execution is often able to mitigate the effects of latencies associated with accessing second-level and third-level caches. Nevertheless, main memory latencies in modern CPUs are on the order of hundreds of cycles. Thus, if a load instruction misses in the cache, it eventually stalls the pipeline since it is very difficult for a CPU to find enough instructions to keep itself busy for so long.

2.2 Address Prediction and Prefetching

Hardware prefetchers often rely on various kinds of address predictors to dynamically predict which memory addresses to prefetch. In this work, we examined stride-based address prediction and Markov address prediction. This section briefly discusses both approaches and introduces the basic algorithms that I later use in this dissertation.

2.2.1 Stride Prefetching

Stride prefetchers represent the most common form of prefetching based on outcome prediction. The stride prefetcher is usually located in the cache controller, where it monitors the stream of cache miss requests observed by the cache. Stride prefetchers identify distinct streams within the sequence of cache misses, associate strides with each of the streams, and issue memory requests for the next few addresses in the stream. The simplest form of stride prefetching is next-sequential

prefetching, in which the prefetcher issues a request for cache line L+1 as soon as line L is referenced [36].

In many cases cache miss addresses are composed of several interleaved streams. A typical case of multiple streams is the traversal of several arrays within a matrix-matrix multiplication loop. Stride prefetchers employ special mechanisms to decipher and disambiguate interleaving streams. If the memory hierarchy propagates the program counter (PC) of the instructions that cause cache misses, the prefetcher can attribute misses to specific instructions and track streams on a per PC basis [10]. We call this *local stride prefetching*. The conventional implementation of a local stride prefetcher uses a table to store stride-related local history information and is shown in Figure 2.2a. The PC of the current load instruction indexes the table. Each table entry holds the load's last stride and the last address. A prefetch is triggered when the load causes a cache miss and its last stride is equal to the current stride.

If PC information is not available, then a *global stride prefetcher* can be used. Such stride prefetchers need to identify distinct streams within the global memory access pattern. Minimum delta prediction and memory partitioning were proposed to handle this problem. Minimum delta prediction associates a miss with the stream or prior miss that is the closest. Memory partitioning separates the physical

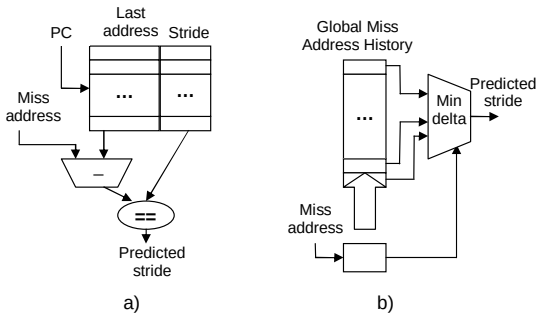

Figure 2.2: Organization of stride prefetchers

memory address space into regions and attributes all misses falling within a single region to a single stream [50]. Figure 2.2b shows the organization of a global stride prefetcher. When a cache miss occurs, the global miss history buffer is searched for the minimum difference between the previously observed addresses and the current miss address. An address stream is identified if the global miss history contains an address that differs from the current address by no more than two minimum deltas.

For each identified address stream, the prefetcher allocates a prefetch stream buffer from a limited number of available buffers. This buffer contains information about the current stream base address and the associated stride. Typically, stride prefetchers use an LRU replacement policy for stream buffers. The newly allocated stream buffer issues prefetches and then waits for the prefetched cache lines to be requested by the processor. Upon receiving such a notification, the stride prefetcher looks up its stream table to see which stream entry the consumed address belongs to. If it finds a match, it increments the corresponding stream address by the stream's stride and issues one new prefetch to keep up with the data consumption of the processor.

Overall, majority of cache misses in a wide range of applications exhibit stride pattern behavior (See chapter 7). This universal applicability and relative simplicity made stride prefetchers the most popular hardware prefetching technique in commercial microprocessors. Nevertheless, a significant portion of cache misses exhibit access patterns which are not detectable by stride prefetchers. The next subsection presents prediction approaches that attempt to identify and prefetch more complicated cache miss patterns.

2.2.2 Markov Prefetching

Markov prefetching [21] is another example of outcome-based prediction. A Markov predictor assumes that the address stream of a program can be approximated by a Markov model. A Markov model is a probabilistic state machine with a set of states and state transition probabilities. Each transition from state A to B is assigned a weight representing the fraction of A states that are followed by B states. Figure 2.3a presents an example of a Markov model. The states in the Markov model are determined by the set of previously seen values.

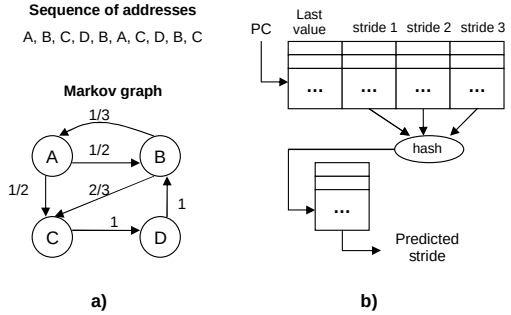

Figure 2.3: Markov prefetching

Markov models are usually characterized by two parameters. The first parameter determines what kind of values defines a state. In case of prefetching, the most common approach is to use either the absolute addresses or the differences between consecutive addresses. The second parameter determines how many values are used to determine the state. An order n Markov predictor associates each state with the n previous values.

Markov prefetching techniques incrementally build an Markov model for the target application at run-time. This model is later used by a prefetching mechanism to predict future addresses. Previous work on hardware-based Markov prefetching concentrated on finding optimal parameters for an accurate Markov model that

work well for many applications and on devising efficient hardware designs to store this model. The most common hardware implementation in the literature is based on a two-level table representation. The first table contains information to determine the current state (i.e., a node in the Markov model). As in the case of stream prefetchers, there are global and a local versions of Markov prefetchers. A local Markov prefetcher, shown in Figure 2.3b, uses the load's PC to index the first-level table, which stores a local history of the last three deltas for that load. A global Markov prefetcher uses a global history of the last three deltas (instead of a per-PC local history).

The calculated state serves as an index into the second-level table, which stores predictions (the immediate neighbors of the current node). To limit the total area required for the table, the second-level table usually contains only a limited number of unique predictions. A Markov prefetcher can prefetch the addresses predicted by the adjacent nodes in the Markov model. We refer to this policy as *width* prefetching. However, it is also possible to perform *depth* prefetching in which the sequence of arcs in the Markov model is traversed with prefetching initiated at each node along the path. We use a combination of width and depth prefetching in our experiments.

Overall, Markov prefetchers can predict many non-stride memory access patterns. This capability, however, often requires significant amount of storage for markov graphs. In spite of a high performance potential, limited transistor budgets and implementation complexity prevented a wide-spread use of markov algorithms in the modern CPUs.

CHAPTER 3

EVALUATION METHODOLOGY

This chapter describes the configuration of the baseline CPU that is used for the cycle-accurate simulations and gives information about the benchmark programs that are used for the performance evaluations.

3.1 Baseline Architecture

All cycle-accurate measurements in this dissertation are based on the DEC Alpha AXP architecture [7]. The various prefetching techniques are evaluated using an extended version of the SimpleScalar v4.0 simulator [27]. The simulator is configured to emulate a two-way high-performance CMP consisting of two identical four-wide dynamic superscalar cores that are similar to the Alpha 21264. It accurately models microprocessor's internal timing behavior, resource constraints, speculative execution as well as memory hierarchy and its latencies. Bandwidth and contention on the memory bus are also fully modelled.

The modeled CPU parameters of each CMP core are shown in Table 3.1. Each simulated CMP core is four-way superscalar, supports up to 128 in-flight instructions, issues instructions out-of-order from a 64-entry instruction window, has a 64-entry load/store queue, four integer and two floating-point units, a 64KB two-way set associative L1 data cache, a 64KB two-way set associative instruction cache, a 2048-entry branch target buffer (BTB), and a 16384-line hybrid gshare-bimodal branch predictor. A 2MB unified eight-way set associative L2 cache is shared between the two cores. All functional units are fully pipelined. Simulated cache latencies are calculated with CACTI 3.2 tool [44].

Table 3.1: Simulated processor parameters

CMP core	
Fetch/dispatch/commit width	4/4/4
I-window/ROB/LSQ size	64/128/64
Physical registers	184
LdSt/Int(IntMult)/FP units	2/4(2)/2
Branch predictor	16k-entry bimodal/gshare hybrid
RAS entries	16
BTB	2k entries, 2-way
Branch misprediction penalty	minimum 12 cycles
Memory Subsystem	
Cache sizes	64kB IL1, 64kB DL1, 2MB L2
Cache associativity	2-way L1, 8-way L2
Cache load-to-use latencies	3 cyc L1, 12 cyc L2
Cache line sizes	64B L1, 64B L2
Cache MSHRs	16 L1, 24 L2
Main memory latency	minimum 400 cycles
Main memory bus	split-trans., 8B-wide, 4:1 frequency ratio, contention, queuing, bandwidth modeled
Hardware stream prefetcher	between L2 and main memory, 16 streams, max. prefetch distance: 8 strides

Unless otherwise noticed, the baseline configuration includes an aggressive hardware global stride prefetcher [37] between the shared L2 cache and main memory. The stream prefetcher tracks the global history of the last 16 miss addresses, detects arbitrary-sized strides, and applies a stream filtering technique by only allocating a stream after a particular stride has been observed twice. It can simultaneously track 16 independent streams, and prefetch up to 8 strides ahead of the data consumption of the processor.

3.2 Benchmarks

This study uses all 26 integer and floating-point programs from the SPECcpu2000 benchmark suite [18]. The programs are run with the SPEC-provided reference inputs. If multiple reference inputs are given, we simulate the corresponding programs with up to the first three inputs and average the results from the different runs. The only exception to this rule is the program *vpr*, which has two ref-

Table 3.2: Benchmark suite details (for the simulated intervals of 500M instructions)

App.	NoPref IPC	loads (M)	L1 miss rate (%)	L2 miss rate (%)	perfect L2 speedup (%)
SPEC INT					
bzip2	1.56	143.94	1.47	0.70	24.55
crafty	1.92	155.85	0.82	0.07	2.10
eon	1.75	148.32	0.12	0.00	0.20
gap	1.44	127.02	0.36	1.22	24.62
gcc	1.33	180.30	2.56	1.10	30.38
gzip	1.69	113.80	3.52	1.77	3.35
mcf	0.04	209.74	23.11	48.62	1399.55
parser	0.84	125.79	2.56	2.40	103.48
perlbmk	1.77	146.69	0.30	0.09	7.42
twolf	1.29	144.63	5.04	0.05	1.84
vortex	2.09	130.30	0.75	0.39	34.55
vpr	0.54	165.19	3.10	3.29	120.41
SPEC FP					
ammp	1.44	132.32	3.90	1.35	31.92
applu	0.97	114.27	2.10	6.64	198.38
apsi	2.43	120.67	1.52	0.78	10.81
art	0.69	148.46	19.72	6.97	183.48
equake	0.26	234.53	7.50	24.40	675.56
facerec	1.15	123.90	2.40	4.83	178.51
fma3d	0.80	150.17	3.01	7.10	217.41
galgel	2.49	184.71	2.94	0.29	2.00
lucas	0.42	80.69	7.91	23.68	431.63
mesa	1.92	129.16	0.32	0.55	23.04
mgrid	0.91	183.09	2.42	19.02	203.19
sixtrack	2.62	96.83	0.23	0.18	4.77
swim	0.42	123.65	8.51	19.11	689.75
wupwise	1.30	114.31	1.15	3.60	134.67

erence inputs. Only one of the reference inputs is simulated (routing) because SimpleScalar could not simulate *vpr* correctly with the second input (placement). The C programs were compiled with Compaq's C compiler V6.3-025 using "–O3 –arch ev67 –non_shared" plus feedback optimization. The C++ and Fortran 77 programs were compiled with g++/g77 V3.3 using "–O3 –static". The Fortran 90 programs were compiled with Compaq's f90 compiler V5.3-915.

SimPoint 3.1 toolset [43] and SimpleScalar's *sim-safe* simulator are used to identify representative simulation points. Each program is simulated for 500 million instructions after fast-forwarding past the number of instructions determined by SimPoint.

Table 3.2 provides information about the benchmarks used. The first column represents the baseline IPC for each program. The second column shows the total number of load instruction in the simulated program interval. The third and fourth columns show local miss rates for L1 data cache and L2 unified cache. Finally, the fifth column represents the speedup for each application in the case where all load requests that miss in L1 cache. Thus, the last column demonstrated the maximum potential speedup achievable by data prefetching techniques.

Out of the 26 programs used in this study, four integer and two floating-point programs are not memory-bound since they obtain less than 5% speedup with a perfect L2 cache. The perfect-cache speedup for the rest of the programs varies greatly and reaches up to 1400% for *mcf*. This large speedup is explained by an exceptionally large number of L1 misses and a very high L2 cache miss rate that reaches 48.6%. Note that for several memory-bound programs (e.g., *mcf*, *art*, *equake*, and *swim*) the perfect L2 cache speedup cannot be obtained even with a perfect prefetching scheme because of memory bus bandwidth limitations.

CHAPTER 4

FUTURE EXECUTION

This chapter presents the investigation of program properties related to cache misses and presents the idea of the Future Execution prefetching technique. Furthermore, it describes the implementation of Future Execution and performance results. A sensitivity analysis of future execution to several architectural parameters concludes this chapter.

4.1 Motivation

This section presents a quantitative analysis of the common program properties that are exploited by future execution. All results are obtained using the benchmark suite and baseline microarchitecture described in Section 4.3.

One of the main program properties exploited by FE is that most load misses occur in "loops" with relatively short iterations. Note that we call any repetitively executed instruction a loop instruction and that FE is completely unaware about the location of loops in a program. Figure 4.1 presents the breakdown of the distance between the load instructions that cause an L2 cache miss and the previous execution of the same load instruction. The bars are broken down by distance: fewer than 100, between 100 and 1000, and between 1000 and 10000 dynamic instructions. The taller the bar, the more often that range of instruction distances occurred. The total height of the bar represents the fraction of L2 cache misses that occur in loops with less than 10000 instructions per iteration.

The data show that on average from 70% to 80% of the misses occur in loops with iterations shorter than 1000 instructions. This observation suggests a prefetching approach in which each load instruction triggers a prefetch of the address

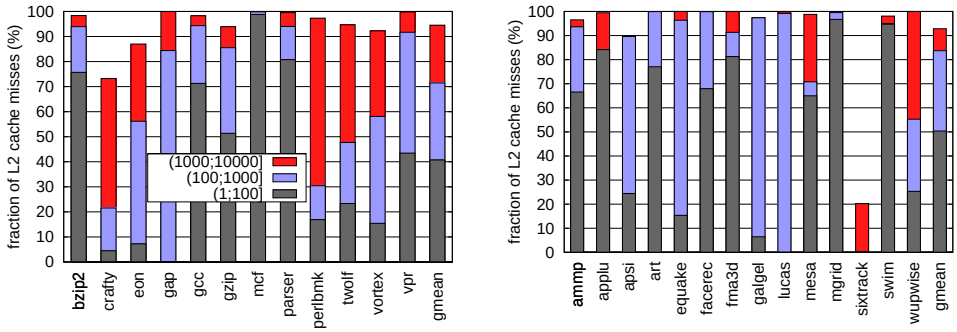

Figure 4.1: Execution distance measured in number of instructions between the loads that result in an L2 cache miss and the previous dynamic execution of the same static loads

that the same load is going to reference in the n^{th} next iteration. Since in most cases the time between the execution of a load instruction and its next dynamic execution is relatively short, this approach is unlikely to prefetch much too early.

Analyzing the instructions in the dataflow graphs of the problem loads, I found that while problem load addresses might be hard to predict, the inputs to their dataflow graphs often are not. Therefore, even when the miss address itself is unpredictable, it may be possible to predict the input values of the instructions leading up to the problem loads and thus to compute an accurate prediction by executing these instructions.

Figure 4.2 shows the breakdown of the load miss addresses in the SPECcpu2000 programs that can potentially be predicted and prefetched by future value prediction and by future execution one iteration ahead of the main program execution. The lower portion of each bar represents the fraction of misses that is directly predictable by a stride-two-delta (ST2D) value predictor [42]. The upper bar shows how many miss addresses that are not predictable by the ST2D predictor can be correctly obtained by predicting the inputs of the instructions in the dataflow graph of the missing loads with ST2D predictor and computing the resulting ad-

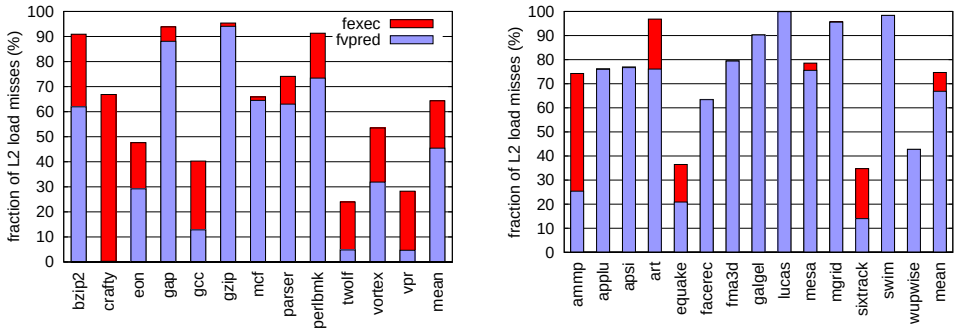

Figure 4.2: Distribution of cache miss addresses that can be correctly predicted directly by a future value predictor (*fvpred*) and using future execution (*fexec*)

dress. The height of the stacked bar indicates the total fraction of misses that can potentially be correctly predicted. To measure the potential prediction coverage of future execution, we reconstruct the dataflow graph of each problem load whenever a cache miss occurs, compare it to the dataflow graph of the same static load during its previous execution, extract the part of the dataflow graph that is the same, and then check if the values provided by the future value predictor during the previous execution would have allowed to correctly compute the load address referenced by the load instruction in the current iteration. The size of analyzed dataflow graph is limited to 64 instructions. This potential study ignores the effects of unpredictable loop-carried dependencies passed through memory, i.e., all load instructions with predictable addresses are assumed to fetch correct data one iteration ahead.

Figure 4.2 illustrates that while value prediction alone is quite effective for some applications, future execution can significantly improve the fraction of load miss addresses that can be correctly predicted and prefetched. Half of the SPECcpu2000 programs experience a significant (over 10%) increase in prediction coverage when future execution is employed in addition to value prediction.

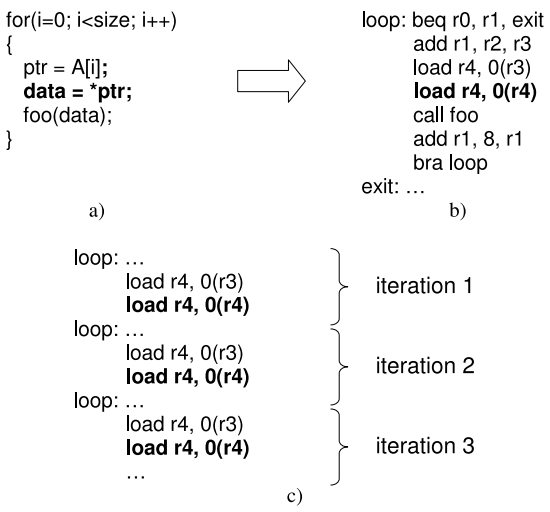

Figure 4.3: Code example

Figure 4.3a shows a code example that exhibits the program properties discussed above. An array of pointers A is traversed, each pointer is dereferenced and the resulting data are passed to the function "foo". Assume that the data referenced by the elements of array A are not cache-resident. Further assume that there is little or no regularity in the values of the pointers stored in A. Under these assumptions each execution of the statement `data=*ptr` will cause a cache miss. As shown in Figure 4.3b, in machine code this statement translates into a single load instruction `load r4, 0(r4)` (highlighted in bold).

A conventional predictor will not be able to predict the address of the problem instruction since there is no regularity in the address stream for this instruction. However, the address references of instruction `load r4, 0(r3)` are regular because each instance of this instruction loads the next consecutive element of array A. Therefore, it is possible to use a value predictor to predict the memory addresses for this instruction, speculatively execute this instruction, and then use the speculatively loaded value to prefetch the data for the problem load instruction. Since the control flow leading to the computation of the addresses of the

problem load remains the same throughout each loop iteration (Figure 4.3c), a value predictor can provide predictions for the next iterations of the loop and the addresses of the problem load will be computed correctly. Therefore, sending the two load instructions to the second core in commit order and future predicting the first instruction makes it possible to compute the addresses of the second load that will be referenced by the main program during the next iterations.

4.2 Implementation of Future Execution

The implementation of future execution is based on a conventional chip multiprocessor. A high-level block diagram of a two-way CMP supporting FE is shown in Figure 4.4. Both microprocessors in the CMP have a superscalar execution engine with private L1 caches. The L2 cache is shared between the two cores. Conventional program execution is performed on the "left" core while future execution is performed on the "right" core. To support FE, a unidirectional communication link is introduced between the cores with a value predictor attached to it. Both the communication link and the predictor are not on the critical path and should not affect the performance of either core in a negative way. The following subsections describe the necessary hardware support and the operation of FE in greater detail.

4.2.1 Overview of Operation

Each register-writing instruction committed in the regular core is sent to the second core via the communication link. The data that need to be transferred to the second core include the decoded instruction, result value, and a partial PC to index the value predictor table. Stores, branches, jumps, calls, returns, privileged instructions, and system calls are not transmitted. If the communication link's

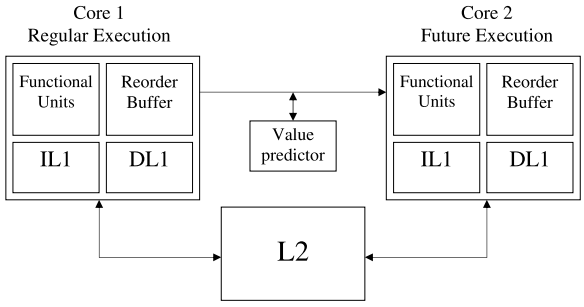

Figure 4.4: The FE architecture

buffer is full, further committed instructions are dropped and not transmitted to the second core. Each sent instruction passes through the value predictor, updates the predictor with its current output and requests a prediction of the value it is likely to produce in the n^{th} next dynamic instance. Each prediction is accompanied by a confidence estimation [28].

If the confidence of the prediction is high, the corresponding instruction is replaced by a *load immediate* instruction, where the immediate is the predicted result. If the predicted instruction is a memory load, an additional non-binding prefetch instruction for that load's address is generated right before the *load immediate* instruction. This allows the future core to prefetch this data without stalling the pipeline if the memory access misses in the cache. All instructions with a low prediction confidence remain unmodified.

After that, the processed stream of committed instructions is sent to the second core, where it is injected into the dispatch stage of the pipeline. Since instructions are transmitted in decoded format, they can bypass the fetch and decode stages. Instruction dispatch proceeds as normal – each instruction is renamed and allocated a reservation station and a ROB entry if these resources are available. Whenever the input values for the instruction are ready, it executes, propagates the produced result to the dependent instructions and updates the register file. If the instruction

at the head of the ROB is a long latency load, it is forced to retire after a timeout period that equals the latency of an L2 cache hit. This approach significantly improves the performance of the FE mechanism as it avoids stalling the pipeline. Timed out instructions set the invalid bit in the corresponding result register. The execution of instructions that depend on the invalidated result is suppressed.

When entering FE mode, i.e., before the prefetching thread starts being injected into the available idle core, all registers of that core are invalidated. This flash invalidation of all registers occurs only once before the future execution thread is launched. The invalid register bits are gradually reset by the executed future instructions that have valid inputs. For example, *load immediate* instructions always make their target registers valid since they do not have input dependencies. This register invalidation policy suppresses the execution of all further instructions whose inputs cannot be predicted or computed with high confidence.

Note that the implementation of future execution in this dissertation differs from the implementation used in some of the previous studies [12, 13]. This study simplifies the implementation so that all instruction transformations take place outside the core. For example, our previous implementation required special logic in the dispatch stage of the pipeline to fill in the result field of the ROB and RS entries of FE instructions with predicted values. The implementation presented in this dissertation is more intuitive, requires no additional dispatch logic to support future execution, and features a simpler way to suppress the execution of unpredictable instructions.

4.2.2 Hardware Support

Future execution requires additional hardware support to transmit decoded instructions, their result values, and partial PCs to the value predictor between the cores. Depending on the microarchitecture, the ROB may have to be augmented to hold the necessary data until instruction retirement. In this study, the communication bandwidth corresponds to the commit width (4 instructions/cycle), which is a reasonable bandwidth for a unidirectional on-chip point-to-point link. Since it is rare for a microprocessor to fully utilize its commit bandwidth for a long period of time, and because not all instructions need to be transmitted to the second core, it may be possible to curtail the bandwidth without significant loss of performance. For example, Section 4.4.4 demonstrates that the communication bandwidth can be reduced to 2 instructions/cycle with little effect on the efficiency of the FE mechanism.

The value prediction module resides between the two CMP cores. The presented implementation uses a relatively simple, PC-indexed stride-two-delta predictor [42] with 4,096 entries. The predictor estimates the confidence of each prediction it makes using 2-bit saturating up-down counters. The confidence is incremented by one if the predicted value was correct and decremented by one if the predicted value was wrong. The particular organization of the value predictor is not essential to our mechanism and a more powerful predictor (e.g., a DFCM predictor [14]) may lead to higher performance.

To support the execution of the future instruction stream, a multiplexer has to be added in front of the dispatch stage of the pipeline. In FE mode, the multiplexer directs instructions to be fetched from the receive buffer of the communication link. In normal mode, instructions are fetched by the processor's front end.

The processor's register file may have to be extended to accommodate an invalid bit for each physical register. Only one extra bit per register is needed. Many modern microprocessors already include some form of dirty or invalid bits associated with each register that could be utilized by the FE mechanism.

Since I model a two-way CMP with private L1 caches, a mechanism is needed to keep the data in the private L1 caches of the two cores consistent. This work relies on an invalidation-based cache coherency protocol for this purpose. Therefore, whenever the main program executes a store instruction, the corresponding cache block in the private cache of the future core is invalidated. Since store instructions are not sent to the future core, future execution never incurs any invalidations.

4.3 Evaluation Methodology

Future execution is evaluated using an extended version of the SimpleScalar v4.0 simulator [27]. The baseline is a two-way CMP consisting of two identical four-wide dynamic superscalar cores that are similar to the Alpha 21264. In all modeled configurations it is assumed that one of the cores in the CMP can be used for future execution. The full description of the baseline architecture and benchmark suite is provided in Chapter 3.

Table 4.1 describes the configuration of future execution parameters. The communication latency between the two cores is 5 cycles and the communication bandwidth corresponds to the commit width (4 instructions/cycle). Note that FE is not very sensitive to the communication latency (see Section 4.4.4). The implementation of the future execution mechanism employs a stride-two-delta (ST2D) value predictor [42] that predicts values four iterations ahead. Predicting four iterations ahead does not require extra time in case of the ST2D predictor. The predictor

Table 4.1: Future Execution Parameters

Future value predictor	4k-entry ST2D, 2bc conf. estimator
Prediction distance	4 strides ahead
Inter-core communication link	5-cycle latency, 4 insns/cycle bandwidth
Communication link buffer size	64 instructions

hardware needs to be modified to add the predicted stride four times, which is achieved by a rewiring that shifts the predicted stride by two bits.

4.4 Experimental Results

In this section, I experimentally measure the effectiveness of the proposed mechanism. Section 4.4.1, evaluates the performance of prefetching based on future execution and compares the speedups with those of stream prefetching. Section 4.4.2 takes a closer look at prefetching itself and gains additional insight by measuring the prefetching accuracy and coverage as well as the timeliness of prefetches. Section 4.4.3 compares the future execution technique to prefetching based on several variations of runahead execution and show that the two techniques are complementary to each other. Finally, Section 4.4.4 studies the sensitivity of future execution to several parameters of the baseline microprocessor, such as the minimum memory latency, the inter-core communication delay/bandwidth, and the prefetch distance.

4.4.1 Execution Speedup

This section compares the performance impact of the proposed prefetching technique to a stream-based hardware prefetcher. The base machine for this experiment is described in Table 3.1. It represents an aggressive superscalar processor with-

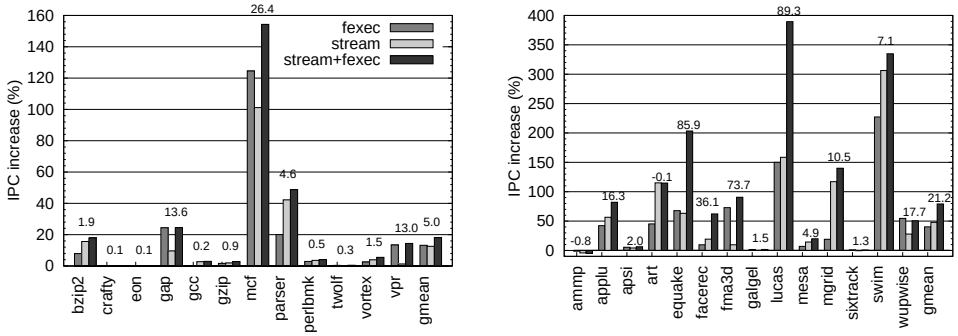

Figure 4.5: Execution speedup

out hardware prefetching. We model three processor configurations: the baseline with prefetching based on future execution (*fexec*), the baseline with an aggressive hardware stream prefetcher between the shared L2 cache and main memory (*stream*) [37], and the baseline with stream prefetching as well as future execution (*stream+fexec*). Figure 4.5 presents speedups for individual programs as well as the geometric mean over the integer and the floating-point programs (integer programs are shown in the left panel, floating-point programs in the right panel). Note that the scale of the y-axis for the SPECint and the SPECfp benchmarks is different. The percentages on top of the bars are the speedups of future execution combined with stream prefetching (*stream+fexec*) over stream prefetching alone (*stream*).

The results show that the hardware stream prefetcher used in this study is very effective, attaining significant speedups for the majority of the programs, with peaks of 306% for *swim* and 159% for *lucas*. The average speedup over the SPECint programs is 13%, while the SPECfp applications experience an average speedup of 48%. Note that the parameters of the stream prefetcher are tuned to maximize the prefetching timeliness and to minimize cache pollution on our benchmark suite.

When the model with only future execution is compared to the model with only stream prefetching, future execution outperforms stream prefetching on five programs, while stream prefetching is better on nine. The remaining twelve programs achieve about the same performance with both models. As the following section will show, in many cases the stream prefetcher can prefetch fewer load misses than future execution, but it provides more timely prefetching and hence larger performance improvements. The timeliness of the prefetches issued by future execution can be improved by adjusting the prediction distance of the future value predictor, but this study uses a fixed prediction distance (except for Section 4.4.4) to make the results comparable. Nevertheless, the *fexec* model provides significant speedup (over 5%) for 12 programs, with an average speedup of 13% for the integer and 40% for the floating-point programs, and a maximum of 227% on *swim*.

The model with the best performance is the one that combines the stream prefetcher and future execution. On average, this model has a 50% higher IPC than the model with no prefetching. Moreover, this model has a 10% higher IPC than the baseline with stream prefetching. Out of the 26 programs used in our study, 12 significantly benefit (over 5% improvement) from future execution when it is added to the baseline that already includes a hardware stream prefetcher. Looking at the behavior of the integer and floating-point programs separately, adding future execution to the baseline with a stream prefetcher increases the performance of SPECint and SPECfp by 5% and 21%, respectively. This indicates that future execution and stream prefetching interact favorably and complement each other by prefetching different classes of load misses.

Overall, the results in this section demonstrate that future execution is quite effective on a wide range of programs and works well alone and in combination

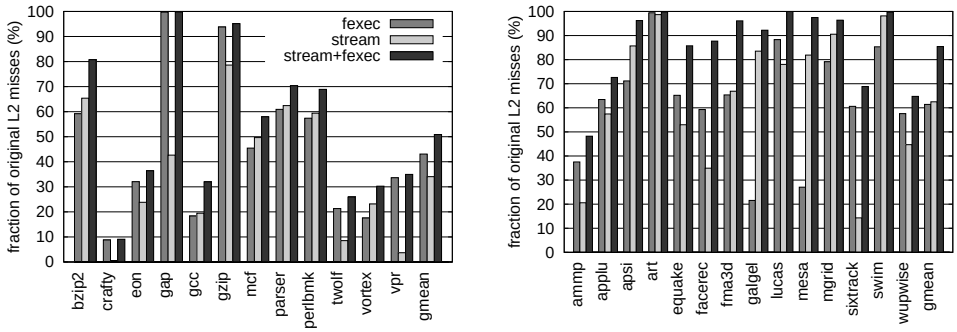

Figure 4.6: Prefetch coverage

with a stream prefetcher.

4.4.2 Analysis of Prefetching Activity

This section provides insight into the performance of prefetching based on future execution by taking a closer look at the prefetching activity. It begins by presenting the prefetch coverages obtained by different prefetching techniques. The prefetch coverage is defined as the ratio of the total number of useful prefetches (i.e., the prefetches that reduce the latency of a cache missing memory operation) to the total number of misses originally incurred by the application.

Figure 4.6 shows the prefetch coverages for different prefetch schemes, illustrating significant coverage, especially for SPECfp. On roughly half of the programs the coverage achieved by future execution is higher than that achieved by the stream prefetcher. The value predictor that assists the future execution makes predictions based on the local history of values produced by a particular static instruction, while the stream prefetcher observes the global history of values. Therefore, the two techniques exploit different kinds of patterns, akin to local and global branch predictors.

When stream prefetching is combined with future execution, the two techniques

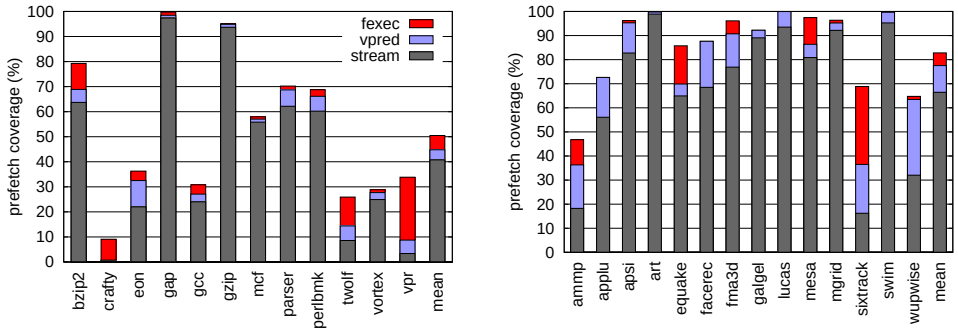

Figure 4.7: Distribution of cache misses that were prefetched by a stream prefetcher (*stream*), based on value predictions (*vpred*), and using future execution (*fexec*)

demonstrate significant synergy. In eleven programs (*bzip2*, *gcc*, *perlbmk*, *ammp*, *applu*, *apsi*, *equake*, *facerec*, *fma3d*, *lucas*, and *mesa*) the coverage is at least 10% higher than when either technique is used alone. Overall, future execution increases the prefetching coverage from 34% to 51% on the integer and from 63% to 85% on the floating-point programs.

Figure 4.7 shows the breakdown of the prefetch coverage for the case when stream prefetching is combined with future execution (*stream+fexec* configuration). The lower segment of each bar corresponds to the prefetches initiated by the stream prefetcher. The middle portion shows how many prefetches were issued based on predictions provided by the value predictor. The upper part of each bar represents the portion of issued prefetch addresses that required the execution of instructions to compute the correct prefetch address. The height of the stacked bar indicates the total fraction of misses that were prefetched. Note that the height of the bar is sometimes slightly less than reported in Figure 4.6 because the origin of some of the prefetches could not be determined. In this study, if multiple prefetch mechanisms issued prefetches for the same memory location, the mechanism that issued the earliest prefetch is given credit for that prefetch request.

Figure 4.7 illustrates that future value prediction and future execution provide

a significant coverage increase over the stream prefetcher (over 10%) for 16 out of the 26 applications used in our study. Out of these 16 applications, six programs benefit mostly from future execution, eight programs owe most of the coverage increase to future value prediction, and two programs benefit roughly equally from value prediction and future execution. Note that in seven programs the addition of future execution makes the stream prefetcher more effective. For example, *gap* gets almost all of its misses prefetched by the stream prefetcher, but the data in Figure 4.6 demonstrate that stream prefetching can prefetch less than a half of the cache misses without future execution. I suspect that this is caused by favorable interactions between the loads issued from the future core and the stream prefetcher, where future loads enable more precise and earlier identification of important streams that are then further prefetched by the stream prefetcher. On average, future execution increases the coverage provided by the stream prefetcher from 34% to 41% on the integer applications and from 63% to 66% on the floating-point programs.

Next, I analyze the accuracy of our prefetching scheme by comparing the number of useless prefetches issued by the prefetching mechanisms to the total number of prefetches issued. A prefetch request is categorized as useless if the prefetched data is evicted from the cache before being used by the main thread. Figure 4.8 shows the percentage of useless prefetches associated with the two prefetching schemes. The results illustrate that a large majority of the prefetches issued are useful in both the SPECint and the SPECfp programs with over 70% of useful prefetches for both techniques. There are a few interesting cases where stream prefetching causes much fewer useless prefetches than future execution. They occur in the programs *eon*, *gap*, *twolf*, *facerec*, and *sixtrack*. I find that useless

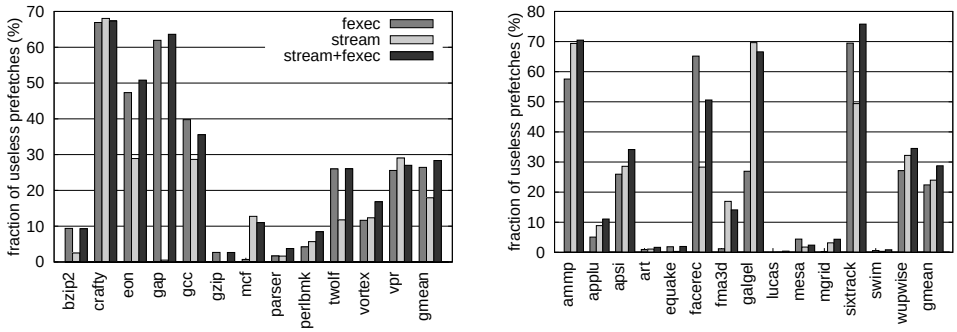

Figure 4.8: Percentage of useless prefetches issued by different prefetching mechanisms relative to the total number of prefetches issued

prefetches occur in loops where many loads depend on the values of a loop-carried dependency passed through memory that is not preserved by future execution. This results in computing the wrong addresses for load instructions and the fetching of useless data. I suspect that in *sixtrack* many prefetches are issued too far in advance and get evicted from the cache before being used. However, even though the accuracy of the stream prefetcher is higher in those cases, the coverage for many of the programs is quite small, meaning that the higher accuracy does not noticeably improve the performance.

Finally, I investigate the prefetch timeliness of the different schemes. The prefetch timeliness indicates how much of the memory latency is hidden by the prefetches. The results are presented in Figure 4.9. For each program, the upper bar corresponds to the *fexec* model, the middle bar to the *stream* model, and the lowest bar represents the *stream+fexec* model. Each bar is broken down into five segments corresponding to the fraction of the miss latency hidden by the prefetches: less than 100 cycles (darkest segment), between 100 and 200 cycles, between 200 and 300 cycles, between 300 and 400 cycles, and over 400 cycles (lightest segment). Therefore, the lightest segment represents the fraction of prefetches that hide the minimum full memory latency.

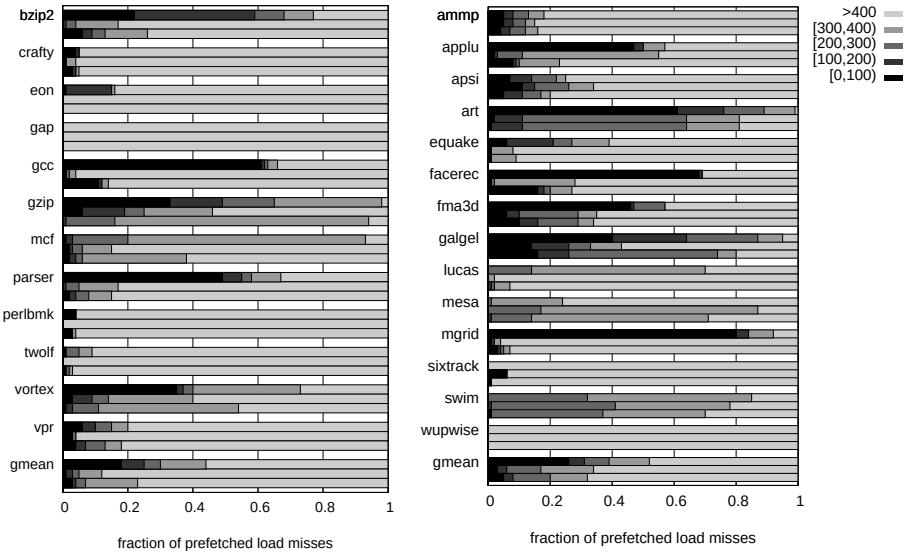

Figure 4.9: Timeliness of the prefetches

Both future execution and stream prefetching are quite effective at hiding the memory access latency. In case of future execution, 65% of the prefetches in SPECint and 55% of the prefetches in SPECfp are completely timely, fully eliminating the associated memory latency. For both the integer and the floating-point programs, only 25% of the prefetches hide less than 100 cycles of latency (one quarter of the memory latency). The timeliness of future execution prefetches can be improved by adjusting the prediction distance of the future value predictor. For example, increasing the prediction distance from 4 to 8 increases the number of completely timely prefetches for most of the programs with a low prefetch timeliness by at least 15%, resulting in significant increase in performance (see Section 4.4.4).

Overall, this section demonstrates that prefetching based on future execution is quite accurate, significantly improves the prefetching coverage over stream prefetching, and provides timely prefetches, which may be further improved by dynamically varying the prediction distance.

4.4.3 Comparison with Runahead Execution

The previous subsections showed that prefetching based on future execution is quite effective and provides significant speedups over the baseline with an aggressive stream prefetcher. In this section I compare our mechanism to several variations of runahead execution, an execution-based prefetching technique.

The concept of runahead execution was first proposed for in-order processors [8] and then extended to perform prefetching for out-of-order architectures [33]. The runahead architecture "nullifies" and retires all memory operations that miss in the L2 cache and remain unresolved at the time they reach the ROB head. It starts by taking a checkpoint of the architectural state and retiring the missing load before the processor enters runahead mode. Once in runahead mode, instructions execute normally except for two major differences. First, the instructions that depend on the result of the load that was "nullified" do not execute but are nullified as well. They commit an invalid result and retire as soon as they reach the head of the ROB. Second, store instructions executed in runahead mode do not overwrite data in memory. When the original "nullified" memory operation completes, the processor rolls back to the checkpoint and resumes normal execution. All register values produced in runahead mode are discarded.

I implemented a version of runahead execution similar to the one described by [33]. Runahead mode is triggered by load instructions that stall for more than 30 cycles. Store data produced in runahead mode is retained in a runahead cache, which is flushed upon the exit from runahead mode.

In addition to this conventional version of runahead execution, I implement and evaluate two extensions. First, I employ value prediction to supply load values for the long-latency load instructions. When such loads time-out, a stride-two-delta

value predictor provides a predicted load value and a prediction confidence. If the confidence is above threshold, the predicted value is allowed to propagate to the dependent instructions. If the confidence is below threshold, the result of the load instruction that timed out is invalidated in the same way loads are invalidated in the conventional runahead mechanism.

Second, I implement the checkpointed early load retirement mechanism (CLEAR) [25], which attempts to avoid squashing the correct program results produced in runahead mode. The CLEAR mechanism utilizes value prediction to provide values for the load instructions that time out and is similar in spirit to checkpoint-assisted value prediction as proposed by [4]. While the conventional runahead mechanism checkpoints the processor state only once before entering runahead mode, CLEAR checkpoints the processor state before every prediction that is made in runahead mode. If the value provided by a value predictor was incorrect, the processor state is rolled back to the checkpoint corresponding to that value prediction. However, if the prediction was correct, the corresponding checkpoint is released and the processor does not have to roll back after the long-latency memory operation completes.

Note that both runahead extensions that are evaluated in this study share value prediction and confidence estimation tables with the future execution mechanism. When runahead is used without future execution, only load instructions update the value predictor. When runahead and future execution are used together, every committed instruction updates the value predictor with the exception of stores, branches, and system calls. The implementation of CLEAR assumes an unlimited number of available checkpoints.

Figure 4.10 shows the execution speedup of different techniques relative to the

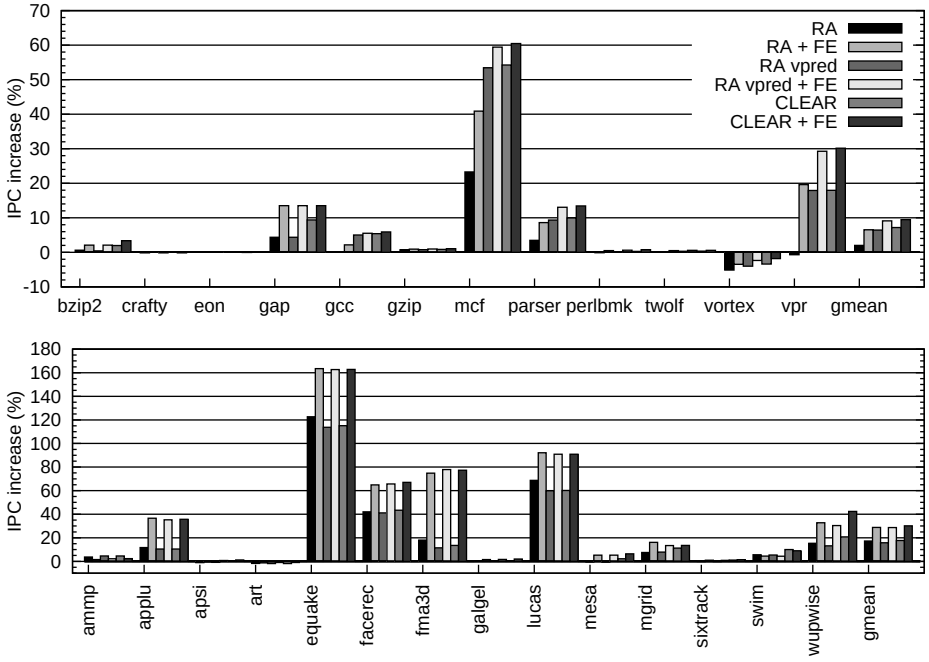

Figure 4.10: Comparison with runahead execution

stream baseline. First, we compare the performance of future execution without runahead execution, shown in Figure 4.5, to the performance of different runahead schemes when they are used without future execution. Overall, the geometric-mean speedups of different runahead techniques and FE are similar when they are applied separately. On average, conventional runahead, runahead with value prediction, and CLEAR provide performance improvements of 2%, 6.4%, 7.2% on SPECint and around 17% on SPECfp. FE provides 5% speedup on SPECint and 21% on SPECfp applications.

When runahead execution and future execution are employed together, their cumulative effect is quite impressive. The average speedups for the conventional runahead implementation rise to 6.5% and 29% for the integer and the floating-point programs, respectively. The average speedups for runahead with value prediction also exhibits a significant boost with future execution, increasing from 6.4%

to 9.1% for SPECint and from 16% to 29% for SPECfp applications. The CLEAR mechanism demonstrates similar performance improvements. The interaction between future execution and runahead execution is especially favorable with eight programs (*mcf*, *vpr*, *applu*, *facerec*, *fma3d*, *equake*, *mgrid*, and *wupwise*), where the speedups are from 5% to 50% higher than when either of the techniques is used alone.

Runahead allows prefetching cache misses that are within close temporal proximity of the long-latency load instructions that initiated runahead mode. Therefore, even though runahead execution obtains significant prefetch coverage while the processor is in runahead mode, its potential is limited by the length of the runahead period. On the other hand, FE prefetches can generally hide more latency than runahead prefetches because the FE mechanism issues memory requests several iterations ahead of the current execution point. In spite of the better prefetch timeliness, FE's coverage is sometimes limited by the value prediction coverage and the regularity of the address-generating slices. The combination of runahead execution and future execution allows to exploit the strengths of both approaches, thus resulting in symbiotic behavior.

4.4.4 Sensitivity Analysis

In this subsection I evaluate the effectiveness of future execution when several hardware parameters are varied. First, I investigate the effect of the memory latency on FE's performance. Second, I compare the performance of FE configurations with different inter-core communication latencies and bandwidth capabilities. Finally, I analyze the performance benefits provided by FE with different prefetch

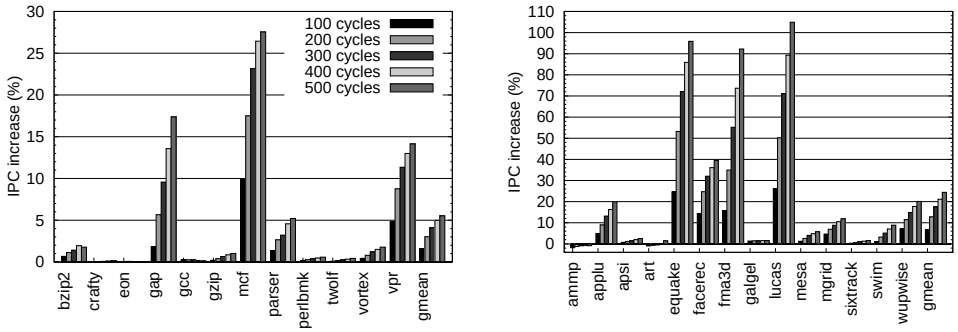

Figure 4.11: Sensitivity of future execution to the main memory latency

distances. All results in this subsection show the speedup provided by FE relative to the *stream* baseline.

Figure 4.11 shows the speedup provided by FE on processors with five different memory latencies, ranging from 100 to 500 cycles. Overall, the performance benefit for both integer and floating-point programs steadily increases with increasing memory latency. The SPECint speedup ranges from 1.6% for a relatively short 100-cycle memory latency to almost 5.5% for a 500-cycle latency. The average speedup for the SPECfp programs increases from 6.7% to 24.4%.

Figure 4.12 demonstrates how the communication latency between the cores affects the performance improvements provided by FE. As the communication latency is increased from 5 to 30 cycles, most of the applications show no significant changes in the amount of speedup obtained by future execution. The speedup changes by more than 5% only for four programs (*vpr*, *applu*, *facerec*, and *fma3d*). We observe that longer communication latencies seem to hurt the performance benefit in *fma3d*. Future execution in *vpr* and *facerec* generally becomes more effective with the increasing communication delay, while *applu* demonstrates no correlation between the speedup and the communication latency. The geometric mean speedups for the SPECint and SPECfp applications change by less than

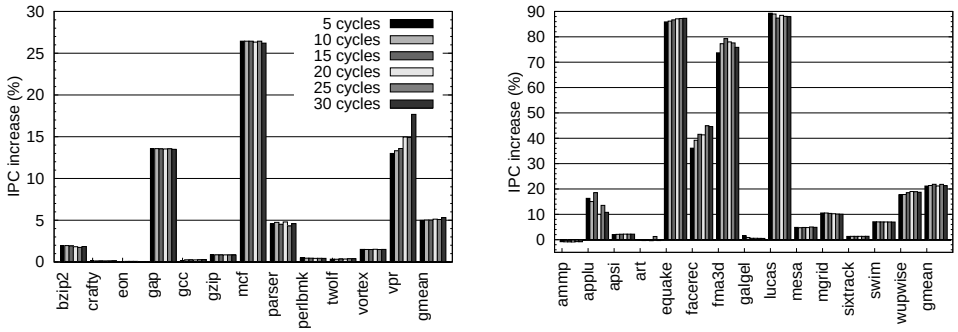

Figure 4.12: Sensitivity of future execution to the inter-core communication delay

0.5%. FE is not very sensitive to the communication delay because of two main reasons. First, prefetching four iterations ahead hides the full memory latency for many of the applications and delaying a prefetch request by an additional 5-25 cycles still results in a timely prefetch. Second, even if the delayed prefetch does not hide the full memory latency, the communication delay constitutes only a small fraction of the total memory latency. Increasing the latency of a prefetched memory request by a few percent does not have a significant performance impact.

Figure 4.13 shows the sensitivity of future execution to the communication bandwidth between the cores. The communication bandwidth is varied from 4 to 1 instructions/cycle. The results show that decreasing the bandwidth from 4 to 3 instructions/cycle has almost no impact on the effectiveness of FE. If the communication bandwidth is cut in half to 2 instructions/cycle, only three programs experience a significant (over 5%) performance degradation (*vortex*, *applu*, and *equake*), while the geometric mean speedups stay practically unchanged. However, further reduction of the bandwidth to 1 instruction/cycle often makes the bandwidth insufficient for the effective operation of FE. In particular, six programs experience a significant performance degradation. Compared to the case where the communication bandwidth corresponds to the processor commit width

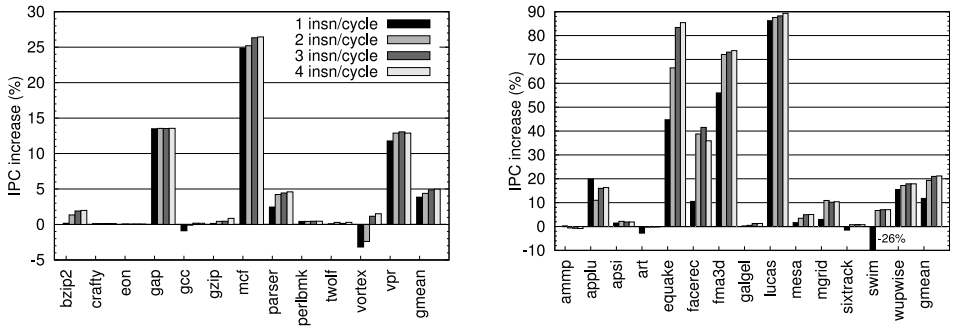

Figure 4.13: Sensitivity of future execution to the inter-core communication bandwidth

(4 instructions/cycle), the geometric mean speedup for integer programs decreases from 5% to 3.8%, while the IPC speedup for the floating-point applications drops from 21% to 12%. In most cases, this degradation is caused by a large number of instructions that are dropped due to the lack of space in the communication buffer. As a result, some prefetch addresses are never computed, while others are computed incorrectly and pollute the cache or the stream prefetcher's miss history (e.g., *swim*).

Next, I analyze the impact of the prefetch distance on the performance of future execution. I vary the prediction distance of the value predictor from 1 to 10 iterations ahead and show the corresponding speedups in Figure 4.14. The results show that most of the programs benefit from an increased prefetch distance. As one might expect, the prefetch coverage generally decreases slightly for larger lookaheads, but the reduction in coverage is compensated for by the improved timeliness of the prefetch requests. *Vpr* and *ammp* are the only programs where the decreasing prefetch coverage dominates the improved timeliness. Surprisingly, some programs (e.g., *bzip2*, *applu*, and *fma3d*) exhibit a growing prefetch coverage with increasing prefetch distance. This phenomenon occurs due to a favorable interaction between future execution and the stream prefetch engine. As

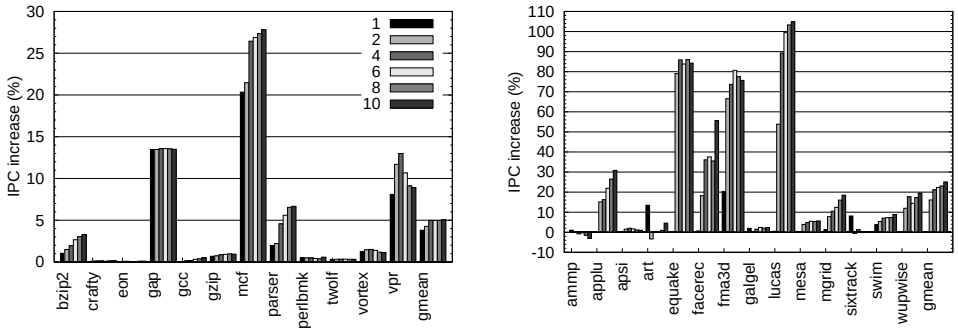

Figure 4.14: Sensitivity of future execution to the future value prediction distance

the prediction distance is increased, the memory requests issued by future execution indirectly increase the prefetch distance of the stream prefetcher and thus make the stream prefetches more timely. Therefore, fewer loads time out in the future core and fewer address-generating slices are invalidated, enabling more future load addresses to be computed. I found that prefetching more than 10 strides ahead does not improve FE performance.

On average, increasing the future prediction distance from 1 to 10 iterations ahead increases the geometric mean speedup for integer applications from 3.2% to 5%, while the IPC speedup for the floating-point applications increases from 11% to 30% over the baseline with aggressive stream prefetching. These results suggest that future execution may greatly benefit from a dynamic mechanism to adjust the prediction distance.

4.5 Summary

This chapter proposes the idea of *future execution* (FE), a data prefetching technique to hide the latency of cache misses using moderate hardware and no ISA, programmer, or compiler support. FE harnesses the power of a second core in a multi-core microprocessor to prefetch data for a thread running on a different

core of the same chip. Unlike previously proposed approaches for execution-based prefetching, this mechanism does not need any thread triggers, features an adjustable lookahead distance, does not use complicated analyzers to extract prefetching threads, requires no storage for prefetching threads, and works on legacy code as well as new code.

This chapter further evaluates the performance of the proposed prefetching technique. Overall, FE delivers a geometric-mean speedup of 5% in integer and 21% in floating-point programs over a baseline with an aggressive stream prefetcher. Furthermore, I demonstrate that future execution is complementary to runahead execution technique and the combination of these two techniques significantly raises the average speedup.

Finally, I study sensitivity of future execution to several architectural parameters, such as the minimum memory latency, the intercore communication delay/bandwidth, and the prefetch distance. The results demonstrate that prefetching based on future execution delivers robust performance improvements across many processor configurations. Moreover, the performance of future execution is very sensitive to future prediction distance which suggests that it may greatly benefit from a mechanism to dynamically adjust the prediction distance.

CHAPTER 5

EVENT-DRIVEN HELPER THREADING

This chapter discusses prefetching techniques based on outcome prediction and motivates the idea of the Event-Driven Helper Threading. Furthermore, it describes the architectural support for Event-Driven Helper Threading, its operation, and performance results. A sensitivity analysis of Event-Driven Helper Threading to several architectural parameters concludes this chapter.

5.1 Motivation

Over the past years considerable research effort has been put into developing hardware prefetching algorithms that are based on address prediction. In spite of the large number of interesting proposals, their introduction into commercial designs is often hampered by the considerable storage requirement for prediction tables, which would consume valuable chip area. In many cases, the proposed techniques are application specific and thus do not justify their implementation in a general-purpose microprocessor. Moreover, the specificity of such algorithms may well hurt the performance of other programs.

Having analyzed the limitations of previously proposed prefetching techniques, I decided to approach the central question of this dissertation from a different angle. Instead of devising new prefetching algorithms, I considered ways to alleviate these limitations by employing the execution capabilities of available cores in CMP architectures. This work resulted in a lightweight architectural framework to emulate prefetching algorithms via a special class of software helper threads. I call this technique *event-driven helper threading* (EDHT). The key principle behind EDHT is to use simple and fast hardware to expose information about individual

cache miss events to software. Thus, a special software helper thread executing on another core of a CMP can read this cache miss information and utilize it to implement various prefetching algorithms. This EDHT concept efficiently combines simple and fast hardware to communicate event data and flexible software to implement prefetchers of almost arbitrary complexity.

The EDHT framework allows multi-core processors to provide immediate benefits and presents a relatively simple yet effective architectural enhancement to exploit additional cores to speed up individual threads. Unlike previously proposed approaches for software prefetching, the EDHT mechanism can improve performance without the need to modify or analyze the original binary. Moreover, EDHT solves many problems that have hampered the introduction of complex hardware prefetching algorithms into commercial microprocessors. Specifically, EDHT needs minimal hardware modifications, does not require specialized hardware storage for prediction tables, and can be easily reconfigured to customize prefetching algorithms for individual applications.

Looking into the future, the EDHT concept opens opportunities for designing novel prefetching techniques. For example, the flexibility of a software approach provides an interesting possibility for the automatic generation of program-specific EDHT threads. Furthermore, EDHT allows the design of new prefetchers that dynamically adapt to the program behavior. Hybrid prefetchers that run several pre-fetching algorithms in parallel may also be possible. Finally, the EDHT framework makes it feasible to quickly prototype novel prefetching techniques on real hardware without the need to recompile applications, modify the silicon, or resort to slow simulations.

5.2 Implementation

Hardware prefetching algorithms based on outcome prediction are naturally decoupled from the execution of the target thread. The only data dependence between the target thread and the prefetching algorithm is the information about the cache miss address and the load instruction that caused it. As such, these prefetch mechanisms could be emulated by a software prefetching thread that is started whenever the target thread experiences a cache miss. However, a software implementation of a hardware prefetcher faces two obstacles. First, such prefetch threads would need to be spawned quickly on microarchitectural events as opposed to program events in the conventional multithreading paradigm. Second, complex prefetch algorithms, such as Markov prefetching, often require a large amount of state that needs to be stored somewhere.

To overcome these issues, it is necessary for a processor architecture to support explicit communication of microarchitectural events from one thread to another. Ideally, the target thread should not be aware of this communication so that prefetching can be easily turned on and off depending on the availability of hardware resources. To achieve this, I propose Event-Driven Helper Threading (EDHT), which provides a way for a low-latency, unidirectional event trigger transmission between regular and helper threads. EDHT threads execute on a conventional core of a chip multiprocessor and the state of the underlying prefetching algorithm is loaded into the private data cache of the core via the conventional memory hierarchy. The rest of this section explains the hardware support for EDHT and its operation.

5.2.1 Overview of Operation

Figure 5.1a shows a typical pointer chasing loop. Each iteration of the loop processes a node of a linked list and fetches the next node to be processed. Assuming that the linked list is not cache-resident, each access to the field `data` in the list's node will result in a cache miss and stall the processor. As shown in Figure 5.1a, this access translates into a single load instruction `data=pointer->data`. Markov prefetchers are a good fit for prefetching such linked list traversals since they can memorize previous traversals of this list and accurately predict the next node's location.

A typical hardware implementation of a Markov prefetcher will snoop the cache miss addresses coming out of the cache. For each observed cache miss, it will look up its prediction table similar to the one shown in Figure 2.3b and determine which addresses are likely to be referenced next. After that, it will issue prefetch requests for those addresses into the memory hierarchy in the hope that the program will soon request those data as well.

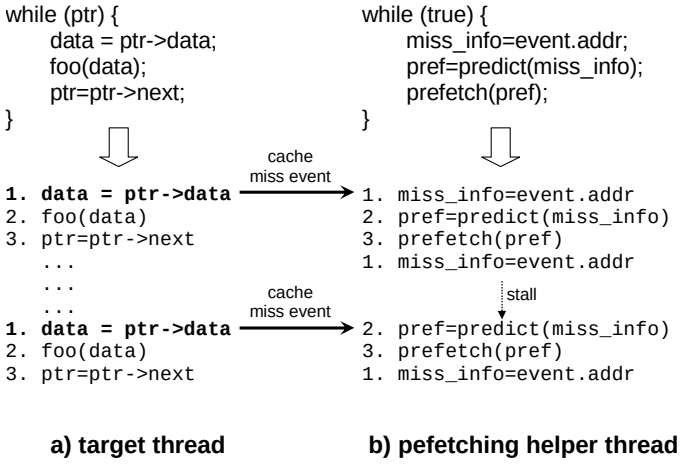

Figure 5.1: Code example

Figure 5.1b shows a conceptual prefetching thread that can emulate a prefetcher. The prefetching thread consists of an infinite while loop. In each iteration, the prefetching loop stalls waiting for cache miss event data to arrive from the target thread. When the target thread experiences such a miss, the prefetching thread loads the cache miss address along with the associated PC value and uses this information to run its prediction algorithm. Finally, it issues a prefetch instruction and loops back. If by this time there is a new miss event waiting in the event queue, it immediately executes the prefetching algorithm with the new data. Otherwise, it stalls until a new event arrives.

Of course, a dedicated hardware implementation will execute the prediction algorithm much faster than its software thread equivalent. Hence, there may be situations when the frequency of cache miss events outpaces the speed at which the prefetching thread can process them. In this case, limited buffering can be provided to store unprocessed events. The next subsection describes the organization of such a buffer and its interaction with other components of the CPU.

5.2.2 Implementation and Hardware Support

Even if the architecture provides a way to communicate architectural events to user-level threads, prefetching-based event-driven helper threading is likely to perform poorly on current hardware due to the following reasons. First, synchronization must occur via the operating system or via spin-locks. In the OS case, the trap and return time is so large that it can negate any performance advantage. In addition, the communication between the threads must occur via shared memory, increasing the contention for the shared cache ports and wasting dynamic

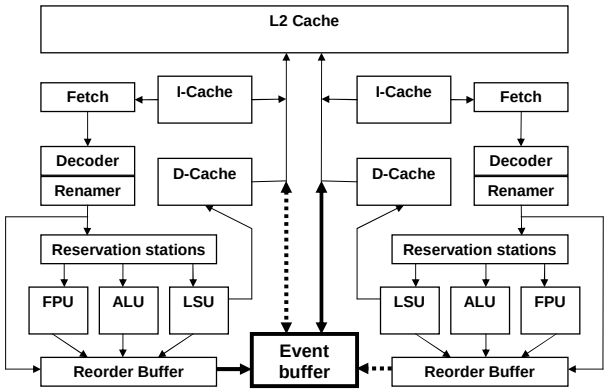

Figure 5.2: System Architecture

power. Thus, for EDHT-based prefetching to be successful, it is imperative that the implementation of the event delivery mechanism be efficient.

Since much of the problem related to the implementation of fast inter-thread communication is due to the use of shared memory, I propose to use an *event buffer*. Figure 5.2 shows the architecture of an out-of-order CMP that supports event-driven helper-threading. The event buffer and associated datapaths are highlighted in bold. The regular computation thread executes on the "left" core, while its EDHT prefetching thread is running on the "right" core. The event buffer is located between the two cores. It represents a FIFO structure of limited capacity. It receives information about cache miss events from the ROB of one of the cores. The ISA abstraction for the event buffer is an I/O device that can be accessed by program threads via I/O read instructions from a reserved address. The event buffer controller snoops the cache bus and supplies data for all event buffer I/O reads.

The instructions in the left core execute and commit normally. If a load instruction experiences a cache miss, a prefetch trigger is transmitted to the event buffer once the load commits. Event transmission is also triggered if a load instruction loads data that has been prefetched by the prefetching thread. The data that needs

to be transferred includes the instruction's PC value and the referenced memory address. Some prefetching mechanisms also require information on whether the event was triggered by a cache miss or a correctly prefetched cache miss.

In the meantime, the prefetch thread is running on the helper core. When the prefetching thread issues an I/O read instruction to obtain data about a cache miss event, it stalls waiting for a reply. When the event buffer receives this data, it supplies the received information to the stalling read request. Then the prefetching thread calculates the prefetch address based on the underlying algorithm and executes a non-binding prefetch load instruction that will fetch data into the shared memory hierarchy. After that, it loops back and issues another I/O read instruction to obtain information about the next architectural event.

The cache lines that are touched by the helper core are tagged. When the regular core references a tagged cache line, it marks the executed load instruction as the consumer of the prefetched data. When this load instruction commits, it causes a prefetch trigger to be transmitted to the helper core as if this instruction had experienced a cache miss. This mechanism essentially mimics the way traditional hardware prefetchers work and allows the helper core to stay ahead of the data consumption of the regular thread.

The operation of EDHT has the following implications on the operating system. EDHT threads and the main application run in the same address space and hence share a page table. EDHT threads do not use absolute code addresses (they are fully relocatable) and use a thread-private data area to avoid conflicts with the main thread. The OS scheduler should also be aware of the helper threads associated with each application and schedule them as a group to execute on cores that share at least one level of memory hierarchy. Note that, while I have presented

EDHT for two cores running a single main thread, the technique can be extended to support multiple cores by providing either additional event buffers between pairs of cores or the ability to communicate events from different cores to a centralized buffer.

5.2.3 Prefetching Algorithms

To demonstrate the efficiency and flexibility of the proposed architectural framework, I implemented a number of prefetching algorithms on it. This section presents a detailed description of these algorithms.

First, a conventional stride prefetcher is implemented. I evaluate both a local and a global stride prefetcher and name them *lstride* and *gstride*, respectively. The local stride prefetcher uses a table with 1K entries to keep track of strides on a per-load basis. The global prefetcher keeps a history of the 8 last miss addresses to identify address streams. Both kinds of prefetchers have 8 stream buffers and utilize an LRU replacement policy for buffer allocation.

Second, I implement two prefetching algorithms that are based on a third-order delta-correlation Markov model with a table size that fits into the L1 data cache. Unlike with stride prefetching, the algorithmic difference between the global and local versions of the Markov prefetcher is minimal. Therefore, I evaluate only local prefetchers. The first prefetching algorithm is based on the differential finite context method (*DFCM*) value predictor [15]. This algorithm uses a 2-bit counter-based confidence estimator in the first-level table to suppress low-confident predictions for load instructions that exhibit unpredictable behavior. The second algorithm is more similar to a conventional Markov model. It stores two distinct values (predictions) in each entry of its second-level table. Each value in the second-

level table is associated with a 1-bit confidence counter, which is incremented every time a particular prediction is observed to be correct. Thus, the confidence is associated with transitions in the Markov graph rather than with the predictability of individual load instructions. This algorithm uses a prefetch width of two and a prefetch distance of four, generating up to eight prefetch addresses on each invocation of the algorithm. I call this algorithm *Markov* prefetcher.

Finally, this work shows how the EDHT framework can be used to implement prefetching schemes based on very large Markov models. To this end, I implement a first order Markov prefetcher with address correlation containing 256K entries in its second-level table. Each entry contains two Markov graph neighbors. In addition to the delta for the next address, each table entry records the four deltas that last followed the most recent address in MRU order. This organization of the Markov prefetcher is similar to the replicated correlation prefetcher used by Solihin et al. [46]. Each value in the second-level table is associated with a 1-bit confidence counter, similar to the previously described *Markov* algorithm. When a table entry is accessed by the prefetcher, all addresses recorded in this entry are prefetched. Each second-level table entry is configured to fit into an L2 cache line. I call this algorithm *Correlation* prefetcher.

To emulate these prefetching techniques, I manually constructed five different prefetching threads. Table 5.1 summarizes the properties of these threads. The third column provides the number of instructions in each thread up to the first prefetch instruction. The stride prefetching mechanisms execute different instruction sequences depending on whether the event is associated with a cache miss or an access to a correctly prefetched cache line. The values in parentheses indicate the properties of a thread associated with the access to prefetched data. The

Table 5.1: EDHT threads for emulating hardware prefetching mechanisms

Prefetching algorithm	Description	Number of instructions	Load instructions	Longest dep. chain
global stride	8 simultaneous streams, 8-entry miss history buffer, prefetch distance of 8	52 (23)	12 (10)	15 (10)
local stride	1K-entry PC-indexed prediction table, prefetch distance of 8	18 (23)	4 (10)	6 (10)
DFCM	128-entry 3rd order L1 table, 2bc confidence, 16K-entry L2 table, select-fold-shift-xor (SFSX) hash function, prefetch distance of 8	26	6	7
markov	128-entry 3rd order L1 table, 8K-entry L2 table storing 2 distinct predictions, SFSX hash function, prefetch distance of 4	29	7	8
correlation	128-entry 1st order L1 table, 256K-entry L2 table storing 2 distinct sequences of 4 predictions, prefetch distance of 4	24	6	6

fourth column specifies the total number of load instructions, and the last column indicates the length of the longest instruction dependence chain. Note that the number of static and dynamic instructions for each prefetching thread up to the issue of the first prefetch is the same because all branches are removed from the code via loop unrolling and extensive use of conditional move instructions. Global stride prefetching requires the most instructions due to the large number of comparisons when the global history is searched for two repeating strides. The next section evaluates how well these prefetching threads work.

5.3 Evaluation Methodology

The performance of the EDHT framework is evaluated using an extended version of the SimpleScalar v4.0 simulator [27]. The baseline is a two-way CMP consisting of two identical four-wide dynamic superscalar cores that are similar to an Alpha 21264. In all modeled configurations it is assumed that one of the cores in the CMP can be used for executing a prefetching thread. The full description of the

baseline architecture and benchmark suite is provided in Chapter 3.

The event buffer can hold information about up to 20 recent cache misses. The communication latency between each core and event buffer is 5 cycles. All evaluated prefetching algorithms monitor L2 cache misses and issue prefetch requests for the L2 cache only.

The operation of a hardware stride prefetcher used in the chapter differs from the the operation of the stride prefetcher in Chapter 4. In my prior work, the stream prefetcher was activated on cache misses or when the prefetched cache blocks were accessed by a processor. My further research on prefetcher activation policies showed that the performance of a stream prefetcher can be significantly improved if it is activated on accesses to all cache blocks that were touched by the stream prefetcher, whether or not they resulted in useful prefetches. Thus, this and subsequent chapters of this dissertation use a more powerful version of the hardware stride prefetcher.

5.4 Experimental Results

This section experimentally measures the effectiveness of the proposed mechanism. In Section 5.4.1, I evaluate the performance of various prefetching schemes based on EDHT and compare the speedups with those of hardware implementations of the same prefetching mechanisms. In Section 5.4.2, I demonstrate how EDHT-based prefetching can improve single thread performance by combining hardware stride prefetching with Markov EDHT prefetching. Finally, Section 5.4.3 illustrates how EDHT prefetching compares to two other previously proposed hardware techniques that use extra cores to speed up single threads.

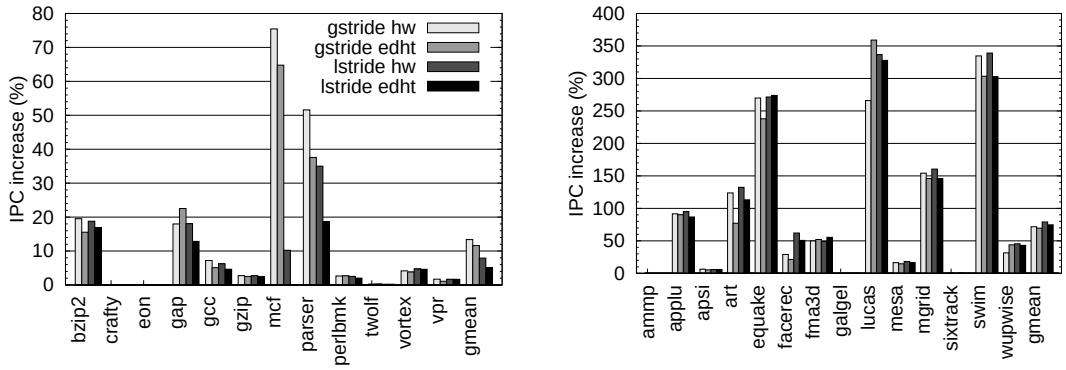

Figure 5.3: Performance of hardware prefetchers and their EDHT counterparts for stride prefetchers

5.4.1 Prefetching Emulation Performance

This section evaluates and compares EDHT-based prefetchers with their conventional, hardware-based counterparts. The baseline machine for this experiment is described in Table 3.1. I measure the performance of four different prefetching schemes: global stride prefetching (*gstride*), local stride prefetching (*lstride*), differential finite-context method prefetching (*dfcm*), and Markov prefetching (*markov*). The hardware implementations of each scheme are marked with a *hw* identifier, while the EDHT versions have an *edht* identifier after the name of the prefetching algorithm. Figure 5.3 presents speedups for the stride prefetchers and Figure 5.4 demonstrates results for the DFCM and Markov algorithms. The performance for the integer programs is shown in the left panel, floating-point application in the right panel.

The results show that the prefetching techniques used in this study are very effective, attaining significant speedups for the majority of the programs. When the hardware implementations are compared with the EDHT implementations, we find that hardware does outperform the software helper threads, but the performance gap is not large. In case of *gstride* prefetching, hardware provides 13% speedup on

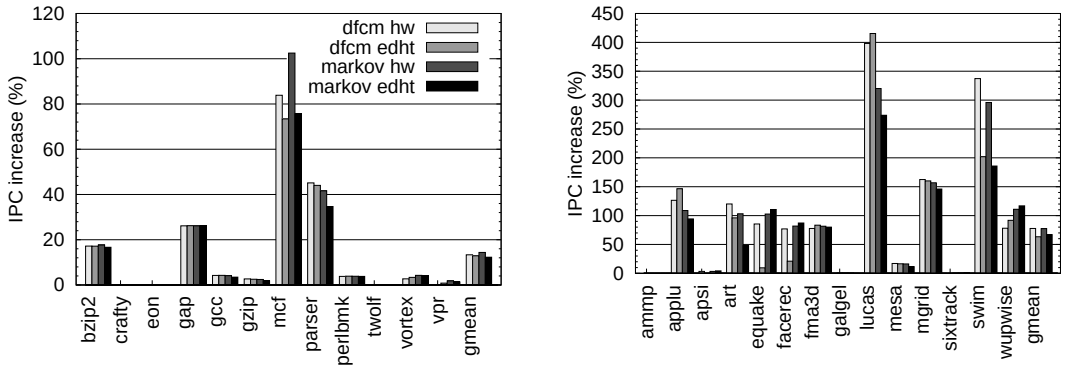

Figure 5.4: Performance of hardware prefetchers and their EDHT counterparts for dfcm and markov prefetchers

the integer applications and 72% speedup on the floating-point programs, while the EDHT implementations achieve 11% and 70% average speedup on the integer and floating-point programs, respectively. In case of *lstride* prefetching, the difference between the hardware and EDHT implementations also amounts to two percentage points. The main reason for the performance difference is the delayed issue of the prefetch requests by the helper thread. Helper threads are triggered only when delinquent load instructions commit and they take longer to compute and issue prefetches. Hardware prefetchers initiate the prefetching algorithm as soon a delinquent load instruction issues and generate prefetch requests much faster.

In case of Markov and DFCM prefetchers, there is almost no difference between hardware and EDHT in integer programs, but the difference is significant in floating-point applications. The high frequency of cache miss events in floating-point applications is the main reason for the decreased speedup. In case of *dfcm* prefetching, the high frequency of cache misses in *equake* and *facerec* causes 86% and 60% of the cache miss events to be dropped. EDHT *markov* prefetching faces a similar problem with *facerec* and *swim*.

Surprisingly, some programs exhibit higher speedups with EDHT prefetching.

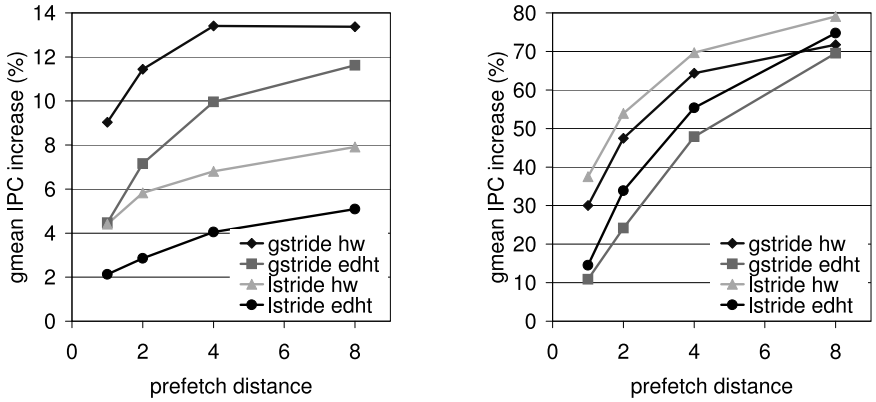

Figure 5.5: Sensitivity to prefetch distance for stride prefetchers

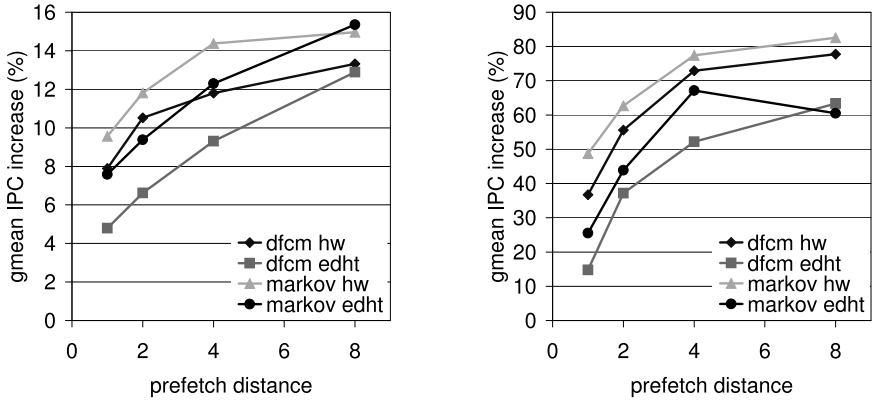

Figure 5.6: Sensitivity to prefetch distance for DFCM and Markov prefetchers

Hardware prefetchers observe cache misses in the issue order of the load instructions, while EDHT threads observe the sequence of cache miss events in commit order. Commit order allows to get a more precise cache miss history. In addition, EDHT threads do not suffer from cache miss history pollution caused by wrong-path loads.

It might be unexpected that the performance of the hardware and software implementations of these prefetching algorithms differs by so little. After all, hardware prefetches are issued at least 400 cycles earlier than software prefetches. It seems unlikely that such a big delay would have so little impact of performance.

Figures 5.5 and 5.6 explains this phenomenon. They illustrate the average

speedup obtained by the prefetching schemes as the prefetch distance is varied from 1 to 8. In each figure, average speedup for integer programs in shown on the left panel, floating-point programs on the right panel. The most interesting feature of this graph is how the performance difference between each hardware and EDHT pair decreases with increasing prefetch distance. For example, with a prefetch distance of 1 the average speedup for a hardware global stride prefetcher is 9% in intgeger and 30% in floating-point applications. The corresponding EDHT prefetcher achieves only 4.5% and 10% speedup in integer and floating-point programs, respectively. However, prefetching with higher distances decreases the relative performance gap. At a prefetch distance of 8 both hardware and EDHT prefetching perform almost the same in relative terms. Higher prefetch distances provide timelier prefetches and at some point it does not matter whether prefetch requests are issued with a 400-cycle delay or not. They are still issued early enough to mask the full memory latency. Therefore, a high prefetch distance is the key to good performance of software prefetching.

Interestingly, a *markov edht* configuration experiences a decrease in average speedup when prefetch distance in larger than 4. This is caused mainly by a significant drop in the performance improvement for *swim* program. As it was mentioned earlier, *swim* has an exceptionally high frequency of cache misses. With increasing prefetch distance, Markov EDHT needs to process more instructions per cache miss event and at some point it is not fast enough to process all events in the event buffer. This results into in a large number of cache miss events dropped from the buffer and, consequently, decreased performance.

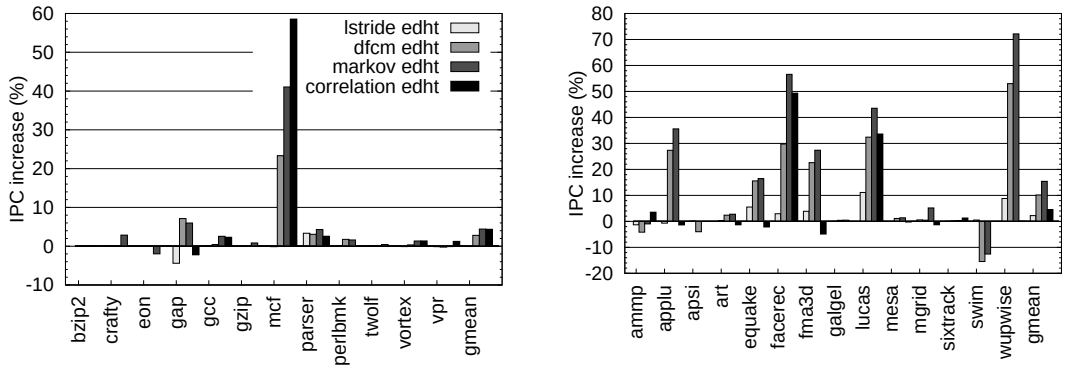

Figure 5.7: Speedup provided by different EDHT prefetching mechanisms over a baseline with hardware stride prefetching

5.4.2 Combining Hardware and EDHT Prefetching

Current high-performance microprocessors already include some form of stride prefetching. More complicated prefetching schemes are typically not implemented because of algorithm complexity and/or large storage requirements. EDHT offers an attractive alternative implementation of complex prefetching schemes. In this section, I investigate how hardware prefetching can be combined with a more complex prefetching scheme implemented as an EDHT thread. The base machine for this experiment includes the hardware global stride prefetcher described in Table 5.1. We measure the performance of four different EDHT prefetching schemes: local stride prefetching (*lstride*), differential finite-context method prefetching (*dfcm*), Markov prefetching (*markov*), and duplicated correlation prefetching (*correlation*). Figure 5.7 shows the program speedups relative to the *gstride hw* baseline.

The results show that adding a local stride prefetcher in most cases provides little additional benefit. Only *lucas* and *wupwise* experience a significant (over 5%) speedup. On the other hand, *markov* EDHTs deliver significant speedups for eight out of the 26 SPEC CPU2000 programs used in the study. The performance of the *dfcm* prefetcher is similar to *markov*, but is generally slightly lower. The

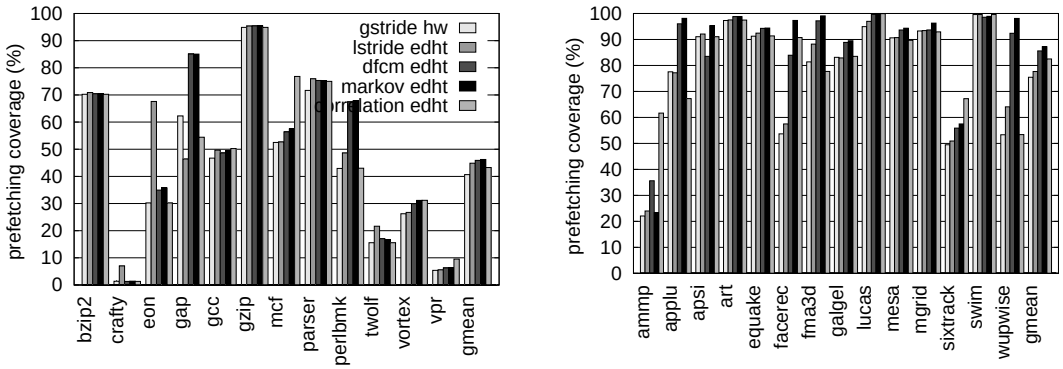

Figure 5.8: Prefetching coverage

correlation prefetcher performs well for three programs. It is especially successful with *mcf*, where it significantly outperforms all other prefetching algorithms. In integer applications, on average local stride prefetching improves performance by 0%, DFCM by 2.8%, markov by 3.4%, and correlation by 4.4%. In floating-point programs, local stride prefetching improves performance by 2%, DFCM by 10%, markov by 15%, and correlation by 5%.

To gain additional insight about the prefetching activity, I measured the prefetch coverage of the different algorithms. In this work, prefetch coverage is defined as the ratio of the total number of useful prefetches to the total number of L2 cache misses incurred by an application. Figure 5.8 illustrates that the speedup numbers of each algorithm largely correspond to their prefetch coverages. The *markov* EDHT is very successful on all SPECfp programs where it pushes the prefetch coverage above 90% for all but two programs. The correlation prefetcher is at its best on *mcf*, where it increases the prefetch coverage from 52% to 78%. The integer programs prove to be tough targets for prefetching. Programs *gap*, *mcf*, and *perlbmk* are the only integer programs that experience a significant coverage increase.

Overall, this section demonstrates that the combination of hardware stride prefetching and the more complex prefetching mechanisms implemented in our EDHT framework can yield significant performance improvements for a wide variety of programs. Both approaches exhibit significant synergy, as the hardware prefetcher detects and prefetches simple patterns, causing only undetectable patterns to be exposed to the more complex software prefetching algorithms. In addition, some algorithms suit applications much better than others. This further justifies the use of EDHT-based prefetching as prefetching threads can easily be customized on a per-program basis.

5.4.3 Comparison with Other Multi-Core Prefetching Techniques

The previous subsections showed that emulating hardware prefetchers as EDHTs is quite effective and provides significant speedups over the baseline with or without a hardware stride prefetcher. In this section, I compare the EDHT mechanism with two other hardware-only techniques that use an extra core of a CMP to speed up single threads.

First, I consider the case of Future Execution (FE) [13], the prefetching technique that was presented and evaluated in the previous chapter. FE dynamically creates prefetching threads by directing a copy of the stream of committed instructions to a helper core. On the way to the helper core, a value predictor modifies this stream to compute the results that these instructions are likely to produce during their n^{th} next dynamic execution. Executing this modified instruction stream on another core computes predictions for the future data addresses and issues prefetches into the shared memory hierarchy. In this study, the FE mechanism is

supplied with a 4K-entry stride-two-delta hardware value predictor and a buffering capacity of 100 instructions between the cores.

Second, I evaluate the performance of the dual-core execution (DCE) architecture [52]. Instead of using the idle core to run specialized prefetching threads, this technique uses it to launch and execute a copy of the original program in runahead mode [33]. This runahead thread attempts to follow the program path and to execute all instructions that are not dependent on the load instructions that miss in the cache. Thus, it effectively extends the instruction window and allows to issue load requests for data that may be needed in the near future. The non-speculative core re-executes all instructions committed by the runahead core and makes sure that the program execution stays on the correct path. We implement a variation of DCE with a 2K-entry result queue between the cores, a 4KB runahead cache, and an optimistic 1-cycle latency to copy the architectural state between the cores.

Note that both FE and DCE impose much higher hardware requirements and complexity than the EDHT framework. FE needs a prediction table and a high-bandwidth communication link between the cores. DCE requires hardware support for a large result queue, a runahead cache, and misprediction recovery logic. Both FE and DCE also require special multiplexing support to fetch instructions from another core (in addition to from the instruction cache).

Figure 5.9 shows the speedup of the different techniques relative to the *gstride hw* baseline. I chose the *markov edht* algorithm to represent EDHT prefetching since it performs best on average. Out of the 26 programs used in this study, EDHT performs best on four programs and DCE provides a significant performance lead (of over 5%) on five. In integer applications, all three prefetching approaches deliver a similar average speedup of about 4%. In floating-point programs, the

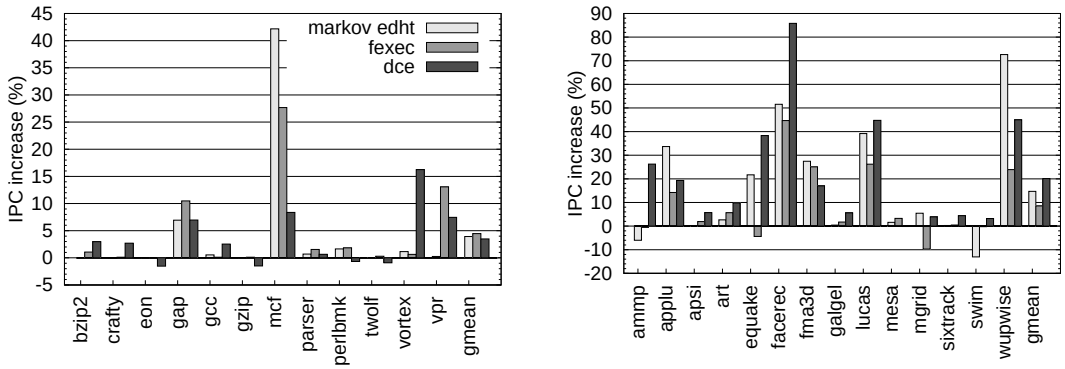

Figure 5.9: Speedup provided by different multi-core prefetching mechanisms over a baseline with stride prefetching

EDHT Markov prefetcher delivers 15% speedup, the FE technique improves performance by 9%, and DCE shows the best average speedup of 20%. Note that the performance improvement provided by the FE technique differs from the results demonstrated in Chapter 4. As explained in Section 5.3, this difference emerges from the use of a more advanced hardware stride prefetcher in the baseline processor configuration.

One of the explanations for DCE's comparatively good performance in the integer programs is its ability to significantly decrease the penalty of the branch mispredictions observed by the target thread. Both EDHT prefetching and Future Execution can only reduce load latencies. In floating-point programs, DCE often provides more timely prefetch requests, even though its prefetch coverage is generally somewhat lower than that of EDHT (e.g., *facerec*). This deficiency can be mitigated by dynamically increasing the prefetch distance of the *markov edht* prefetching thread. The investigation of this approach is left for future work.

Figures 5.10 and 5.11 provide additional insight about the operation of the evaluated techniques. Figure 5.10 shows the total increase in the number of issued instructions compared to single-core execution. For almost all programs EDHT

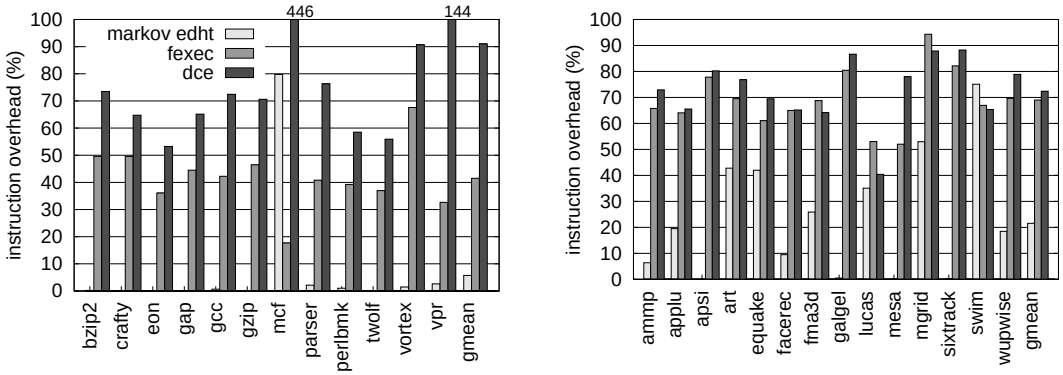

Figure 5.10: Instruction overhead

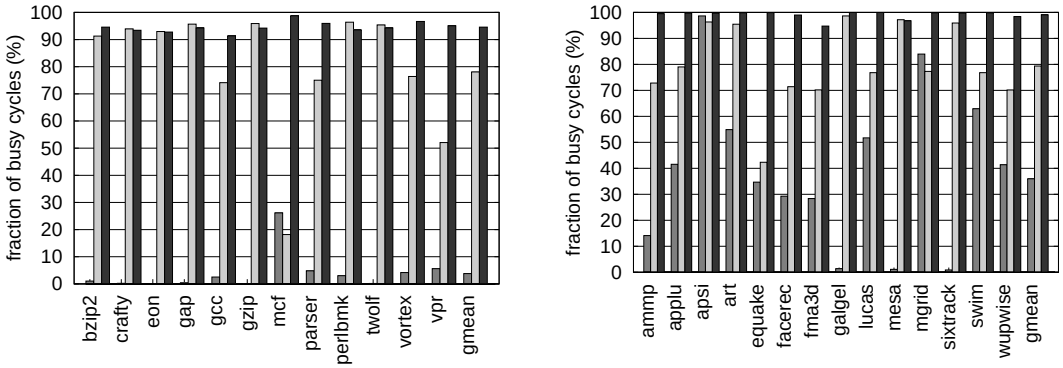

Figure 5.11: Helper core occupancy

executes the least number of instructions. The only exceptions are *mcf* and *swim*, which are explained by an exceptionally high frequency of cache misses. The FE overhead varies from 18% to 95%, while DCE in two cases increases the total number of executed instructions by more than a factor of two (*mcf* and *vpr*). This happens mainly due to fetching and executing many instructions along a mispredicted branch path. DCE always executes more instructions than EDHT or FE on all but five floating-point programs. In integer applications, the DCE technique executes about 90% extra instructions, compared to 41% for future execution and 5% for *markov* EDHT. In floating-point programs, DCE and FE execute 72% and 70% more instructions, respectively, compared to a 21% instruction overhead for EDHT.

Figure 5.11 estimates how often the helper core is busy. In case of EDHT, it is assumed that the helper core is idle if it is fully stalled waiting for the next cache miss event to occur. For FE and DCE, the idle periods correspond to the cycles when the helper core's ROB is empty. DCE keeps both cores active almost all the time with an average occupancy of 95% in integer and 99% in floating-point programs. FE is less demanding and the helper core occupancy, resulting in an average occupancy of 79%. EDHT prefetching represents the lightest load on the helper core by activating it on average only 4% and 36% of the time in integer and floating-point programs, respectively. These results highlight another strength of EDHT-based helper threading. By virtue of the event-driven thread communication mechanism, helper threads become active for a short time only when a cache miss occurs. The other two hardware techniques are active all the time or/and have no fast way of exiting helper mode.

The results in this section demonstrate that prefetching based on EDHT can provide performance improvements that are on par with those provided by the dual-core execution paradigm and future execution. At the same time, it requires considerably less complex hardware support and executes fewer instructions, thus keeping the helper core available for other tasks. For example, in a CMP with more than two cores, several EDHTs could time-share on one helper core to prefetch for the regular computation threads running on the other cores. Moreover, the results suggest that EDHT is considerably more energy-efficient than other dual-core prefetching techniques. The energy-efficiency is further investigated in Chapter 6.

5.5 Summary

This chapter explores the idea of exploiting available cores on a chip multiprocessor to improve the performance of individual program threads. I propose to use extra cores to execute prefetching threads that can emulate the behavior of complex outcome prediction-based prefetching algorithms. However, for this threading technique to be effective, a low overhead mechanism for communicating microarchitectural events is required. To accomplish this, this chapter presents the *event-driven helper threading* (EDHT) framework, which uses lightweight hardware support for efficient event communication. EDHT solves many problems that have hampered the introduction of complex outcome-prediction prefetching algorithms into commercial systems. Specifically, my scheme needs minimal hardware modifications, does not need specialized hardware storage for prediction tables, and can be easily reconfigured to tailor prefetching algorithms for individual applications.

The performance analysis reveals that EDHT-based prefetching provides essentially the same speedup as pure hardware implementations of prefetching algorithms. I further demonstrate that running prefetching EDHTs on top of a baseline with a hardware stride prefetcher yields speedups between 5% and 100% on a wide range of programs. Finally, I compared EDHT with two other hardware techniques for multi-core execution and show that even without customization, EDHT prefetching can provide competitive performance improvements while executing fewer instructions and requiring considerably simpler hardware support.

CHAPTER 6

IMPROVING THE ENERGY-EFFICIENCY OF MULTI-CORE PREFETCHING

This chapter presents the evaluation of the energy consumption of the previously presented multi-core helper threading techniques. Furthermore, it proposes a set of techniques that improve the energy-efficiency of the individual helper execution designs. A comparative evaluation of energy-efficient versions of Future Execution, Event-Driven Helper Threading, and Dual-Core Execution concludes this chapter.

6.1 Motivation

The previous chapters of this dissertation have discussed the performance of various prefetching techniques only in terms of execution time. However, the power consumption of high-performance microprocessors has recently become a high-priority concern for computer architects. New generations of microprocessors are now designed to deliver maximum performance within a restricted power budget. As a result, microprocessor designers can no longer ignore the power implications of new performance-enhancing architectural techniques.

Taking these concerns into account, I decided to analyze the impact of different prefetching techniques on energy consumption. In particular, I it is important to quantify the energy overhead associated with using two cores instead of one for the execution of a single thread. All previously discussed multi-core prefetching will incur energy overhead due to the extra instructions that need to be executed on a core devoted to prefetching technique. Moreover, the number of extra instructions executed is likely to be different for each of the prefetching techniques presented previously. For example, the execution of EDHTs is determined by the frequency

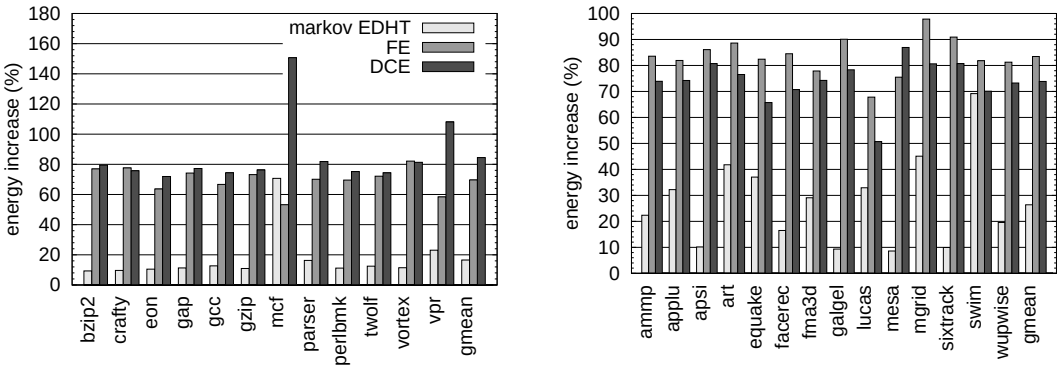

Figure 6.1: Increase in energy consumption compared to single-core execution

of cache misses. Thus, the instruction overhead of the EDHT technique is directly proportional to the number of cache misses. In case of FE, the prefetching thread is generated by transforming the committed stream of instruction. Therefore, the overhead is directly proportional to the number of register-writing instructions *committed* by the non-speculative core. The overhead of the DCE technique is proportional to the number of *executed* instructions that would be executed by a program thread in a single-core mode.

This chapter evaluates the energy consumption using the Wattch framework [1] for architectural-level power analysis. In our experiments, we use 70nm technology with a clock frequency of 4 GHz and aggressive conditional clocking. The conditional clocking style assumes that 5% of maximum dynamic energy is consumed when a particular module is disabled.

methodology in greater detail.

Figure 6.1 presents the energy overhead of the different multi-core prefetching techniques relative to a single-core processor with hardware stride prefetching.

EDHT incurs the smallest energy overhead for all SPEC programs but *mcf*. On average, integer programs are executed with 17% energy overhead, while floating-point programs experience 26% energy overhead. The *mcf* and *swim* programs

incur the highest energy overhead of 70% due to an extremely large number of cache misses. Both the FE and DCE techniques consume much more energy than EDHT. Interestingly, FE incurs less overhead than DCE on integer applications (on average, 70% for FE vs. 85% for DCE), while DCE is more energy-efficient on floating-point applications (on average, 74% for DCE vs. 84% for FE).

DCE performs worse than FE on integer applications mainly due to the large number of mispredicted branches in the front core. The DCE technique effectively extends the instruction window by the size of the result queue, which significantly raises the number of instructions executed on the mispredicted path. This scenario is especially dominant in *mcf* and *vpr*, where DCE incurs the highest overhead of 150% and 108%, respectively. Since floating-point programs have more predictable control flow, there are fewer branch mispredictions. As a result, DCE has less energy overhead in floating-point programs than FE.

The results in Figure 6.1 demonstrate that the EDHT technique is significantly more energy-efficient than both FE and DCE. However, the high overhead of these techniques can potentially be decreased by using relatively simple architectural enhancements to the original designs. In the following sections, I will present techniques for improving the energy-efficiency of previously discussed helper threading techniques and evaluate their effectiveness.

6.2 Energy-Efficiency Techniques for Future Execution

As discussed above, the overhead of Future Execution is directly proportional to the number of instructions *committed* by the non-speculative core. However, many programs in the SPEC CPU2000 benchmark suit do not experience many cache misses and cannot be accelerated any prefetching technique. Therefore, there is no

need to executing an FE thread on a CMP core if the program execution thread experiences very few cache misses.

To address this issue, I propose to adaptively switch FE on and off depending on the dynamic behavior of the target thread. adaptive mode switching. The algorithm for adaptive mode switching computes the L2 cache miss rate for every one million committed instructions and turns FE mode off if the cache miss rate is below 2.5 misses per 1000 committed instructions. The parameters in the algorithm (i.e., the miss rate thresholds) are empirically determined to provide the best average performance for the studied workloads.

I have also noticed that instructions with certain opcodes essentially never participate in the computation of load addresses. For example, there is a little chance that a floating-point *add* instruction is used in the computation of a load address. Therefore, such instructions can be safely excluded from the future execution instruction stream. Following this intuition, I have profiled all applications to find out instruction opcodes that participate in the computation of load addresses.

Based on this finding, I propose to augment the Future Execution technique with an additional instruction filter. This filter needs to monitor the opcodes of the instructions committed by the non-speculative core. When the filter detects an instruction that belongs to the category of unimportant opcodes, it simply removes this instruction from the committed instruction stream. The filtered instruction stream further proceeds to the value predictor, where it undergoes the transformations described in Section 4.2.

Figures 6.2 and 6.3 present the effect of these techniques on the energy consumption and IPC, respectively. Baseline future execution without any energy-saving techniques is represented by the *baseline FE* bar. The next configuration

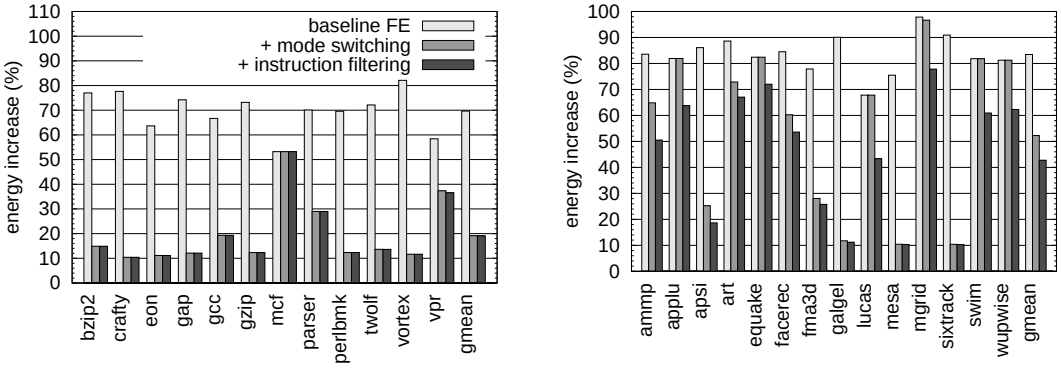

Figure 6.2: Energy overhead of baseline Future Execution configuration and two energy-efficient configurations

adds dynamically switching of FE mode on and off depending on the frequency of cache misses (*+ mode switching*). The third bar (*+ instruction filtering*) combines the dynamic switching and the unimportant instruction filtering techniques.

Figure 6.2 demonstrates that mode switching is very effective at reducing the energy overhead. On average, it reduces the overhead from 70% to 19% for integer and from 84% to 53% for floating-point programs. Most integer applications have very few cache misses and, therefore, gain a larger benefit from mode switching than the more memory-bound floating-point applications. Instruction filtering is quite effective on floating-point programs. When instruction filtering is added to mode switching, the energy overhead of floating-point applications decreases from 53% to 43%. Integer applications execute very few unimportant floating-point instructions and, therefore, do not benefit from instruction filtering.

Figure 6.3 shows the effect of the energy-saving techniques on the execution speedup provided by Future Execution. While instruction filtering has very little impact on the IPC increase, three programs (*gap*, *vpr*, and *fma3d*) experience a significant performance degradation when FE mode switching is introduced. This degradation occurs mainly due to the omitted prefetching opportunities when the

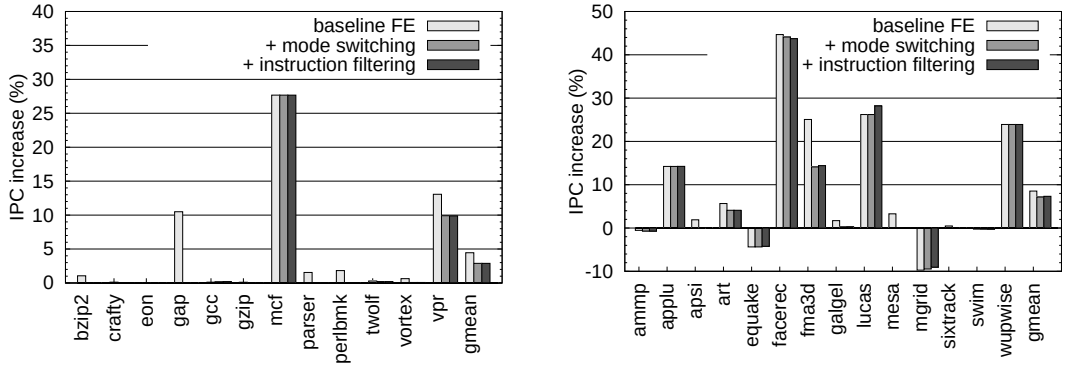

Figure 6.3: IPC increase provided by the baseline Future Execution configuration and two energy-efficient configurations

target thread experiences a lot of cache misses, but FE is turned off. This problem is largely an artifact of the last-interval cache miss frequency prediction policy. In cases of bursty cache miss behavior, the frequency of misses in consecutive time intervals is not well-correlated. Thus, it may be possible to devise techniques that can predict which intervals of 1 million instructions are likely to have a lot of cache misses. The investigation of this idea is left for future work.

6.3 Energy-Efficiency Techniques for Event-Driven Helper Threading

Figure 6.1 demonstrates that the baseline version of Event-Driven Helper Threading is significantly more energy-efficient than other helper threading techniques. Nevertheless, I developed two approaches that allow to further reduce EDHT's energy overhead. Both of these techniques are based on the observation that the number of instructions executed by EDHT threads is directly proportional to the number of events passing through the event buffer. Therefore, the most straightforward way to reduce the number of executed instructions is to decrease the number of generated events.

As described in Section 5.2, the baseline EDHT configuration triggers events in the following two cases. First, an event is generated if a load instruction is committed that experienced a cache miss. The second case involves the commitment of a load that hit in the cache, but the cache block accessed by this load was previously accessed by an EDHT prefetch instruction. Note that when an EDHT prefetch instruction accesses a cache block, it marks it with an EDHT tag independent of whether it was a cache hit or miss.

The energy-efficient techniques that I propose modify the cache block tagging policy for EDHT prefetch instructions that hit in the cache. The original policy marks cache blocks with an EDHT tag for all EDHT prefetches, even if a prefetch instruction hit in the cache. This, however, may generate many events that do not help the prefetching activity. Instead of being invoked only for data patterns that miss in the cache, such a tagging policy may invoke EDHT events for all predictable address streams, whether they hit or miss in the cache. As a result, extra EDHT instructions are executed and more energy is consumed than necessary. However, a stream prefetcher could prefetch those cache blocks in advance of EDHT prefetches. Such prefetched addresses would still belong to important address streams that miss in the cache.

Based on these observations, I propose two techniques to improve the energy-efficiency of EDHT. In the first technique, I suggest to modify the policy of marking cache blocks with EDHT tags in case of cache hits. I propose to mark blocks that hit in the cache only if those blocks are already marked with a stream prefetcher tag. Second, I suggest to completely remove the policy of tagging cache blocks that already reside in the cache.

Figures 6.4 and 6.5 present the effect of the proposed techniques on the energy

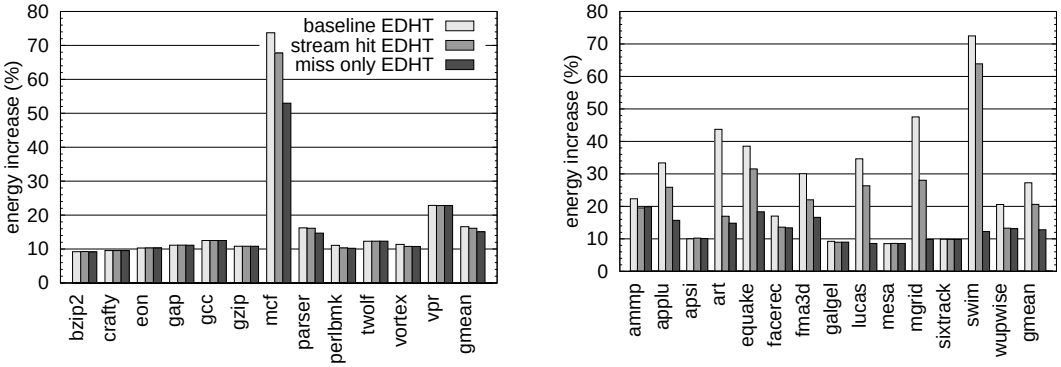

Figure 6.4: Energy overhead of the baseline Markov EDHT configuration and two energy-efficient EDHT versions

consumption and IPC, respectively. The baseline EDHT implementation without any energy-saving technique is represented by the *baseline EDHT* bar. The next configuration marks cache blocks that hit in the cache with an EDHT tag only if those blocks are already tagged with a stream prefetcher tag (*stream hit EDHT*). The third bar (*miss only EDHT*) represents the configuration that completely removes the policy of tagging cache blocks that already reside in the cache.

Figure 6.4 shows that the proposed techniques are relatively efficient at reducing the energy overhead of memory-bound applications. In floating-point programs, the first tagging policy brings the overhead down from 27% to 21%, while the second tagging policy brings the overhead down to 13%. The second policy is especially successful since many floating-point applications are dominated by strided address streams. The second policy avoids trigger generation on the accesses to the data prefetched by the stream prefetcher, therefore reducing the total number of cache miss events observed by the EDHT mechanism. In integer applications, only *mcf* significantly benefits from the modified tagging policies.

While the proposed policies are effective at reducing the energy overhead, Figure 6.5 demonstrates a significant drop in execution time speedup. In case of the

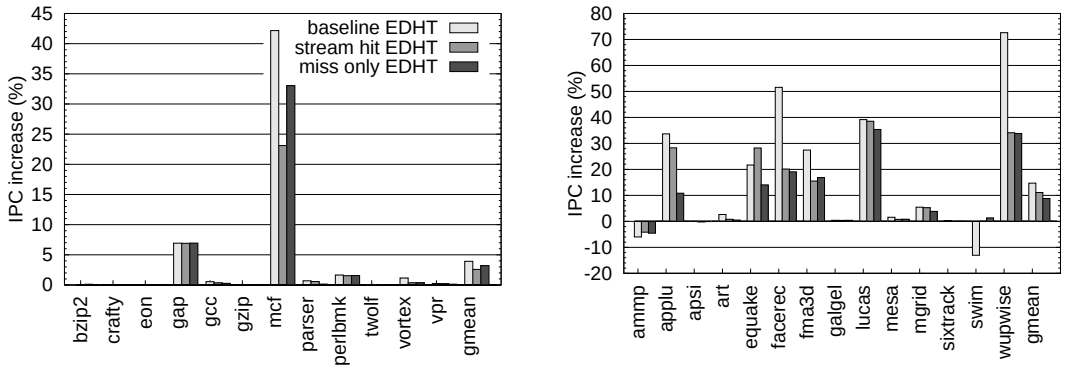

Figure 6.5: IPC increase provided by the baseline Markov EDHT configuration and two energy-efficient EDHT versions

first policy, the penalty is especially significant in *mcf*, *facerec*, *fma3d*, and *wupwise*. The main reason for this performance penalty is the reduction in prefetching coverage. Since some of the values from the address stream are never propagated to the prediction algorithm, the history of the pattern gets obfuscated. Thus, it is harder for the markov prediction algorithm to correctly predict values for the next load addresses. At the same time, two applications perform better with the first tagging policy than with the original baseline EDHT configuration. In *equake*, this happens due to fewer dropped triggers. In *swim*, the extra performance benefit is explained by the reduced interference of EDHT prefetches with the stream prefetcher.

The same reason causes the performance to drop even further in case of the second tagging policy. On average, the first tagging policy reduces IPC increase from 3.9% to 2.6% in integer applications and from 15% to 11% in floating-point programs. In case of the second tagging policy, integer programs obtain a 3.2% speedup and floating-point programs exhibit 9% speedup.

6.4 Energy-Efficiency Techniques for Dual-Core Execution

The energy overhead of the Dual-Core Execution technique comes from two main sources. First, all instructions that execute in the front core have to be re-executed in the back core. The second reason is the large number of fetched and executed instructions that follow branch mispredictions that are not resolved in the front core. Since the DCE technique forms a very large instruction window by concatenating the ROBs of the two cores and the result queue, the maximum number of instructions that are fetched after a mispredicted branch is much higher than in a single core. The authors of DCE proposed several techniques to attack these sources of energy inefficiency [49]. This section describes these techniques in greater detail and evaluates their effect on the energy consumption.

First, the authors propose to dynamically turn DCE on and off depending on the frequency of cache misses in the front core. The algorithm is similar to the one I used for Future Execution. It computes the L2 cache miss rate for every one million committed instructions and turns DCE mode off if the cache miss rate is below 2.5 misses per 1000 committed instructions. The original version of the algorithm is more complicated and uses multiple cache miss rate thresholds depending on the branch misprediction rate. I experimented with these versions and found it to perform only marginally better than a single-threshold scheme. Therefore, I used the same algorithm as for future execution.

The second technique attempts to reduce the number of instructions that are fetched and executed after mispredicted branches that are unresolved in the front core. The authors suggest to resize the result queue (RQ) depending on the branch misprediction rate in the back processor. It computes the back-end branch misprediction rate for every million committed instructions. If the branch misprediction

rate in the back core is less than 0.6 per 1000 committed instructions, the result queue size is set to 128 entries. If the misprediction rate is between 0.6 and 0.3, the RQ size is set to 256 entries, between 0.6 and 0.15 to 512 entries, and below 0.15 mispredictions to 1024 entries.

The third technique further attacks the problem of execution overhead after mispredicted branches. The authors suggest to prevent invalidations of traversal loads (loads that load an address to be consumed by the same static load) in the front core. In case of pointer chain traversals, invalidating such loads can result in invalidating a lot of dependent instructions, including branches. Thus, preventing such invalidations can potentially reduce the number of instructions fetched and executed along an incorrect program path.

The final energy-saving technique proposed by the DCE authors involves preventing re-execution of the instructions that produced correct results in the front core. This involves the introduction of special mechanisms to re-execute only load instructions and compare the loaded values with the values loaded by these loads in the front core. The other instructions are transformed into "move immediate" instructions of the pre-computed values into the result registers. Since this technique significantly raises the design complexity by requiring the modification of the original DCE operation and the introduction of a new recovery mechanism, I decided not to include into my evaluation. In addition, executing "move immediate" instructions is only marginally more energy-efficient than executing the original version of the instructions.

Figures 6.6 and 6.7 present the effect of the three implemented techniques on energy consumption and IPC, respectively. The baseline DCE implementation without any energy-saving techniques is represented by the *baseline DCE* bar.

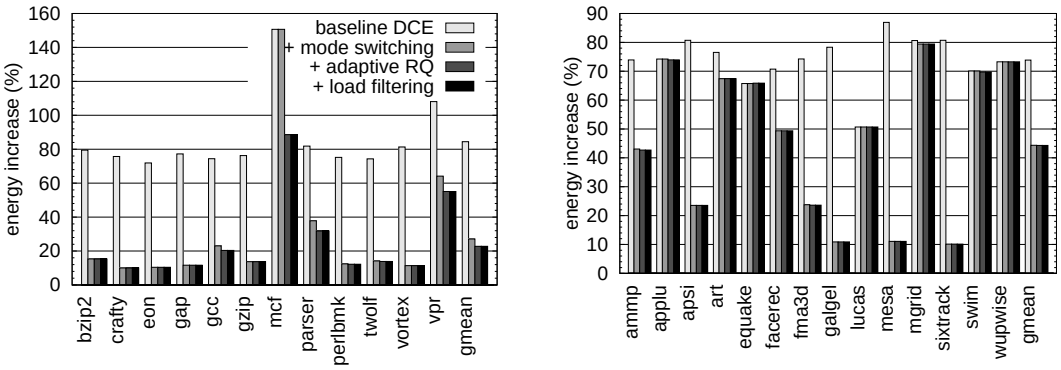

Figure 6.6: Energy overhead of the baseline Dual-Core Execution configuration and three energy-efficient DCE configurations

The next three configurations cumulatively add energy-efficient techniques. The first configuration, represented by the second bar, adds switching DCE mode off and on depending of the L2 cache miss rate (*+ mode switching*). The third bar (*+ adaptive RQ*) represents the configuration that adds dynamic resizing of the results queue. Finally, the fourth bar (*+ recursive load filtering*) adds preventing recursive load invalidations.

Figure 6.6 shows that dynamic switching of the DCE mode has the biggest impact on reducing the energy overhead. On average, mode switching lowers energy overhead for integer applications from 84% to 27%, while floating-point programs experience a reduction from 74% to 44%. Adaptive result queue sizing further reduces the energy overhead to 23% for integer applications, but has no impact for floating-point programs. Preventing recursive load invalidation did not result in a noticeable effect on the energy consumption.

Figure 6.7 presents the impact of the energy-saving techniques on the speedup. DCE mode switching has the most dramatic effect on the performance. In integer applications, it decreases the speedup from 3.5% to 2.3%. The reduction is less severe in floating-point programs, where the IPC increase drops from 20% to 17%.

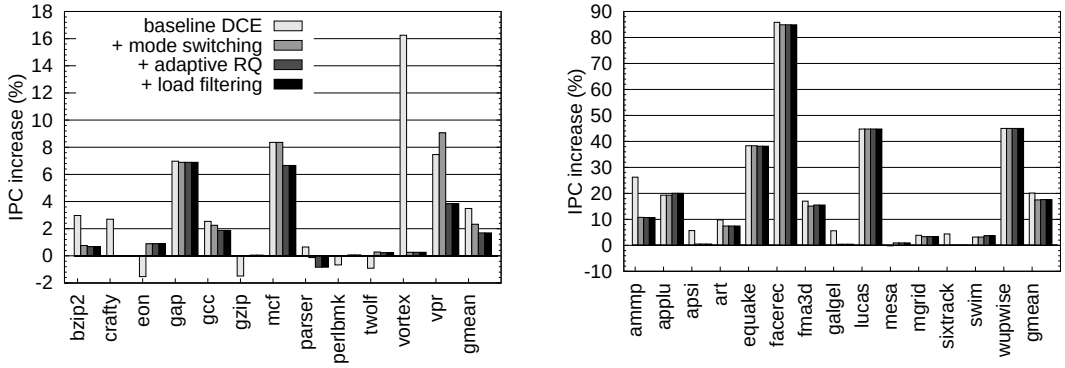

Figure 6.7: IPC increase provided by the baseline Dual-Core Execution configuration and three energy-efficient DCE configurations

In general, almost all programs that experience less than 5% benefit in the baseline DCE version lose that performance after mode switching is introduced. Adaptive RQ sizing has no effect on floating-point applications, but further reduces the speedup of integer programs to 1.7%.

6.5 Comparing Energy-Efficient Multi-Core Prefetching Techniques

This section provides a side-by-side comparison of energy-efficient implementations of different helper threading techniques. Figure 6.8 demonstrates the energy overhead of the baseline markov EDHT technique (*baseline EDHT*) as well as the most energy-efficient versions of EDHT (*energy-efficient EDHT*), future execution (*energy-efficient FE*), and dual-core execution (*energy-efficient DCE*).

In case of integer programs, the energy overhead of the various helper threading techniques significantly differs only in five applications - *bzip2*, *gcc*, *mcf*, *parser*, and *vpr*. This variation is substantially less than the variation between the techniques without the energy-saving features (see Figure 6.1). Nevertheless, in these five applications EDHT still exhibits a significant advantage over the other techniques.

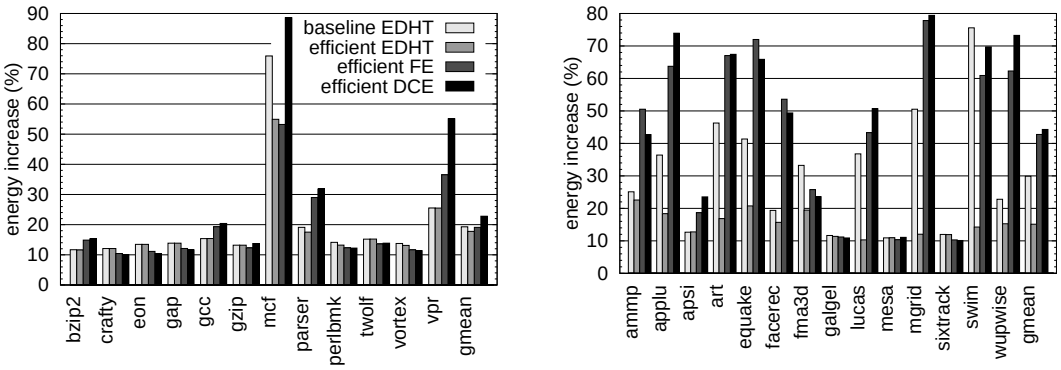

Figure 6.8: Energy consumption overhead of a the baseline markov EDHT technique, and energy-efficient versions of markov EDHT, future execution, and dual-core execution

On average, the energy-efficient versions of the helper threading techniques exhibit very similar energy overheads, which vary between 18% for energy-efficient EDHT and 23% for energy-efficient DCE. Thus, we conclude that the proposed energy-efficiency techniques are very effective on integer applications.

For floating-point programs, however, the situation is different. There is a significant variation between different helper threading techniques in all but three programs (*galgel*, *mesa*, and *sixtrack*). All three programs have few cache misses. The energy-saving techniques are very effective at detecting this behavior and dynamically switching off the prefetching activity. For most other applications, energy-efficient DCE incurs the highest energy overhead of 44%. The FE technique incurs less energy overhead than DCE (42% on average), but the difference is quite small. The *baseline EDHT* technique incurs less energy overhead than either FE or DCE for all but two applications (*fma3d* and *swim*). The *energy-efficient EDHT* model has the lowest energy overhead on all floating-point programs. On average, it exhibits an overhead of 15%, which is almost three times less than the overheads of FE and EDHT.

Figure 6.9 provides the comparison of execution time speedup provided by

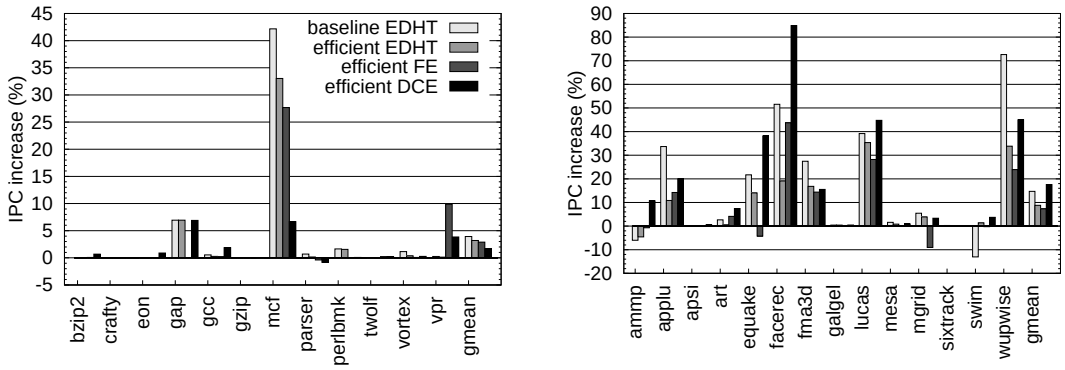

Figure 6.9: IPC increase provided by a baseline markov EDHT technique, and energy-efficient versions of markov EDHT, future execution, and dual-core execution

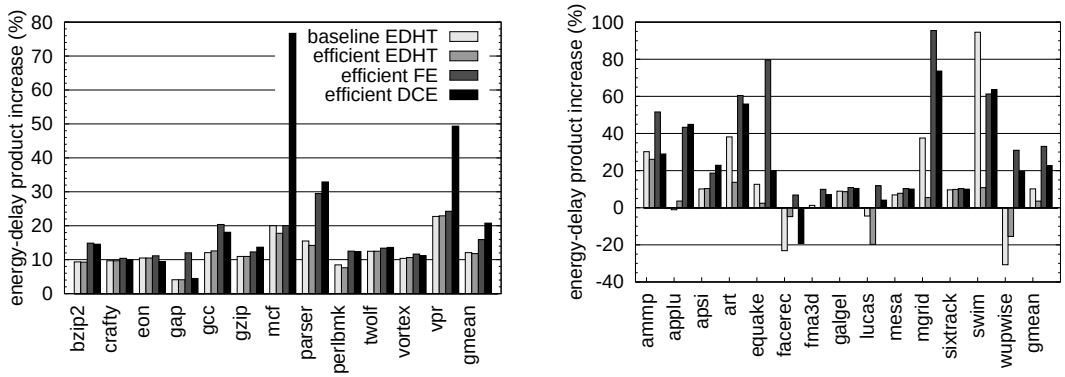

Figure 6.10: Increase in the energy-delay product for the baseline markov EDHT technique, and energy-efficient versions of markov EDHT, future execution, and dual-core execution

various techniques. The data demonstrates that if energy efficiency is taken into account, the *baseline EDHT* emerges as the most attractive alternative. It provides the highest IPC increase in integer programs (4%) and the second highest speedup in floating-point programs (15%). While DCE delivers the best performance of 17% in floating-point programs, it is only two percentage points better than the *baseline EDHT* model. The *energy-efficient EDHT* and *energy-efficient FE* techniques deliver the lowest performance of 9% and 7%, respectively.

Figure 6.10 demonstrates the increase in the energy-delay product incurred by the various helper threading techniques. In this graph, the lower the height

of the bar, the better the energy efficiency. On average, energy-efficient EDHT provides the lowest increase in the energy-delay product across both integer (12%) and floating-point applications (4%). The baseline EDHT model is the second best technique with a 12% increase for integer applications and a 10% increase for floating-point applications. Therefore, the two versions of the EDHT technique perform quite closely on the energy-delay metric. Both FE and DCE perform significantly worse on the energy-delay metric than either EDHT version.

6.6 Leakage Energy

All results in this chapter estimate dynamic energy consumption and do not consider the impact of the various prefetching techniques on static energy due to transistor leakage. Technology scaling, however, has been increasing the relative contribution of static power dissipation [9]. Thus, it is important to understand the effect of the various prefetching techniques on the static power dissipation.

Due to limitations of the simulation infrastructure, this section evaluates basic leakage trends using the analytical framework proposed by Butts and Sohi [3] and later refined in the HotLeakage framework [51]. In this framework, static energy can be estimated at the architectural level based on the following simple four-parameter model:

$$E_{static} = t * V_{cc} * N * k_{design} * I_{leakage} \tag{6.1}$$

where t is the execution time, Vcc is the power supply voltage, N is the number of transistors in the design, k_{design} is an empirically determined parameter representing the characteristics of an average device, and $I_{leakage}$ is a technology parameter describing the per device subthreshold leakage.

The HotLeakage framework further elaborates this equation by computing $I_{leakage}$ dynamically during the simulation using the BSIM3 leakage-current equation [29]. As a result, the leakage

$$I_{leakage} = \mu_0 * C_{OX} * \frac{W}{L} * e^{b*(V_{dd}-V_{dd0})} * v_t^2 * (1 - e^{\frac{-V_{dd}}{v_t}}) * e^{\frac{-|V_{th}|-V_{off}}{n*v_t}} \quad (6.2)$$

where μ_0 is the zero bias mobility, C_{OX} is the gate oxide capacitance per unit area, $\frac{W}{L}$ is the aspect ratio of the transistor, $e^{b*(V_{dd}-V_{dd0})}$ is the DIBL factor derived from the curve fitting method, V_{dd0} is the default supply voltage for each technology, $v_t = k * T/q$ is the thermal voltage, V_{th} is threshold voltage, which is also a function of temperature, n is the subthreshold swing coefficient, V_{off} is an empirically determined BSIM3 parameter, which is also a function of threshold voltage.

Based on these equations, it is relatively easy to see the relationships of some major factors that affect the leakage energy. Given a fixed k_{design}, the leakage energy is directly proportional to program execution time, operating voltage, the number of transistors in the design, and temperature. The following paragraphs discuss the effect of the considered multi-core prefetching approaches on each of these terms.

Execution time. The results in Figure 6.9 demonstrate that the average difference in execution time for all prefetching techniques does not exceed 2% on the integer programs and 10% on the floating-point applications. Moreover, the difference between the baseline markov EDHT and energy-efficient DCE does not exceed 2% on both the integer and the floating-point programs. Based on these observations, if all other terms of leakage current equations are equal, the energy-efficient versions of markov EDHT and FE are likely to consume up to 10% more static energy than the baseline EDHT or the energy-efficient DCE techniques on

the floating-point applications. On the integer applications, the difference in consumed static energy due to the execution time is likely to be less than 2%.

Number of Transistors. The number of transistors utilized by the prefetching approaches discussed in this section consists of the total transistors in the helper core and the transistors in additional hardware structures. While all prefetching techniques use the same general-purpose CMP core, they require a different amount of additional hardware support. In terms of transistor budget, EDHT requires the least amount of hardware support, while DCE and FE require a considerably higher hardware investment. FE requires several kilobytes of storage for prediction tables and is likely to incur the highest hardware overhead. Nevertheless, even several kilobytes of storage represent an insignificant portion of the total transistor budget (e.g., compared to a 2MB L2 cache) and will not affect leakage energy in a significant way.

Power Supply Voltage. The impact of supply voltage on leakage energy in the context of the various prefetching approaches greatly depends on whether the baseline CMP architecture has a dynamic voltage scaling (DVS) capability. In the absence of DVS, helper cores will have the same supply voltage and the different prefetching techniques will contribute equally to the supply voltage term of equations 6.1 and 6.2. If DVS is enabled, the supply voltage and operating frequency of the helper core can be decreased dynamically at runtime to achieve a lower power consumption at the expense of a slower execution speed. The impact of DVS on the total energy consumption overhead of the active helper core represents a separate research topic and is outside the scope of this work. Nevertheless, we can study the impact of DVS on leakage energy when the helper core is idle.

The energy-saving techniques proposed in this chapter for the FE and DCE

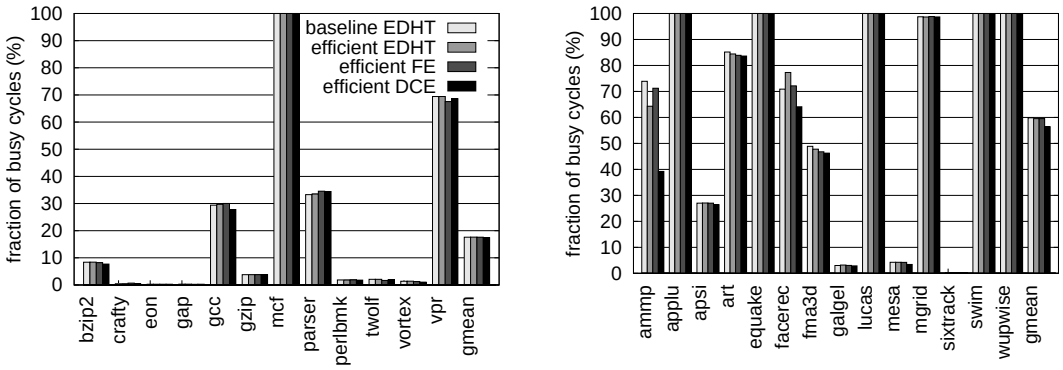

Figure 6.11: Fraction of time the helper core is activated for the baseline markov EDHT technique and the energy-efficient versions of markov EDHT, future execution , and dual-core execution

prefetching approaches dynamically clock-gate the helper core off if the main application thread experiences very few cache misses. Thus, the dynamic energy can be saved since no switching happens in the helper core. With DVS, it is further possible to reduce the static energy of the helper core by lowering the supply voltage of the helper core during such idle periods.

Figure 6.11 demonstrates the fraction of time that the helper core is not in idle mode for each energy-efficient prefetching technique. Note that energy-efficient markov EDHT was augmented with a dynamic switching capability. This capability decreases the average speedup provided by markov EDHT prefetching by less than 1%. Thus, our previous conclusions about the impact of execution time on leakage remain valid. The parameters used for mode switching among all energy-efficient techniques are the same: The algorithm for adaptive mode switching computes the L2 cache miss rate for every one million committed instructions and turns prefetching mode off if the cache miss rate is below 2.5 misses per 1000 committed instructions. The results show that each prefetching technique keeps the prefetching core idle for approximately the same number of cycles. As a consequence, even if DVS is enabled to lower the supply voltage during helper core idle

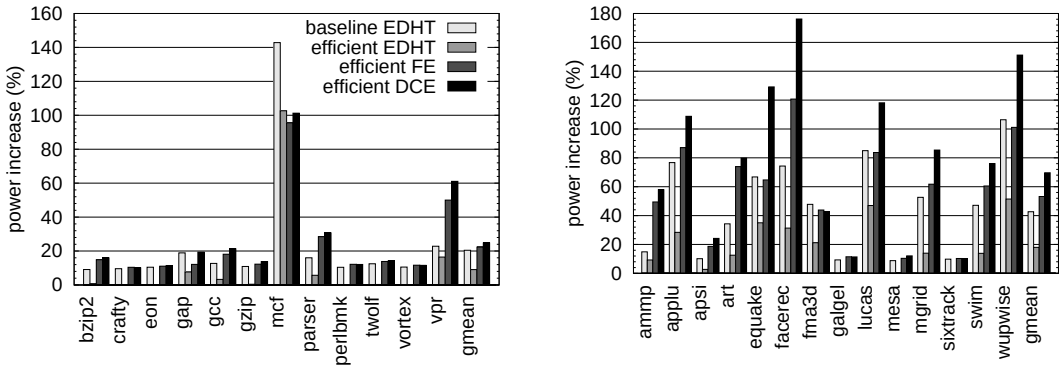

Figure 6.12: Increase in power consumption for the baseline markov EDHT technique and the energy-efficient versions of markov EDHT, future execution, and dual-core execution

cycles, the resulting energy savings will be the same for all prefetching approaches.

Temperature. The energy consumed by the microprocessor is converted into heat, which causes an increase in die temperature. At the same time, temperature is one of the most important factors affecting leakage energy since leakage current is proportional to the square of the temperature (see equation 6.1). The simulation infrastructure used in this dissertation does not provide exact temperature values, but it is possible to approximate the impact of the various prefetching techniques on the temperature by comparing the power dissipation of these techniques. While this approach ignores the effects of uneven heat dissipation (i.e., the existence of hot spots [45]) and static power dissipation, it allows to reach general conclusions about average die temperatures due to dynamic power dissipation.

Figure 6.12 compares the increase in power consumption incurred by the various prefetching approaches. All prefetchers use the dynamic mode switching technique to disable all activity in the helper core in case of a low cache miss rate. As in the case with energy consumption, energy-efficient EDHT incurs the smallest power overhead of 10% on the integer programs and 20% on the floating-point applications. All other techniques cause twice as much overhead on the integer workload

and up to three times as much overhead on the floating-point applications. Note that baseline markov EDHT has a significantly lower power overhead than both FE and DCE, especially on the floating-point programs. Overall, we conclude that both the baseline and energy-efficient versions of EDHT cause the lowest increase in die temperature due to dynamic power dissipation and, therefore, are likely to cause the lowest increase in static energy consumption.

Overall, this section estimates the impact of considered prefetching approaches on leakage energy consumptions by analytical analysis of their effect on execution time, operating voltage, number of transistors in the design, and temperature. Analysis shows that all considered prefetching approaches require similar amount of hardware and are affected equally by dynamic voltage scaling techniques. Thus, the main differences in leakage energy consumption will likely originate from differences in program execution time and temperature. While energy-efficient DCE provides the shortest execution time, it incurs significantly higher die temperature increase than EDHT prefetching technique. As a result, the baseline and energy-efficient versions of event-driven helper threading are likely to incur the least amount of leakage energy overhead.

6.7 Summary

This chapter shows that the helper threading techniques presented in the previous chapters considerably increase the processor energy consumption. For example, the Event-Driven Helper Threading, Future Execution, and Dual-Core Execution techniques increase energy consumption for floating-point programs by 30%, 84%, and 74%, respectively. Since modern microprocessors are often power-limited,

high energy overheads might present a significant challenge for the introduction of helper threading techniques into commercial microprocessors.

To address this problem, this chapter investigates techniques to reduce the energy overheads of Future Execution and Event-Driven Helper Threading. The proposed techniques are quite effective and reduce the energy overhead of Future Execution by a factor of three on the integer applications and by factor of two on the floating-point programs. Furthermore, two techniques are proposed to reduce the energy overhead of Event-Driven Helper Threading. While the energy-saving EDHT techniques are effective at reducing the overheads by more than a factor of two on the floating-point programs, they significantly penalize the execution time speedup. In addition, the chapter analyzes energy-saving techniques for Dual-Core Execution proposed in the literature and validates their effectiveness.

Finally, this chapter provides a side-by-side comparison of energy-efficient implementations of different helper threading techniques. It demonstrates that the energy-efficient version of EDHT consumes the least amount of dynamic energy among all considered helper threading techniques, but provides significantly less execution time speedup than energy-efficient DCE or the baseline EDHT version. Moreover, if FE, EDHT, and DCE techniques are evaluated based on the energy-delay metric, both the baseline and energy-efficient versions of EDHT emerge as the best choices by providing a high execution time speedup at a low energy cost.

CHAPTER 7

ANALYZING THE SPEC CPU2006 PROGRAMS

This chapter evaluates the cache miss rate of the SPEC CPU2006 benchmark programs and validates the importance of prefetching techniques in future generations of computer systems. Furthermore, it proposes a set of fast and simple microarchitecture-independent simulation techniques to evaluate the potential of the previously discussed prefetching methods. A comparative evaluation of address value prediction, future execution, and runahead execution techniques on the SPEC CPU2000 and CPU2006 benchmark suites concludes the chapter.

7.1 Motivation

The previous chapters of this dissertation concentrated on improving the performance of the programs from the SPEC CPU2000 benchmark suite. This benchmark suite became the standard for the evaluation of CPU performance after its introduction in December 1999. However, in September 2006 the SPEC organization released a new version of the benchmark suite, SPEC CPU2006, which introduced many significant changes to the suite. For example, only four out of the 26 CPU2000 programs were carried over to the new release. Moreover, the memory requirement and the execution time of the programs increased by more than an order of magnitude. Finally, the benchmark programs' source code significantly grew in size and complexity. The new suite represents a balanced mix of programs written in C, C++, Fortran, and a combination of C and Fortran.

Since this dissertation focuses on application-driven research, the new benchmark suite raises the question of whether the previously outlined research findings are valid in the context of the new benchmark suite. For example, it is vital to un-

derstand whether the memory wall remains a significant performance bottleneck. If the load cache miss rate is still an important problem, it is further necessary to quantify the prefetchability of these misses.

This observation motivated me to step back and look at studying the prefetching problem from a more fundamental perspective. It is difficult to draw definitive conclusions about the effectiveness of the various prefetching algorithms with the methodology used in the previous chapters of this dissertation. The main reason for this is the use of a detailed cycle-accurate simulator as the main research tool. The advantages of using a simulator are accurate estimations and definitive conclusions about the impact of various architectural techniques on execution time, energy consumption, reliability, etc.

Nevertheless, detailed cycle-accurate simulators also present a set of disadvantages. First, there are very slow and force architects to look at small instruction intervals. Second, they are often limited to specific ISAs, which makes it difficult to port research tools integrated with cycle-accurate simulators to new platforms. Third, by virtue of being detailed and complex, cycle-accurate simulators impose research studies to be microarchitecture-specific. However, the conclusions about the impact of a prefetching technique on the performance of wide-issue out-of-order microprocessors may not be valid for in-order single-issue architectures. Finally, studying prefetching techniques in the context of detailed simulators results in substantial simulation "noise". In particular, it becomes more difficult to isolate the cause-effect relationships that impact the effectiveness of prefetching due to an overwhelming number of factors that influence system performance.

Taking these challenges into account, this chapter presents microarchitecture-independent study of the memory system behavior of the SPEC CPU2006 bench-

mark programs. It builds upon modeling only the most vital aspects of caching with a trace-driven simulator and abandons the methodology of detailed cycle-accurate simulation. While this approach results in a certain loss of accurate execution time speedup estimation, it allows to reach more general conclusions about the fundamental tradeoffs among the various prefetching techniques.

7.2 Coverage Analysis Simulation Methodology

The trace-driven simulator used for the experimental studies in this chapter is built using the Pin toolkit [31]. Pin is a dynamic binary instrumentation system for x86 (both 32-bit and 64-bit) and Itanium instruction-set architectures. Similar to ATOM [48], Pin provides an API for writing program analysis tools. For these studies, I created a set of analysis tools that simulate caches of different sizes as well as various prefetching techniques.

All workloads from the SPEC CPU2000 and CPU2006 benchmark suites are run to completion using their reference input sets. All benchmark programs were compiled with Intel's C/C++ (v9.1) and FORTRAN (v5.4) compilers. The binaries were generated for a 64-bit version of x86 ISA using the *base* -O3 compiler options provided with the SPEC distribution.

The cache simulator created for this study can simulate eight separate caches in parallel. In particular, it simulates caches with sizes of 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, and 32MB. Each cache uses a 64B line size and 16-way associativity, allocates on writes, and implements a true LRU replacement policy. Due to the generally small working set for instructions, the effects of instruction caching are ignored and only data caches are simulated. The cache sizes were

chosen to represent a wide practical spectrum of microprocessor architectures, ranging from embedded to high-performance server-class microprocessors.

7.2.1 Event-Driven Helper Threading

The power of the Event-Driven Helper Threading technique lies in the ability of address value prediction techniques to correctly predict the next load address that will miss in the cache. To evaluate the potential of Event-Driven Helper Threading in a microarchitecture-independent way, we need to abstract implementation details such as cache tagging policies, sizes of event trigger buffers, etc. Instead, we need to study the inherent predictability properties of cache miss addresses. This subsection describes the methodology used for measuring the cache miss address predictability and discusses the experimental results.

We use the TCgen tool [2] to generate the analysis tool that measures the address stream predictability. TCgen is a tool that automatically generates VPC trace compressors that operate based on value prediction techniques. While the generated source code is tailored for trace compression, it can be easily adapted to study the predictability properties of any value stream. We use a set of four value predictors to characterize the cache miss address predictability. Table 7.1 presents detailed information on the type and organization of the value predictors. Note that each predictor can provide two distinct prediction values. Similar to the naming convention in Chapter 5, we refer to a DFCM predictor that provides a single prediction value as *dfcm* predictor, and a DFCM predictor that provides two distinct predictions as *markov* predictor.

The analysis tool is organized as follows. First, each load address issued by the simulated programs updates the cache simulator and indicates whether it was

Table 7.1: Value Predictors

predictor type	L1 entries	L2 entries	order
Stride	65536	NA	NA
FCM	65536	131072	1
FCM	65536	524288	3
DFCM	65536	524288	3

a cache hit or miss. If it is a cache miss, the PC of the instruction is used to query a set of value predictors for prediction of the load address. The predictions are compared to the actual load address and hit counters corresponding to the different value predictors are updated. Finally, the value predictors are updated with the current load address. This process is repeated for every load instruction executed in the benchmark programs.

7.2.2 Future Execution

We measure the potential of Future Execution by simulating the stride predictor from Table 7.1 and tracking the propagation of predictable values through the architectural register file. The analysis tool consists of two parts. The first part tracks the predictability of register values. For each executed instruction, the analysis tool first checks the predictability of the input operand values. Input operands can be correctly predicted in two ways - via direct prediction by a value predictor or via speculative computation. If all input operand values are correctly predicted by the stride predictor, the analysis tool marks all result registers of the instruction as predictable. If the input operands are not directly predictable by the value predictor, the analysis tool checks if the input registers are marked as correctly computed by the preceding instructions.

The second part of the analysis tool checks the predictability of the cache miss load addresses and updates the corresponding statistics. For each cache miss, the analyzer checks if the load address is directly predictable by the stride predictor or if the input registers are marked as correctly computed by the preceding instructions. If either case is true, the cache miss is considered to be prefetchable.

Note that this study makes the important assumption that the data-flow graph is very stable between subsequent loop iterations. We can conclude from the studies in Chapter 4 that this assumption generally holds true for instruction sequences with fewer than 64 instructions. Thus, we integrate a special register decay mechanism to invalidate predictable registers after 64 committed instructions.

7.2.3 Runahead Execution

The effectiveness of Runahead Execution depends to a large degree on the independence of the data-flow streams that compute the load addresses. To evaluate the potential of Runahead Execution in a microarchitecture-independent way, we need to abstract implementation details such as runahead distance, the sizes of the runahead cache, the penalties associated with starting and stopping runahead helper threads, etc. Instead, we study the inherent predictability properties of cache miss addresses. This subsection describes the methodology used for measuring the cache miss address predictability and discusses the experimental results.

Similar to the Future Execution study, the analysis tool consists of two parts. The first part tracks information about which register values are not dependent on load instructions that miss in the cache. This is achieved by maintaining an array of valid/invalid tags that directly correspond to the architectural register file. For each executed instruction, the analysis tool checks if any input register operand

is invalid. If any input register is marked as invalid or if the analyzed instruction is a load cache miss, the analyzer invalidates the tags of all output registers. If all input registers are valid and the instruction under consideration is not a cache miss, all output registers are marked as valid.

The second part of the analysis tool checks the prefetchability of cache miss load addresses and updates the corresponding statistics. The load address is considered to be prefetchable if all input registers for the analyzed load instruction are valid.

Note that for the case of runahead execution with a finite runahead distance all registers eventually become valid. To account for this property, we reset the invalid bits for all registers that were updated more than 2000 instructions ago. The value of 2000 instructions was chosen to fully hide the memory latency of up to 500 cycles for aggressive superscalar architectures with a maximum IPC of 4 (it would take at least 500 cycles to execute 2000 instructions on such an architecture).

7.3 Experimental Results

7.3.1 Load Cache Miss Rates in SPEC CPU2006 Programs

Figure 7.1 presents the load cache miss rates for individual SPEC CPU2006 programs. The results for integer and floating-point applications are shown in the upper and lower panels, respectively. Interestingly, seven out of the 28 benchmark programs have negligible cache miss rates even with cache sizes as small as 256KB (*gobmk*, *h264ref*, *perlbench*, *sjeng*, *gamess*, *namd*, and *povray*). Moreover, the cache miss rates for five floating-point programs (*GemsFDTD*, *bwaves*, *lbm*, *milc*, *zeusmp*) appear to be largely unaffected by increases in the cache size. As with the SPEC CPU2000 benchmark suite, *mcf* remains the most memory-intensive

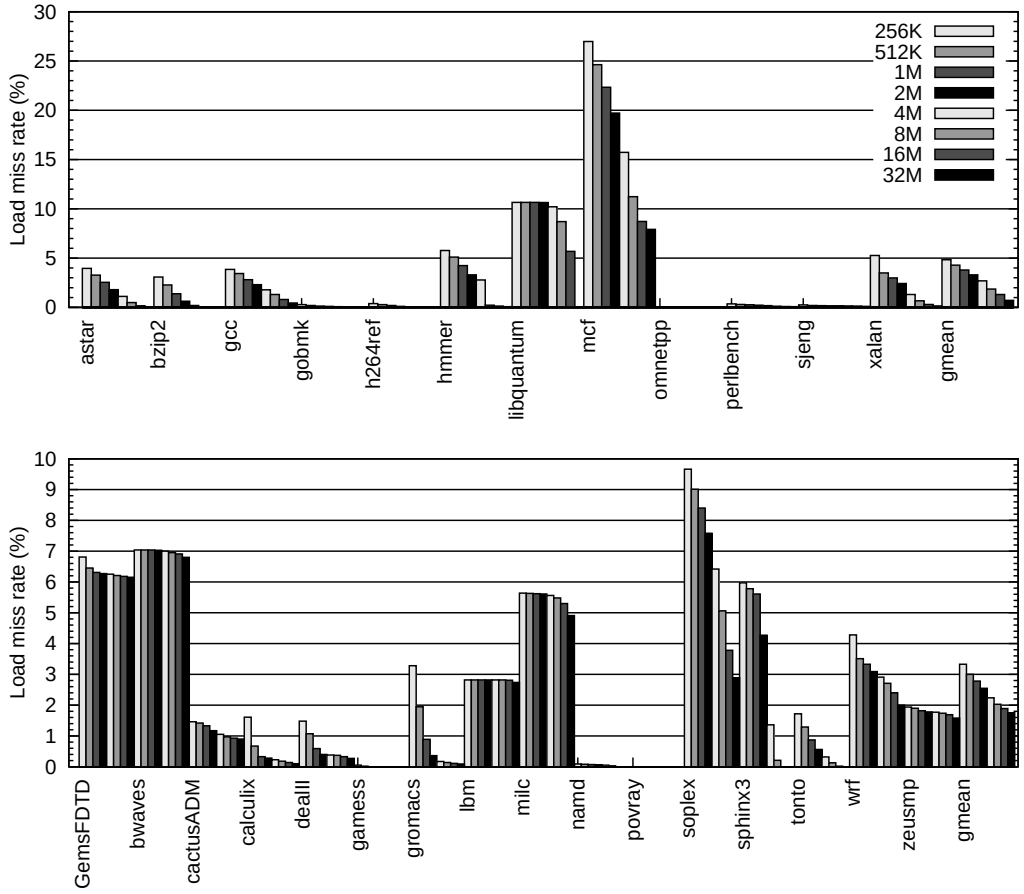

Figure 7.1: Load cache miss rate for the SPEC CPU2006 applications

program with the highest cache miss rate for every considered cache size. The rest of the programs generally exhibit a linear decrease in cache miss rate with exponentially increasing cache size.

These results demonstrate that the behavior of the individual benchmark programs is very similar between the SPEC CPU2000 and CPU2006 benchmark suites. With the exception of a few programs, most applications have relatively high miss rates. The response of the individual benchmark programs to increasing cache sizes varies greatly, but the geometric mean load miss rate decreases linearly with an exponentially increasing cache size.

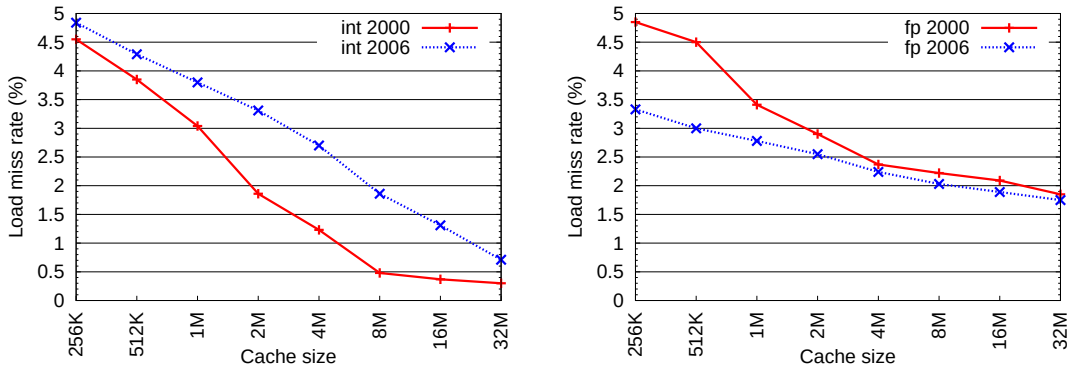

Figure 7.2: Sensitivity of the average load cache miss rate to the cache size

7.3.2 Comparing Cache Miss Rates in the SPEC CPU2000 and CPU2006 benchmark suites

Figure 7.2 compares the geometric mean cache miss rates of the two versions of SPEC CPU benchmark suite. The curves in the left panel demonstrate the sensitivity of the geometric mean cache miss rate of integer programs to the cache size. The right panel provides data for floating-point programs.

Integer programs in the CPU2006 benchmark suite have a higher cache miss rate for every considered cache size. The difference in cache miss rates varies greatly depending on the cache size. The difference between the two benchmark suites is relatively small for small cache sizes (256KB and 512KB). However, the difference rapidly increases with increasing cache size and reaches its maximum for cache sizes between 2MB and 8MB. However, the gap between the CPU2000 and CPU2006 suites quickly decreases beyond a cache size of 8MB.

Surprisingly, the floating-point component of the CPU2000 benchmark suite has a consistently higher cache miss rate than the later CPU2006 release. In particular, the CPU2000 programs have an almost 50% higher cache miss rate than the CPU2006 programs for small cache sizes (256KB and 512KB). Nevertheless, the

difference between the two versions of the benchmark suite quickly decreases with larger cache sizes and becomes relatively small (less than 15% relative difference) for sizes of 4MB and larger.

The data in Figure 7.2 leads to two observations. First, the "memory wall" remains an important problem in the new release of the benchmark suite. In fact, the severity of the problem is worse for integer programs and remains about the same for floating-point applications. Second, the microarchiture-independent data in Figure 7.2 can be used to quickly estimate the worst-case impact of cache misses on the execution time of the benchmark programs. The next subsection presents the methodology for the worst-case cache miss delay estimation in greater detail.

7.3.3 The Potential of Event-Driven Helper Threading

Figure 7.3 compares the impact of address value prediction techniques on the geometric mean cache miss rates of the SPEC CPU2006 benchmark suite. The data for integer programs are shown in the left graph and floating-point programs in the right graph. Each graph contains five curves. The first curve corresponds to the original load cache miss rate (*org miss rate*). The second curve (*stride vpred*) represents the cache miss rate if all cache miss addresses that are correctly predicted by a stride predictor are not considered to be cache misses. The next three curves demonstrate how many more cache misses can be eliminated if the stride predictor is augmented with a DFCM predictor (*dfcm vpred*), a markov predictor (*markov vpred*), or a combination of all predictors listed in Table 7.1 (*hybrid vpred*).

The stride predictor is very effective at predicting the cache miss addresses of both integer and floating-point applications. In particular, stride prediction covers

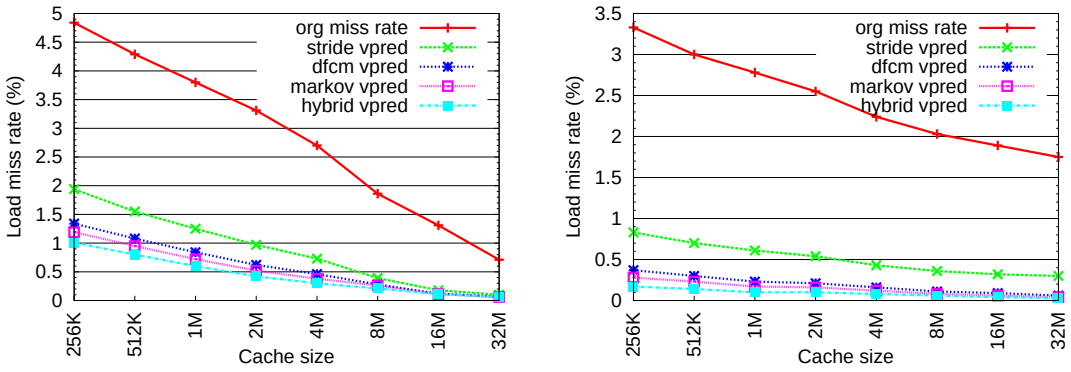

Figure 7.3: Impact of value prediction on the load cache miss rate

between 60% and 70% of all cache misses in integer programs and between 80% and 85% of the cache misses in floating-point programs. More sophisticated prediction algorithms are able to further reduce the effective cache miss rate, but the difference between the DFCM, Markov, and hybrid algorithms is relatively small. Moreover, this difference gets generally smaller with increasing cache size. This observation leads to the conclusion that building complicated hybrid prefetchers is likely to have a limited impact on the performance benefit.

None of the prediction algorithms can predict all cache miss addresses. The question of how much prediction coverage is sufficient greatly depends on the micro-architectural parameters. Let's consider an example of a conventional out-of-order four-wide superscalar architecture with a cache miss penalty of 200 cycles. Let's assume that on average 25% of all instructions are loads and the system can deliver an average IPC of 2 in the absence of cache misses. Finally, let's assume it is not desirable to invest in designing prefetching techniques if they deliver less than a 10% theoretical average performance improvement. Based on this information, it is possible to calculate the cache miss rate beyond which further improvements in prefetcher designs will bring no tangible benefit.

First, we can calculate that prefetching will be valuable only if cache misses

reduce the average IPC below 1.8. Second, we can determine that the average cache miss rate in this scenario should be at least 1 per 3600 committed instructions. Applying the property of every fourth instruction being a load, we can calculate that the minimal cache miss rate has to be 0.1%. None of the prefetching techniques is capable of reducing the cache miss rate below 0.1% on integer applications. In floating-point programs, however, the Markov predictor achieves effective cache miss rates of 0.12% and 0.08% for cache sizes of 4MB and 8MB, respectively. Thus, I conclude that the address value prediction technique is potentially sufficient to resolve the problems associated with cache misses at cache sizes equal to or larger than 4MB.

7.3.4 The Potential of Future Execution

Figure 7.4 demonstrates the impact of future execution on the geometric mean cache miss rates of the SPEC CPU2006 programs. The data for integer programs are shown in the left graph and for floating-point programs in the right graph. Each graph contains three curves. The first curve corresponds to the original load cache miss rate (*org miss rate*). The second curve (*stride vpred*) represents the cache miss rate if all cache miss addresses that are correctly predicted by a stride predictor are not considered to be cache misses. The last curve demonstrates how many more cache miss addresses can be predicted if the Future Execution technique is employed (*fexec*).

The results show that Future Execution behaves consistently across all cache sizes. On integer applications, Future Execution has the potential to correctly predict about 50% of the cache miss addresses that cannot be predicted by the stride predictor. On floating-point programs, Future Execution provides less ben-

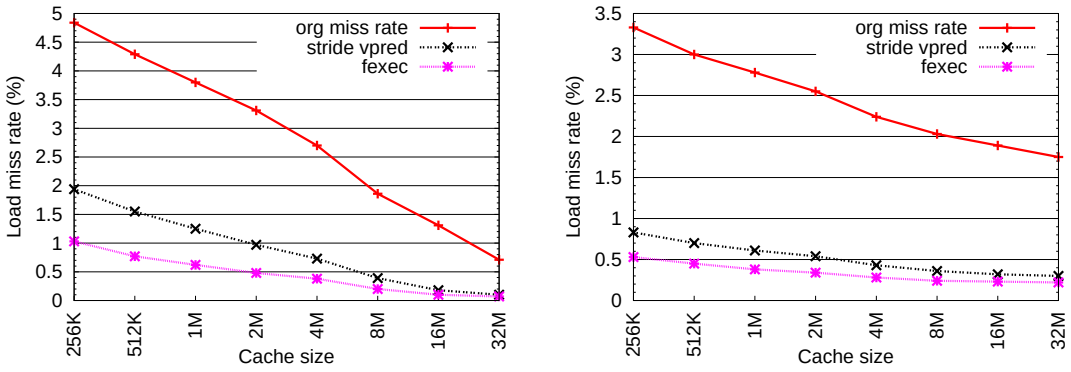

Figure 7.4: Impact of future execution on the load cache miss rate

efit when used in addition to a stride predictor. It is more effective for small cache sizes, but its additional benefit decreases rapidly with increasing cache size.

Similar to the behavior of the address value prediction techniques, the absolute difference between the stride value predictor and the more complex Future Execution technique reduces with growing cache size. Assuming that the absolute difference of less than 0.1% in cache miss rate is insignificant, Future Execution provides no tangible benefits beyond a cache size of 8MB for integer programs and 4MB for floating-point applications.

7.3.5 The Potential of Dual-Core Execution

Figure 7.5 demonstrates the impact of Dual-Core Execution on the geometric mean cache miss rates of the SPEC CPU2006 benchmark suite. The data for integer programs are shown in the left graph and for floating-point programs in the right graph. Each graph contains three curves. The first curve corresponds to the original load cache miss rate (*org miss rate*). The second curve (*stride vpred*) represents the cache miss rate if all cache miss addresses that are correctly predicted by a stride predictor are not considered to be cache misses. The last curve demonstrates

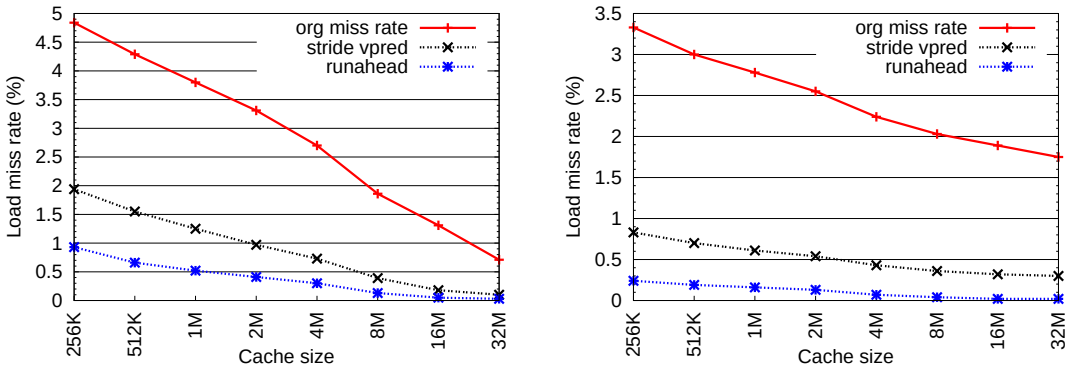

Figure 7.5: Impact of runahead execution on the load cache miss rate

how many more cache miss addresses can be predicted if the Runahead Execution technique is employed (*runahead*).

Similar to address value prediction and future execution, the prefetching coverage trends for runahead execution are consistent across multiple cache sizes. On integer applications, runahead execution can potentially prefetch between 50% and 60% of the load cache misses that are not predictable by a stride predictor. Runahead Execution is even more effective on floating-point programs, where is can correctly prefetch between 70% and 80% of the cache miss addresses not captured by the stride predictor. Moreover, runahead execution pushes the absolute cache miss rate below 0.1% for cache sizes beyond 8MB with integer programs and beyond 2MB with floating-point applications.

The absolute gap between the stride predictor and the runahead execution curves stays relatively constant for floating-point programs, but it decreases with size for integer benchmark suite. Thus, the benefits of runahead execution are more pronounced on integer programs for cache sizes below 4MB, but gets more important on floating-point applications for cache sizes beyond 4MB.

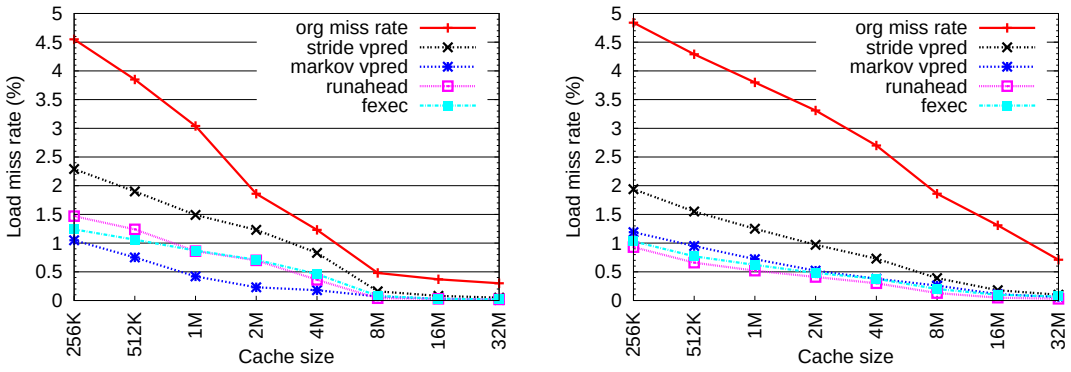

Figure 7.6: Impact of prefetching techniques on average load cache miss rate of integer applications in SPEC CPU2000 (left graph) and CPU2006 (right graph) suites

7.3.6 Comparing the Load Cache Miss Rate of the Various Prefetching Techniques

Figure 7.6 provides a side-by-side comparison of the prefetchability patterns of the CPU2000 and CPU2006 integer programs. The data for the CPU2000 programs are shown in the left graph and for the CPU2006 programs in the right graph. Each graph contains five curves. The first curve corresponds the original load cache miss rate (*org miss rate*). The second curve (*stride vpred*) represents the cache miss rate if all cache miss addresses that are correctly predicted by the stride predictor are not considered to be cache misses. The next three curves demonstrate how many more cache misses can be eliminated if a stride predictor is augmented with a Markov predictor (*markov vpred*), Future Execution (*fexec*), or Runahead Execution (*runahead*).

The experimental data demonstrates that many properties of load cache misses remain very similar between the two versions of the benchmark suite. First, the stride predictor can consistently predict a significant portion of the load cache miss addresses, but the other prefetching approaches can decrease the effective

cache miss rate even further. In addition, the absolute values of the effective cache miss rates provided by the stride prefetcher are almost the same between SPEC CPU2000 and CPU2006. Second, runahead execution and future execution behave similarly for cache sizes above 512KB. Moreover, the absolute distance between the stride prefetching curve and runahead/future execution in the CPU2006 applications did not change from the previous benchmark suite. Therefore, the average benefit of runahead and future execution on the CPU2006 applications is likely to remain similar to CPU2000 results.

Interestingly, the effect of the Markov predictor on the applications from the CPU2006 suite changed significantly. On the CPU2000 applications, the Markov predictor significantly outperforms the other prefetching techniques for all cache sizes of less than 8MB. On the CPU2006 applications, the Markov predictor is consistently worse than runahead or future execution. The main reason for this drastic change is the behavior of the *mcf* program. This benchmark program appears to be easily prefetchable by a big correlation predictor in the CPU2000 version, but the new CPU2006 version of the same application caused cache miss patterns that are not predictable by the same type of predictor. Overall, this implies that the importance of big correlation prefetchers on the SPEC CPU2006 integer applications is significantly reduced.

Figure 7.7 provides a side-by-side comparison of the prefetchability patterns in the floating-point applications of the CPU2000 and CPU2006 benchmark suites. The data for the CPU2000 programs are shown in the left graph and the CPU2006 programs in the right graph. The results for the two versions of the floating-point benchmark suite show almost no difference. The only significant change between the two benchmark suites is the relatively worse performance of future execution

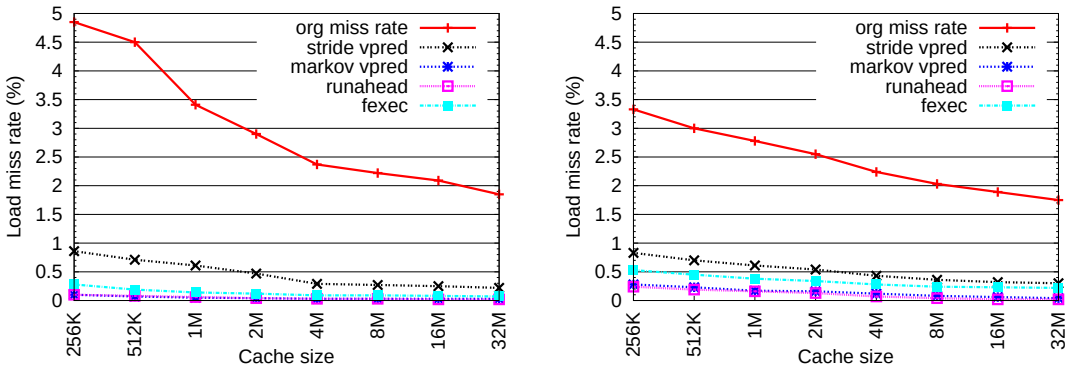

Figure 7.7: Impact of prefetching techniques on average load cache miss rate of floating-point applications in SPEC CPU2000 (left graph) and CPU2006 (right graph) suites

on the CPU2006 applications. Thus, we can expect future execution to provide less prefetching coverage in SPEC2006 compared to either runahead execution or Markov value prediction.

In general, the experimental results in this section demonstrate that all considered prefetching techniques in the context of the SPEC CPU2006 benchmark suite will have small differences in terms of potential prefetching coverage. Value prediction and runahead execution are practically equivalent in terms of prefetching coverage for any architecture with cache sizes above 1MB. If a difference of less than 0.1% in absolute cache miss rate has a negligible impact on execution time (which is a likely case for modern high-performance microprocessors), then address value prediction and runahead will not provide any competitive advantages over each other in terms of prefetching coverage. If a difference of less than 0.3% is tolerable, the prefetch coverage of all three considered techniques becomes indistinguishable.

Therefore, the competitive advantage of different prefetching techniques in the SPEC CPU2006 applications will not originate from increased prefetch coverage. Instead, the timeliness of the prefetch requests is likely to be a more differentiating

aspect when choosing the best technique.

7.4 Summary

This chapter validates the general conclusions of this dissertation in the context of the SPEC CPU2006 benchmark suite. First, it analyzes the load cache miss rates and demonstrates that the high latency of memory accesses is likely to remain an important problem in future generations of computer systems. The SPEC CPU2006 benchmark suite is characterized by a relatively large geometric mean cache miss rate across a wide range of cache sizes. In particular, when the cache size is increased from 256KB to 32MB in power of two increments, the geometric mean load cache miss rate changes almost linearly from 4.8% to 0.7% in integer programs and from 3.3% to 1.8% in floating-point programs. Compared to the previous SPEC CPU2000 version, SPEC CPU2006 has significantly higher cache miss rates in integer applications and similar cache miss rates in floating-point programs.

Furthermore, this chapter presents program analysis techniques to evaluate the potential prefetching coverage of different prefetching techniques. The analysis techniques concentrate on fundamental principles behind each considered prefetching technique and abstract away implementation details. As a result, the potential of event-driven helper threading, future execution, and runahead execution is evaluated by tracking specific data-flow information and analyzing the predictability of program values.

Using the proposed program analysis techniques, I investigated the differences between the previously discussed prefetching approaches on the SPEC CPU2000 and CPU2006 applications. I found that Markov value prediction and runahead

execution provide the same level of prefetching coverage on the CPU2006 applications, especially for cache sizes of 2MB and larger. Moreover, the usefulness of big correlation prefetchers is singinficantly reduced compared to the CPU2000 suite. Therefore, I conclude that the difference in performance improvement provided by markov value prediction and runahead execution will likely originate from differences in prefetch timeliness rather than coverage. This result is similar to the properties observed for the SPEC CPU2000 programs and, therefore, my previous findings are likely to remain relevant in the context of the CPU2006 benchmark suite release.

CHAPTER 8

RELATED WORK

This chapter describes ideas from the current data prefetching literature, most of which are used or built upon in this dissertation, and discusses some of the differences to the presented work.

8.1 Prefetching based on outcome prediction

Most data prefetching techniques are based on one of two broad classes of predictors – outcome prediction or operation prediction. Hardware prefetching techniques based on outcome prediction typically use various kinds of value predictors (e.g., [28, 37, 42]) and/or pattern predictors to dynamically predict which memory references should be prefetched. One of the first hardware prefetchers based on outcome prediction was the concept of stream buffers [23]. Subsequently, a number of other outcome prediction-based prefetching techniques was introduced. Examples include stride prefetching [10], the Markov prefetcher [22], content-directed prefetching [6], tag correlating prefetching [19], and dead-block correlating prefetching [26].

The advantage of prefetching schemes based on outcome prediction is the ability to implement the schemes in the cache controller so that other parts of the microprocessor do not need to be modified. This way the implementation of the prefetching scheme can be decoupled from the design of the execution core, significantly lowering the complexity and the verification cost. The downside of these prefetching schemes is their limited coverage and their inability to capture misses that exhibit irregular behavior.

I developed the Future Execution technique to alleviate some of these limitations. Unlike previous approaches, future execution employs value prediction only to provide initial predictions. These initial predictions are then used to compute all values reachable from the predictable nodes in the program dataflow graph, i.e., to obtain predictions for otherwise unpredictable values. I demonstrate that our approach significantly improves the prediction coverage relative to conventional value prediction. Since future execution requires relatively simple modifications to the execution core of the microprocessor, I believe it provides a reasonable tradeoff between the implementation cost and the resulting performance improvement.

To be effective on a wide range of applications, many of the aforementioned prefetching techniques require relatively large dedicated tables. While several approaches have been proposed to reduce the table sizes of differential Markov predictors [24, 34], these techniques are not applicable to other important Markov-based algorithms. Moreover, hardware designs typically cannot be reconfigured, which is why designers prefer to implement conservative prefetching algorithms that are unlikely to hurt any application. These issues have hampered the introduction of many promising techniques into commercial microprocessors.

Event-Driven Helper Threading represents an architectural framework that allows to use available cores in a CMP to run various prefetching algorithms, eliminating the need for dedicated hardware and table space. I further demonstrate that EDHT prefetching works naturally in combination with hardware stride prefetching, where the hardware prefetcher handles the simple patterns and the EDHT thread tackles the more complicated cases. Thus, the EDHT framework provides unique support for customizable prefetcher designs.

Similar to future execution, Dual-Core Execution [53] used value prediction

to speculatively compute unpredictable values of instructions currently held in the instruction window and speculatively issue load instructions. However, our mechanism provides better latency tolerance due to the use of future predictions and delivers a higher prediction coverage because the speculation scope is not limited by the number of instructions in the instruction window.

Similar to EDHT, Solihin et al. [46] propose to emulate hardware prefetching algorithms in software. Specifically, they employ user-level memory threads (ULMT) that are executed on a processor in the memory controller or in a DRAM chip for memory-side correlation prefetching. I found that PC information is crucial for the effective operation of Markov prefetchers, but PCs are not usually available at the memory interface. Hence, ULMT is algorithmically limited to exploiting only the global cache-miss history. In addition, ULMT observes cache misses in issue order. EDHT, on the other hand, has access to the PCs and observes the misses in program order, thus enabling it to achieve a higher prefetching coverage and accuracy. ULMT faces scalability challenges in multi-core environments where many threads might simultaneously try to access the shared memory. EDHT naturally scales with the number of cores. Finally, our EDHT approach mainly relies on the pre-existing hardware resources of a CMP and conventional user-level threads while ULMT requires a programmable memory processor, OS support to load the prefetching threads into specialized processors, and compiler support for the memory processor's (most likely different) ISA.

8.2 Prefetching based on operation prediction

Prefetching techniques based on pre-execution [32, 39, 40] typically use additional execution pipelines or idle thread contexts in a multithreaded processor to execute

helper threads that perform dynamic prefetching for the main thread. Typically, these techniques create pre-execution helper threads (PEHT) by extracting program slices that compute critical data addresses. Then they insert triggers for these helper threads into the original program. The execution of the helper threads at run-time precomputes critical data addresses ahead of the original program and issues prefetch requests. Helper threads can be constructed dynamically by specialized hardware structures or statically. If a static approach is used, the prefetching threads are constructed manually [54] or generated by the compiler [30] or a trace analysis tool [41]. If PEHTs are constructed dynamically, a hardware analyzer extracts execution slices from the dynamic instruction stream at run-time, identifies trigger instructions to spawn the helper threads, and stores the extracted threads in a special table. Examples of this approach include slice-processors [32] and dynamic speculative precomputation [5].

Future Execution has the following advantages over pre-execution prefetching. First, FE allows to dynamically change the prefetching distance through a simple adjustment in the predictor. Second, since a PEHT is only able to execute once all inputs to the thread are available, it runs a higher risk of prefetching late. FE, on the other hand, does not need all inputs to initiate prefetching. Third, if any load with dependent instructions in a PEHT misses in the cache, the prefetching thread will stall, preventing further prefetching. FE often breaks dataflow dependencies through value prediction and thus can avoid stalling the prefetch activity. Compared to software PEHT approaches, FE does not require re-compilation or binary rewriting and thus can speedup legacy code.

EDHT differs from these approaches in several ways. First, EDHT threads are spawned on architectural events and as such do not require any explicit thread

triggers to be inserted into the original program. Second, speculative precomputation threads sometimes require explicit progress synchronization with the main thread during their execution. EDHT threads require no synchronization with the main thread once they are launched. Third, EDHT helper threads can be generated without knowledge about the main program. Therefore, EDHT threads can provide benefits without the need for program analysis or recompilation.

On the other hand, prefetching based on pre-execution can potentially provide a higher prefetching coverage than future execution since it is not limited by the predictability of the program data. In addition, PEHTs typically need to execute fewer instructions than FE and as such can operate profitably on the same core together with the main thread. Finally, pre-execution requires no value prediction table and software approaches need no hardware support at all. Notwithstanding, I believe my approach may well be complementary to software-controlled pre-execution helper threads.

Slipstream prefetching [20] is another form of a software-controlled pre-execution that targets distributed shared-memory (DSM) applications. Slipstream prefetching threads represent a reduced version of the target computation threads. This reduced version dynamically skips the execution of shared memory stores and synchronization primitives and thus is able to run ahead of the target thread and generate an accurate address stream. As a result, Slipstream prefetching can provide higher prefetching coverage than FE. However, the proposed approach targets only DSM applications and cannot speed up single-threaded programs.

Runahead execution is another proposal for prefetching based on speculative execution [8,33]. In runahead processors, the processor state is checkpointed when a long-latency load stalls the head of the ROB, the load is removed and the pro-

cessor continues to execute speculatively. When the data is finally received from memory, the processor rolls back and restarts execution from the load. Future execution does not need to experience a cache miss to start prefetching, requires no checkpointing support or any other recovery mechanism and, as I demonstrate is this dissertation, works well in combination with runahead execution.

Similar to FE and EDHT, hardware-only techniques such as the minimal dual-core speculative multi-threading architecture [47] (SpMT) and the dual-core execution paradigm (DCE) [52] utilize idle cores of a CMP to speed up single-threaded programs. The DCE approach effectively proposes to launch a Runahead execution thread whenever a long-latency load instruction fully stalls the execution of a thread. The Runahead core then tries to follow the program path and execute all instructions that do not depend on the results of load instructions that miss in the cache while invalidating all instructions that stall the execution. The regular core re-executes all instructions committed by the runahead core. If the regular core detects that the runahead core deviated from the correct control path, it flushes the runahead core's pipeline and restarts runahead execution. In contrast to DCE, FE and EDHT techniques never require to check the results produced by the speculative threads and are, therefore, recovery-free. In addition, DCE requires the non-speculative core to redirect the instruction fetch engine upon reaching the speculation point, while in FE and EDHT the non-helper computation is not even aware that prefetching is taking place.

The SpMT approach spawns speculative threads on procedure calls, loop boundaries, or cache misses and executes them on another core. Speculative threads prefetch important data, precompute branch outcomes, and perform some useful computation that can later be integrated into the non-speculative thread. The

main difference between SpMT and the techniques proposed in this dissertation is that the former needs mechanisms to control the execution of the speculative threads by tracking the violation of memory and register dependences. The execution of helper threads proposed in this dissertation is completely decoupled from the non-speculative execution, which eliminates the need for any checking mechanism and makes it recovery-free. In general, SpMT requires more complex hardware support but may provide more performance benefit since it allows the reuse of speculatively computed results.

Similar to EDHT, the concept of Informing Memory Operations (IMO) [16,17] proposes to expose information about individual cache miss events to software via a special class of load instructions. The IMO approach is shown to enable a number of software-based memory optimizations in the context of performance monitoring, cache coherence, software-controlled multithreading and software-controlled prefetching. Despite some similarities, EDHT and IMO prefetching differ in two significant ways. First, software prefetching via IMO support is based on operation prediction, while EDHT is used primarily for outcome prediction. Second, IMO functionality is focused on modifying the original applications to insert IMO instructions and prefetching code. EDHT proposes to completely decouple the execution of the prefetching algorithms from the original application. As a result of these differences, IMO approach requires more extensive software and compiler support, while the implementation of EDHT technique may require more hardware resources.

CHAPTER 9

SUMMARY AND CONCLUSIONS

Scaling the performance of applications with little thread-level parallelism is one of the most serious impediments to the success of multi-core architectures. At the same time, the long latency of memory accesses represents one of the largest performance bottlenecks for individual program threads. The goal of this dissertation is to develop simple yet efficient techniques that utilize extra cores of a CMP as helper engines to increase the performance of single program threads. In particular, this work focuses on using available cores to run data prefetching algorithms that mitigate the detrimental effects of long memory access latencies.

An investigation of program properties revealed that most load cache misses occur in loops with relatively short iterations. This observation suggests a prefetching approach in which each load instruction triggers a prefetch of the address that the same load will reference in the n^{th} next "iteration". Furthermore, I analyzed the instructions in the dataflow graphs of the problem loads and found that, while problem load addresses might be hard to predict, the inputs to their dataflow graphs often are not. Therefore, even when the miss address itself is unpredictable, it may be possible to predict the input values of the instructions leading up to the problem loads and thus to compute an accurate prediction by executing these instructions.

To exploit these properties, I developed the idea of the Future Execution (FE) data prefetching technique. Future execution continuously generates a prefetching thread by applying a set of simple transformations to the stream of instructions committed by a target program thread. It further uses the execution capabilities of an available core in a multi-core microprocessor to execute the generated pre-

fetching thread and prefetch data for its target thread, which runs on a different core of the same chip.

Experimental results from a cycle-accurate processor simulator demonstrate that the FE technique can provide significant performance improvements for a wide range of applications. Overall, FE can deliver a geometric-mean speedup of 5% on integer and 21% on floating-point programs over a baseline with a hardware stream prefetcher. Furthermore, I demonstrate that future execution is complementary to runahead execution and the combination of these two techniques significantly raises the average speedup. I analyzed the sensitivity of future execution to several architectural parameters, such as the minimum memory latency, the inter-core communication delay/bandwidth, and the prefetch distance. The results demonstrate that prefetching based on future execution delivers robust performance improvements across many processor configurations.

Next, I analyzed the limitations of previously proposed outcome-prediction based prefetching techniques and decided to approach the central question of this dissertation from a different angle. Instead of devising new prefetching algorithms, I considered ways to alleviate these limitations by utilizing the execution capabilities of available cores in CMP architectures. As a result, I propose the Event-Driven Helper Threading (EDHT) lightweight architectural framework to emulate prefetching algorithms via a special class of software helper threads. The key principle behind EDHT is to use special hardware to expose information about individual cache miss events to the ISA and flexible software to implement the prefetching algorithms. Thus, the EDHT concept efficiently exploits architectural levels of abstraction by utilizing simple and fast hardware to communicate event data and flexible software to implement prefetchers of almost arbitrary complexity.

The performance results reveal that EDHT-based prefetching provides similar speedup as pure hardware implementations of the same prefetching algorithms. Furthermore, running prefetching EDHTs on top of a baseline with a hardware stride prefetcher yields speedups between 5% and 70% on a wide range of SPEC CPU2000 programs. Thus, the combination of hardware stride prefetching and more complex prefetching mechanisms implemented in the proposed EDHT framework exhibits significant synergy, as the hardware prefetcher detects and prefetches simple patterns, while software EDHT tackles more complicated load cache miss patterns.

This dissertation provides performance numbers for the future execution, event-driven helper threading, and dual-core execution prefetching techniques in the same environment, making it possible to compare the performance of the studied multi-core prefetchers. The results lead to two main observations. First, none of the techniques uniformly outperforms the others. Second, while dual-core execution provides the best average speedup on floating-point applications, the EDHT framework provides competitive performance, executes fewer instructions, and requires considerably simpler hardware support.

Since the power consumption of high-performance microprocessors has recently become a high-priority concern for computer architects, this dissertation further describes and evaluates techniques to improve the energy-efficiency of the studied multi-core prefetching mechanisms. Experimental results show that the proposed energy-efficient techniques are very effective and reduce the energy overhead of the FE and EDHT techniques by more than a factor of two. Moreover, if the FE, EDHT, and DCE techniques are evaluated based on the energy-delay metric, both the baseline and the energy-efficient version of EDHT emerge as the best choices

by providing a high execution time speedup at a low energy cost.

The final chapter of this dissertation validates the general conclusions of this dissertation in the context of the SPEC CPU2006 benchmark suite. First, I analyze the load cache miss rates and demonstrate that the high latency of memory accesses is likely to remain an important problem in future generations of computer systems. Second, I utilize microarchitecture-independent analysis techniques to evaluate the potential prefetching coverage of different prefetching techniques. The results show that Markov value prediction and runahead execution provide the same level of prefetching coverage, especially for cache sizes of 2MB and larger. Therefore, the difference in performance improvement provided by the various prefetching techniques will likely originate from differences in the prefetch timeliness rather than from coverage. This result is similar to the properties observed for SPEC CPU2000 programs and, therefore, the general conclusions of this dissertation are likely to remain relevant in the context of a newer benchmark suite release.

Looking into the future, the EDHT concept opens opportunities for designing novel prefetching techniques. For example, the flexibility of a software approach provides an interesting possibility for the automatic generation of program-specific EDHT threads. Hybrid prefetchers that run several pre-fetching algorithms in parallel may also be possible. Finally, the EDHT framework may make it feasible to quickly prototype novel prefetching techniques on real hardware without the need to recompile applications, to modify the silicon, or to resort to slow simulations.

The prefetching techniques proposed in this dissertation utilize a fixed prefetch distance. Nevertheless, sensitivity studies demonstrate that both the FE and the EDHT technique can greatly benefit from a dynamic mechanism to adjust the prefetch distance. In the future, it may be worthwhile to investigate dynamic

prefetch distance mechanisms to fully utilize the performance potential of the proposed techniques.

In this work, I focus on how to use EDHT for prefetching because this is its most immediately beneficial application. However, the idea can trivially be extended to implement other helper engines. For example, additional events such as mispredicted branches, TLB misses, other stall events, certain touched addresses, temperature events, etc. could be exposed in a similar manner. Such a system might provide flexible, non-intrusive, real-time performance monitoring beyond the capabilities of traditional hardware counters and could support online phase detection and code re-optimization, facilitate debugging, gather and compress execution traces, or emulate complex branch predictors.

BIBLIOGRAPHY

[1] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94, New York, NY, USA, 2000. ACM Press.

[2] Martin Burtscher and Nana B. Sam. Automatic generation of high-performance trace compressors. In *Proceedings of the international symposium on Code generation and optimization*, pages 229–240, Washington, DC, USA, 2005. IEEE Computer Society.

[3] J. Adam Butts and Gurindar S. Sohi. A static power model for architects. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 191–201, New York, NY, USA, 2000. ACM Press.

[4] Luis Ceze, Karin Strauss, James Tuck, Jose Renau, and Josep Torrellas. Cava: Hiding l2 misses with checkpoint-assisted value prediction. *IEEE Comput. Archit. Lett.*, 3(1):7, 2004.

[5] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317, 2001.

[6] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 279–290, 2002.

[7] Digital Equipment Corporation. *Alpha Architecture Handbook*, 1992.

[8] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th international conference on Supercomputing*, pages 68–75, 1997.

[9] Don Edenfeld, Andrew B. Kahng, Mike Rodgers, and Yervant Zorian. 2003 technology roadmap for semiconductors. *Computer*, 37(1):47–56, 2004.

[10] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th annual international symposium on Microarchitecture*, pages 102–110, 1992.

[11] F. Gabbay. Speculative execution based on value prediction. Tech. report 1080, Technion - Israel Institute of Technology, NOvember 1996.

[12] Ilya Ganusov. Hardware prefetching based on future execution in chip multiprocessor architectures. Master's thesis, Department of Electrical and Computer Engineering, Cornell University, Ithaca, New York, August 2005.

[13] Ilya Ganusov and Martin Burtscher. Future execution: A hardware prefetching technique for chip multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 350–360, September 2005.

[14] B. Goeman, H. Vandierendonck, and K. de Bosschere. Differential fcm: Increasing value prediction accuracy by improving table usage efficiency. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, 2001.

[15] Bart Goeman, Hans Vandierendonck, and Koen de Bosschere. Differential fcm: Increasing value prediction accuracy by improving table usage efficiency. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[16] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing memory operations: providing memory performance feedback in modern processors. In *Proceedings of the 23rd annual international symposium on Computer architecture*, pages 260–270, New York, NY, USA, 1996. ACM Press.

[17] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing memory operations: memory performance feedback mechanisms and their applications. *ACM Trans. Comput. Syst.*, 16(2):170–205, 1998.

[18] http://www.spec.org/osg/cpu2000/.

[19] Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. Tcp: Tag correlating prefetchers. In *Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture*, page 317, 2003.

[20] Khaled Z. Ibrahim, Gregory T. Byrd, and Eric Rotenberg. Slipstream execution mode for cmp-based multiprocessors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 179, 2003.

[21] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 252–263, New York, NY, USA, 1997. ACM Press.

[22] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 252–263, 1997.

[23] Norman 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*, pages 364–373, 1990.

[24] Gokul B. Kandiraju and Anand Sivasubramaniam. Going the distance for tlb prefetching: an application-driven study. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 195–206, Washington, DC, USA, 2002. IEEE Computer Society.

[25] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez. Checkpointed early load retirement. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 16–27, 2005.

[26] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th annual international symposium on Computer architecture*, pages 144–154, 2001.

[27] E. Larson, S. Chatterjee, and T. Austin. Mase: a novel infrastructure for detailed microarchitectural modeling. In *Proceedings of the The Second International Symposium on Performance Analysis of Systems and Software*, pages 1–9, 2001.

[28] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 138–147, 1996.

[29] W. Liu, X. Jin, J. Chen, M-C. Jeng, Z. Liu, Y. Cheng, K. Chen, M. Chan, K. Hui, J. Huang, R. Tu, P.K. Ko, and Chenming Hu. Bsim 3v3.2 mosfet model users' manual. Technical Report UCB/ERL M98/51, EECS Department, University of California, Berkeley, 1998.

[30] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th annual international symposium on Computer architecture*, pages 40–51, 2001.

[31] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[32] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-processors: an implementation of operation-based prediction. In *Proceedings of the 15th international conference on Supercomputing*, pages 321–334, 2001.

[33] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In

Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture, page 129, 2003.

[34] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the 10th international symposium on High Performance Computer Architecture*, pages 96–106, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[35] Paramjit S. Oberoi and Gurindar S. Sohi. Parallelism in the front-end. *SIGARCH Comput. Archit. News*, 31(2):230–240, 2003.

[36] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st annual international symposium on Computer architecture*, pages 24–33, 1994.

[37] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st annual international symposium on Computer architecture*, pages 24–33, 1994.

[38] Justin Rattner. Multi-core to the masses. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, page 3, 2005.

[39] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 115–126, 1998.

[40] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, page 37, 2001.

[41] Amir Roth and Gurindar S. Sohi. A quantitative framework for automated pre-execution thread selection. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 430–441, 2002.

[42] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 248–258, 1997.

[43] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, 2002.

[44] Premkishore Shivakumar and Norman P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Tech. report WRL-2001-2, Compaq Western Research Laboratory, December 2001.

[45] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 2–13, New York, NY, USA, 2003. ACM Press.

[46] Yan Solihin, Jaejin Lee, and Josep Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 171–182, Washington, DC, USA, 2002. IEEE Computer Society.

[47] Srikanth T. Srinivasan, Haitham Akkary, Tom Holman, and Konrad Lai. A minimal dual-core speculative multi-threading architecture. In *Proceedings of the IEEE International Conference on Computer Design*, pages 360–367, 2004.

[48] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, New York, NY, USA, 1994. ACM Press.

[49] M. Dimitrov Y. Ma, H. Gao and H. Zhou. Optimizing dual-core execution for power efficiency and transient-fault recovery. *IEEE Transactions on Parallel and Distributed Systems*, 2007.

[50] C. Zhang and S. A. McKee. Hardware-only stream prefetching and dynamic access ordering. In *Proceedings of the 14th international conference on Supercomputing*, pages 167–175, 2000.

[51] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects. Technical Report CS-2003-05, University of Virginia Department of Computer Science, 2003.

[52] H. Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[53] H. Zhou and T. M. Conte. Enhancing memory level parallelism via recovery-free value prediction. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 326–335, 2003.

[54] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th annual international symposium on Computer architecture*, pages 2–13, 2001.