COMPILER-ASSISTED HARDWARE-BASED DATA PREFETCHING FOR NEXT GENERATION PROCESSORS

A Dissertation Presented

by

YAO GUO

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2007

Electrical and Computer Engineering

COMPILER-ASSISTED HARDWARE-BASED DATA PREFETCHING FOR NEXT GENERATION PROCESSORS

A Dissertation Presented

by

YAO GUO

Approved as to style and content by:

Csaba Andras Moritz, Chair

Israel Koren, Member

C. Mani Krishna, Member

Charles C. Weems, Member

Christopher V. Hollot, Department Chair
Electrical and Computer Engineering

ACKNOWLEDGMENTS

First of all, I am indebted to my advisor Prof. Csaba Andras Moritz for his encouragement, advice, mentoring, and research support throughout my whole PhD study. Professor Moritz's insights and ideas have provided me the greatest help to make this dissertation possible.

I want to thank Prof. C. Mani Krishna, Prof. Israel Koren and Dr. Zahava Koren for their help during my research at UMass. Through all the research meetings and discussions, their technical advices have helped my research work and this dissertation greatly.

I also want to thank my dissertation committee member Prof. Charles Weems, along with Profs. Moritz, Koren and Krishna, all the committee members have contributed to this work significantly through my preparation of the proposal to the final defense and revision of this dissertation.

I am fortunate to have the opportunity to work with a group of energetic people from the Software Systems and Architectures Lab. I want to thank all the former and present members of the SSA Lab who have helped me a lot through collaborations and discussions, especially Raksit Ashok, Saurabh Chheda, Mahmoud Bennaser, Zhenghua Qi, Teng Wang, Pritish Narayanan, and Michael Leuchtenburg.

Finally, my heart goes to my family back in China, especially to my parents, for their support and patience through my 25 long years as a student. Last, but not the least, I want to thank my wife Qi Li for her tolerance, encouragement, and love that help me through all the way.

Thanks, to all of you!

ABSTRACT

COMPILER-ASSISTED HARDWARE-BASED DATA PREFETCHING FOR NEXT GENERATION PROCESSORS

MAY 2007

YAO GUO

B.S., PEKING UNIVERSITY

M.S., PEKING UNIVERSITY

M.S., OREGON HEALTH & SCIENCES UNIVERSITY

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Csaba Andras Moritz

Prefetching has emerged as one of the most successful techniques to bridge the gap between modern processors and memory systems. On the other hand, as we move to the deep sub-micron era, power consumption has become one of the most important design constraints besides performance. Intensive research efforts have been done on data prefetching focusing on performance improvement, however, as far as we know, the energy aspects of prefetching have not been fully investigated.

This dissertation investigates data prefetching techniques for next-generation processors targeting both energy-efficiency and performance speedup. We first evaluate a number of state-of-the-art data prefetching techniques from an energy perspective and identify the main energy-consuming components due to prefetching.

We then propose a set of compiler-assisted energy-aware techniques to make hardware-based data prefetching more energy-efficient.

From our evaluation on a number of data prefetching techniques, we have found that if leakage is optimized with recently proposed circuit-level techniques, most of the energy overhead of hardware data prefetching comes from prefetch hardware related costs and unnecessary L1 data cache lookups related to prefetches that hit in the L1 cache. This energy overhead on the memory system can be as much as 30%.

We propose a set of *power-aware prefetch filtering* techniques to reduce the energy overhead of hardware data prefetching techniques. Our proposed techniques include three *compiler-based filtering* approaches that make the prefetch predictor more energy efficient. We also propose a *hardware-based filtering* technique to further reduce the energy overhead due to unnecessary prefetching in the L1 data cache. The energy-aware filtering techniques combined could reduce up to 40% of the energy overhead introduced due to aggressive prefetching with almost no performance degradation.

We also develop a *location-set driven data prefetching* technique to further reduce the energy consumption of prefetching hardware. In this scheme, we use a *power-aware prefetch engine* with a novel design of an indexed hardware history table. With the help of compiler-based location-set analysis, we show that the proposed prefetching scheme reduces the energy consumed by the prefetch history table by 7-11X with very small impact on performance.

Our experiments show that the proposed techniques could overcome the prefetching-related energy overhead in most applications, improving the energy-delay product by 33% on average. For many applications studied, our work has transformed data prefetching into not only a performance improvement mechanism, but an energy saving technique as well.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Prefetching has emerged as one of the most successful techniques to bridge the gap between modern processors and memory systems. On the other hand, as we move to the deep sub-micron era, power and energy consumption has become one of the most important design constraints besides performance. Intensive research efforts have been done on data prefetching focusing on performance improvement [8, 12, 20, 22–25, 27, 29, 37, 44, 46, 51, 54, 59, 59, 60, 63, 69, 76, 81, 84, 87, 88, 97, 98, 104, 106, 107], however, the energy aspects of prefetching have not been fully investigated. This dissertation investigates data prefetching techniques for next-generation processors targeting both energy-efficiency and performance.

1.1 Data Prefetching

The expanding gap between microprocessor and DRAM performance (Figure 1.1) has been encouraging researchers to explore increasingly aggressive techniques to reduce or hide the large latency of main memory accesses. The use of cache memory hierarchies [89] has been the key technique to hide the memory latency. With the introduction of multi-level caches in contemporary microprocessors, the performance degradation due to long memory latency has been greatly reduced.

However, the use of cache memory hierarchy is not the "silver bullet" to solve all the problems caused by the "*memory wall*". As the memory gap keeps increasing, the importance of improving the performance at each level of the memory hierarchy will continue to grow. Researchers have proposed many effective mechanisms beyond the

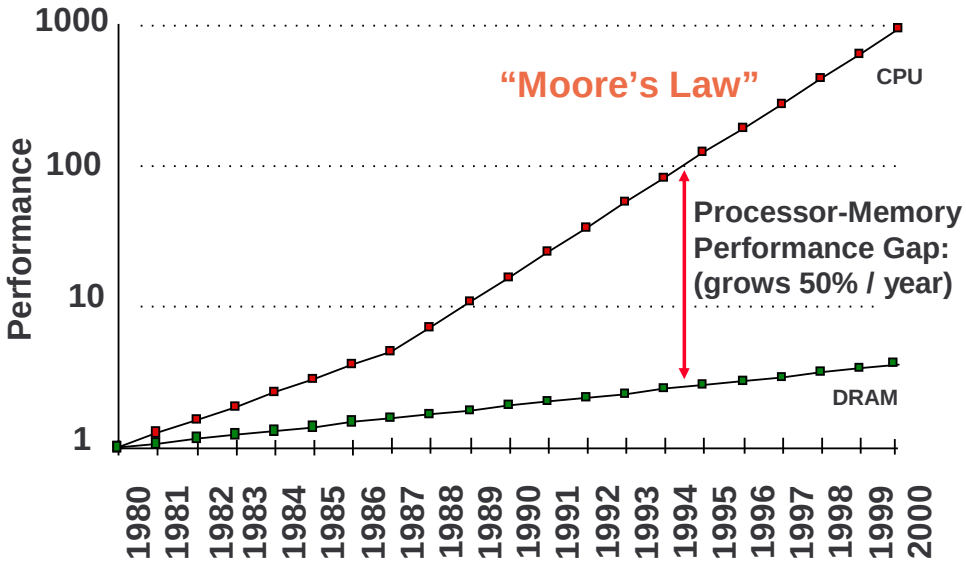


Figure 1.1. The processor and DRAM speed gap keeps increasing.

use of caches to either hide or tolerate the memory latency, including pipelining [77], out-of-order execution [32], multithreading [5] and prefetching. Among all the techniques proposed, prefetching has been one of the most widely studied.

Prefetching, in some form, has existed since the mid-sixties. Early studies [6] of cache design recognized the benefits of fetching multiple words from main memory into the cache. Hardware prefetching of separate cache blocks was later implemented in the IBM 370/168 and Amdahl 470V [88]. Software techniques are more recent. Smith first alluded to this idea in his survey of cache memories [89] but at that time doubted its usefulness. Later, Porterfield [75] proposed the idea of a "cache load instruction" with several RISC implementations following shortly thereafter.

Many prefetching solutions have been proposed in the last 20 years and some of the prefetching mechanisms proposed by researchers have already been implemented in several commercial processors [15, 42, 85].

Prefetching is applicable for both instructions [3, 41, 62, 70, 72, 78, 90, 105, 109, 112, 114, 115] and data [12, 24, 29, 45, 59, 61, 68, 69, 80, 81, 88, 107] and can be controlled by either hardware [12, 24, 29, 74, 80, 81, 84, 88] or software-based [54, 59, 61, 68, 69]

mechanisms. Compared to data prefetching, instruction prefetching is less challenging due to the good locality of instructions during execution. While a number of schemes have been published dealing with instruction prefetching, most of the efforts have been spent on improving the performance of the data cache memory hierarchy.

Both hardware [12, 28, 29, 31, 35, 47, 80, 81, 88] and software [4, 54, 59, 61, 68, 69] techniques have been proposed for data prefetching in recent years. Hardware-based prefetching techniques do not have any instruction overhead but typically require extra hardware such as history tables to record recent memory accesses, and prefetching control logic to predict memory addresses to be prefetched. In contrast, software prefetching requires minimum hardware support but needs explicit prefetching instructions. Software prefetching techniques normally need the help of compiler analyses, inserting explicit prefetch instructions into the executables. Prefetch instructions are supported by most contemporary microprocessors [15, 16, 30, 42, 50, 85, 108].

Data prefetching can be applicable on array-intensive scientific codes and non-scientific general-purpose codes containing many pointer structures. The earlier efforts focused on improving the performance of scientific codes [12, 14, 19, 68, 69] since they normally have highly predictable memory access pattern due to the heavy use of array structures. Data prefetching for pointers [24, 27–29, 44, 61, 80, 81] are focused on establishing the relationship between pointers and the data they point to, which is further complicated by the fact that effective and widely applicable pointer analysis tools are not available. In our work, we will evaluate data prefetching schemes for both array and pointer structures, focusing on making prefetching work effectively for general-purpose programs.

Many recent compiler-assisted data prefetching techniques use profiling as an effective tool to recognize data access patterns for making prefetch decisions. Luk *et al.* [63] uses profiling to analyze executable codes to generate post-link relations

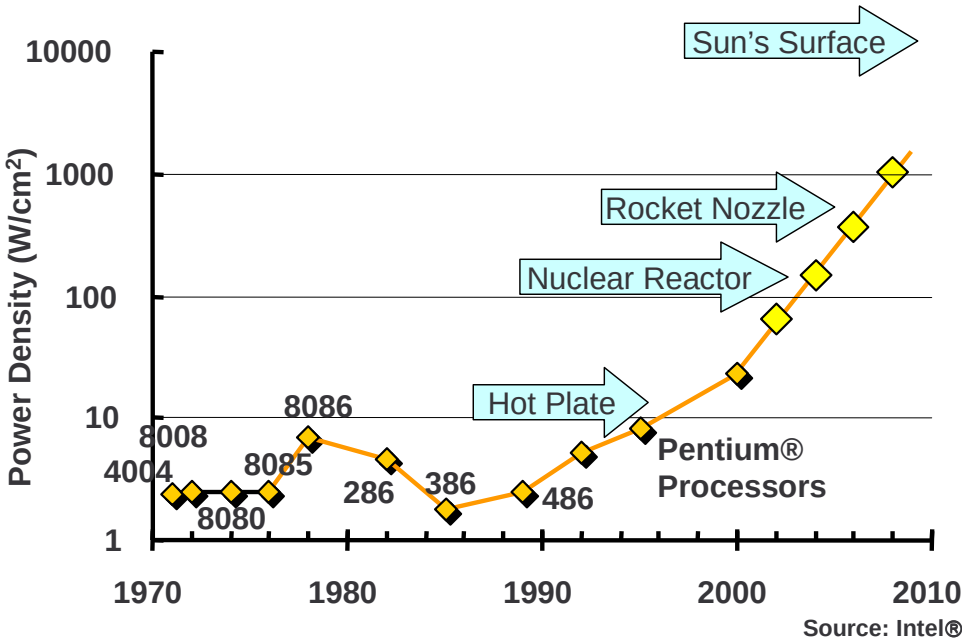


Figure 1.2. Power densities of different Intel Processors.

which can be used to trigger prefetches. Wu [104] proposes a technique which discovers regular strides for irregular codes based on profiling information. Chilimbi *et al.* [25] use profiling to discover dynamic hot data streams which are used for predicting prefetches. Inagaki *et al.* [46] implemented a stride prefetching technique for Java objects. We do not compare to these techniques in this dissertation because our techniques do not need the help of profiling.

1.2 Power/Energy Issues

In the last decade, power consumption has become one of the most important design constraints in microprocessor design. The ever increasing power density (shown in Figure 1.2) on microprocessors introduces many severe problems such as heat dissipation and also impacts the reliability of the chips.

Power consumption can be classified into dynamic power and leakage power dissipation. Dynamic power is consumed during the switching of transistors, while

leakage power is dissipated when the transistors are off. With the scaling of CMOS technology, leakage power has become the dominant factor in power consumption [55]. Many power and energy reduction techniques [48, 64, 71, 73, 110] have been proposed by researchers in the last decade, at both circuit and system levels. Power-aware techniques such as frequency/voltage scaling have already been incorporated into many commercial processors such as the Intel®Pentium®M.

The memory system, including caches, consumes a significant fraction of the total system power. For example, the caches and translation look-aside buffers (TLB) combined consume 23% of the total power in the Alpha 21264 [38], and the caches alone use 42% of the power in the StrongARM 110 [67]. Developing energy-efficient and high-performance memory systems is a key challenge for next-generation processor designs.

Techniques have been proposed to save memory system power/energy at all levels [34, 43, 49, 52, 53, 57, 58, 65, 66, 112, 113]. For example, at the circuit-level, novel power-aware circuits such as CAM-tag design [111] have already been widely adopted in low-power processors. At the higher level, the introducing of techniques such as subbanking [36] can greatly reduce the power dissipation for caches. Compiler-enabled energy-aware solutions have also been developed recently, such as the compiler-enabled cache and memory system designs [9, 10, 95, 96, 101] that can improve cache performance as well as energy consumption.

With the introduction of extra hardware and possibly additional cache/memory accesses, we expect that data prefetching techniques might have a significant impact on the total memory system energy consumption. While most of the prefetching techniques have been focusing on improving the performance, we believe the energy aspects of data prefetching will also become a very important issue in next-generation processor designs as energy and power consumption becomes an equally, if not more, important design constraint as performance.

Most of the current prefetching research work focuses on how to improve performance. Related to our works, a static filter [92] was proposed to reduce memory traffic. Profiling was used to select which load instructions generate data references that are useful prefetch triggers. In our approach, by contrast, we use static compiler analysis and a hardware-based filtering buffer (PFB), instead of profiling-based filters.

Wang *et al.* [98] propose a compiler-based prefetching filtering technique to reduce traffic resulting from unnecessary prefetches. Although the above two techniques have the potential to reduce prefetching energy overhead, there are no specific discussion or quantitative evaluation of the prefetching related energy consumption.

1.3 Research Goals

The goal of this dissertation is to develop data prefetching techniques targeting both performance and energy efficiency. Although numerous data prefetching techniques have been developed to improve the performance of memory systems, as far as we know, the energy aspect of prefetching is almost an untouched area. In this dissertation, we will focus on exploring the energy/performance tradeoffs for data prefetching techniques targeting general-purpose programs containing both array and pointer structures. A very important objective is to develop new energy-aware data prefetching techniques without giving up their performance benefits.

The first issue we want to investigate is to characterize the energy aspects of data prefetching techniques. In order to develop energy-aware data prefetching solutions, we must first understand the data prefetching mechanisms and their implications on energy consumption. We will identify the power-consuming components due to prefetching and evaluate each of them in detail. Both dynamic power and leakage power will be evaluated as leakage will become more and more important in deep sub-micron designs. The evaluation work is the starting point for our attempts to develop new techniques to make data prefetching energy-aware.

The results from our energy evaluation show that most of the energy degradation is due to prefetch-hardware related costs and unnecessary L1 data cache lookups related to prefetches that hit in the L1 cache. Our efforts focus on investigating whether all these extra accesses are necessary and which part of them can be ignored or filtered without affecting performance. As a result, three compiler-based prefetching filtering techniques and one hardware filtering technique are developed to reduce the number of accesses to the energy-consuming components. The filtering techniques combined can reduce a significant part of the extra accesses due to prefetching.

Another direction to make prefetching energy-aware is to develop new prefetching techniques using less power-demanding hardware. We introduce a new Power-Aware pRefetching Engine (PARE) with a novel design of an indexed hardware history table. Compared to the conventional single table design, the new prefetching table consumes 7-11X less power per access. With the help of compiler-based location-set analysis, we show that the proposed PARE design could improve energy consumption significantly.

Surprisingly, the energy consumption is actually reduced with prefetching for some benchmarks based on our evaluation. The reason for this is because the reduction of execution time reduces the leakage energy proportionally. With new energy-aware data prefetching, we can even expect, in some cases, data prefetching could be beneficial for both performance and energy consumption. We will show that this goal is achievable for at least some of the applications.

1.4 Contributions of Dissertation

This dissertation makes the following primary contributions:

- We implement a number of data prefetching techniques and provide detailed simulation results on both performance and energy consumption.

- We modify the SimpleScalar [18] simulation tool to implement the different hardware prefetching techniques and collect statistics on performance as well as switching activity in memory systems. To model the power consumption in the memory system and prefetching hardware, we use state-of-the-art low-power cache circuits and simulate them using HSPICE.
- The simulation results show that although aggressive prefetching techniques help to improve performance, in most of the applications they increase energy consumption by up to 30%. In many systems [38, 67], this is more than 15% increase in chip-wide energy consumption.
- In designs implemented in deep-submicron 70-nm BPTM process technology, cache leakage dominates the energy consumption. We have found that, if cache leakage is optimized with recently-proposed circuit-level techniques, most of the energy overhead is due to prefetch hardware related cost and unnecessary L1 data cache lookups related to prefetches that hit in the L1 cache.
- In addition to the evaluation of hardware-based techniques, we also implement two software data prefetching techniques using SUIF and SimpleScalar. Detailed simulation results are provided to compare the performance and energy consumption between hardware and software prefetching.

- We propose several *energy-aware filtering* techniques for hardware data prefetching to reduce the energy overheads. The techniques include:
  - A *compiler-based selective filtering (CBSF)* approach which reduces the number of accesses to the prefetch hardware by only searching the prefetch hardware tables for selected memory accesses that are identified by the compiler;

- A *compiler-assisted adaptive prefetching(CAAP)* mechanism, which utilizes compiler information to selectively apply different prefetching schemes depending on predicted memory access patterns;
- A compiler-driven filtering technique using a *runtime stride counter(SC)* designed to reduce prefetching energy consumption on memory access patterns with very small strides; and
- A hardware-based filtering technique using a *prefetch filter buffer (PFB)* applied to further reduce the L1 cache related energy overhead due to prefetching.

These techniques are applied on one of the hardware prefetching techniques, which achieves the best performance speedup but also suffers the worst energy degradation. Our experiments show that the proposed techniques successfully reduce the prefetching-related energy overheads, by 40% on average, without reducing the performance benefits of data prefetching.

- To further reduce the energy overheads, we develop a new data prefetching technique called *location-set driven data prefetching*. A *power-aware prefetch engine* called PARE with a novel design of an indexed hardware history table is proposed at the circuit level. Compared to the conventional single-table design, the new prefetching table consumes 7-11X less power per access. With the help of compiler-based location-set analysis, the proposed prefetching scheme can improve energy consumption by as much as 40% in the data memory system.

1.5 Organization of Dissertation

This dissertation comprises the following chapters in addition to this introductory chapter:

Chapter 2 studies a number of data prefetching techniques from an energy perspective. We first evaluate the performance speedup and energy consumption for five hardware-based prefetching techniques. We also evaluate a couple of software prefetching techniques and compare their performance and energy to related hardware techniques.

Chapter 3 explores several filtering techniques that can make the data prefetching techniques more energy-efficient. The techniques proposed include three compiler-assisted prefetch filtering techniques focused on reducing the accesses to the hardware history table and a hardware filtering mechanism trying to reduce the unnecessary extra L1-cache tag checks.

Chapter 4 presents the location-set driven data prefetching techniques using PARE, a novel indexed power-aware prefetch engine. We show that with the help of compile-time location-set analysis, we can divide the memory accesses into different relationship groups, with each group consisting of memory accesses visiting only closely related location-sets. The compiler generated group numbers allow us to use the indexed history table in PARE.

Chapter 5 presents the experimental results on energy-aware data prefetching. We present detailed results on prefetch filtering and PARE power consumption; we also show the energy consumption numbers after applying the proposed techniques. Sensitivity analyses for at 45-nm and 32-nm technologies are also presented.

Finally, Chapter 6 includes a summary of the primary results in this dissertation and a number of directions for future work in this area.

CHAPTER 2

ENERGY CHARACTERIZATION OF DATA PREFETCHING

This chapter evaluates several hardware-based data prefetching techniques from an energy perspective, and explores their energy/performance tradeoffs. We present detailed simulation results and make performance and energy comparisons between different configurations. Power characterization is provided based on HSPICE circuit-level simulation of state-of-the-art low-power cache designs implemented in deep-submicron process technology. This is combined with architecture-level simulation of switching activities in the memory system. The results show that while aggressive prefetching techniques often help to improve performance, they increase energy consumption for most applications. In designs implemented in deep-submicron 70-nm BPTM process technology, cache leakage becomes one of the dominant factors of the energy consumption. We have, however, found that if leakage is optimized with recently-proposed circuit-level techniques, most of the energy degradation is due to prefetch-hardware related costs and unnecessary L1 data cache lookups related to prefetches that hit in the L1 cache. This overhead on the memory system can be as much as 30%.

2.1 Introduction

In recent years, energy and power efficiency has become one of the key design objectives in microprocessors, in both embedded and general-purpose domains. This trend is expected to continue due to the increased power densities of next-generation

deep-submicron designs. In this chapter we consider 70-nm technologies: this is representative of such next generation process technologies. Although a lot of research has been focused on improving the performance of prefetching mechanisms, the impact of such prefetching techniques on processor energy efficiency remains unclear.

Prefetching has been proposed as a technique to hide memory latencies. The idea is to fetch the data to a higher level (for instance, from L2 Cache to L1 Cache) of the memory hierarchy before the data is accessed. Both hardware [12, 29, 80, 81, 88] and software [59, 61, 68] techniques have been proposed for prefetching in recent years. Software prefetching is implemented by inserting explicit prefetch instructions into the executable code. Although there are no hardware requirements for software prefetching (prefetching instructions are supported by most contemporary microprocessors), the compiler process of inserting and scheduling prefetches is complicated. Hardware-based approaches are simpler since they do not require modification to executables. Although hardware prefetching requires extra prefetch hardware in a processor, such additional hardware requirements are typically small.

This chapter evaluates several state-of-the-art hardware-based data prefetching techniques from an energy perspective, and explores their energy/performance tradeoffs. The prefetching techniques studied include:

- Two sequential prefetching approaches [88]: simple *sequential prefetching* (prefetch-on-miss) and *tagged sequential prefetching*;
- *Stride prefetching* [12]: focusing on array-like structures, this technique catches constant strides in memory accesses and prefetching using stride information;
- *Dependence-based prefetching* [80]: designed to prefetch on pointer-intensive programs containing linked data structures where no constant strides can be found; and

- A *combined* stride and dependence-based approach which are focused on both array-intensive and pointer-intensive programs [39].

We present detailed simulation results on each prefetching technique and show performance and energy comparisons. We modify the SimpleScalar [18] simulation tool to implement the different prefetching techniques and collect statistics on performance as well as switching activity in memory systems. To estimate power consumption in the memory systems, we use state-of-the-art low-power cache circuits and simulate them using HSPICE.

As expected, the results show that while aggressive prefetching techniques often help to improve performance, in most of the applications, they increase total memory system energy consumption [1] by as much as 30%. In many systems [38,67], this can be transformed to more than 15% increase in chip-wide energy consumption. In designs implemented in deep-submicron 70-nm BPTM process technology, cache leakage dominates the energy consumption. We have, however, found that if cache leakage is optimized with recently-proposed circuit-level techniques, most of the still remaining energy degradation is due to prefetch hardware related cost and unnecessary L1 data cache lookups related to prefetches that hit in the L1 cache. When the energy cost of off-chip accesses is increased to more pessimistic levels (e.g., due to very large load capacitances driven during off-chip accesses), the other energy effects are becoming less visible.

In addition to the evaluation of hardware-based data prefetching techniques, we also include in this chapter the energy evaluation of two software prefetching techniques and compare their performance and energy consumption with hardware prefetching techniques.

---

[1] We consider energy-consumption of L1 and L2 caches and prefetching related hardware. Main memory is not included because there does not exist an accurate energy model for it. Nevertheless, we have also included a section describing how main memory energy consumption would affect the total energy memory overhead in this chapter.

The rest of this chapter is structured as follows. Section 2.2 presents a brief introduction of the prefetching techniques we study in this chapter. The power estimation of memory system aspects and experimental framework are presented in Section 2.3. Section 2.4 gives a detailed analysis of the experimental results on hardware data prefetching technique. We compare software and hardware prefetching in Section 2.5. We summarize this chapter with Section 2.6.

2.2 Data Prefetching Mechanisms

This section gives a brief introduction to the data prefetching schemes studied. The information provided here is mainly for the purpose of analyzing their energy overhead. Detailed information on these techniques should be referred to the respective papers. The prefetching techniques studied include five hardware-based data prefetching and two software prefetching techniques.

2.2.1 Hardware-based Data Prefetching

2.2.1.1 Sequential Prefetching

Prefetching schemes are designed to fetch data from a lower level in the memory hierarchy to a higher level (i.e., from main memory to cache, from L2-cache to L1-cache). It should be noted that multiple-word cache blocks are themselves a form of data prefetching. However, the size of the cache blocks is limited by cache pollution effects as the cache block size increases.

Sequential prefetching can take advantage of spatial locality without introducing some of the problems associated with large cache blocks. The simplest sequential prefetching schemes are based on the *One Block Lookahead* (OBL) approach; a prefetch for block $b + 1$ is initiated when block b is accessed. OBL implementations differ based on what type of access to block b initiates the prefetch of $b + 1$. In this

dissertation, we evaluate two of the sequential approaches discussed by Smith [88] - *prefetch-on-miss* and *tagged prefetching*.

The prefetch-on-miss algorithm simply initiates a prefetch for block $b + 1$ whenever an access for block b results in a cache miss. If $b + 1$ is already cached, no memory access is initiated. The tagged prefetching algorithm associates a tag bit with every memory block. This bit is used to detect when a block is demand fetched or a prefetched block is referenced for the first time. In either of these cases, the next sequential block is prefetched.

Smith found that tagged prefetching reduces cache misses in a unified (both instruction and data) cache by between 50% and 90% for a set of trace-driven simulations. The prefetch-on-miss technique is no more than half as effective as tagged prefetching in reducing miss rates. We will use it as our prefetching baseline since there is virtually no hardware cost, and thus no power overhead in prefetching hardware. Tagged prefetching does require one extra bit for each cache line to indicate whether the cache line is prefetched or not.

OBL prefetching schemes are not as efficient as more recent schemes but they require only very simple hardware. An OBL scheme was implemented in the HP PA7200 [21]; it uses a modified version of tagged prefetching scheme and shows significant performance improvement for some benchmarks.

In general, sequential hardware prefetching techniques perform poorly when non-sequential memory access patterns are encountered. Scalar references or array accesses with large strides can result in unnecessary prefetches because these types of access patterns do not exhibit the spatial locality upon which sequential prefetching is based. To enable prefetching of strided and irregular data access patterns, several more elaborate hardware prefetching techniques have been proposed, as we discuss next.

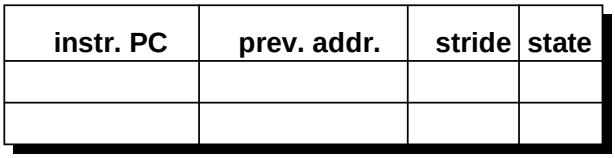


Figure 2.1. Structure of Reference Prediction Table(RPT) used for stride prefetching

2.2.1.2 Stride Prefetching

Stride prefetching [12] monitors memory access patterns in the processor to detect constant-stride array references originating from loop structures. This is normally accomplished by comparing successive addresses used by memory instructions.

Since stride prefetching requires the previous address used by a memory instruction to be stored along with the last detected stride, a hardware table called the *Reference Prediction Table*, or RPT, is added to hold the information for the most recently used memory instructions. A representation of RPT is shown is Figure 2.1. Each RPT entry contains the address of the memory instruction, the address of the instruction as accessed previously, a stride value for those entries that have established a stride, and a state field used to control the actual prefetching. The LRU replacement algorithm is be used to select a victim entry when the table is full.

Stride prefetching is more selective than sequential prefetching since prefetch commands are issued only when a matching stride is detected. It is also more effective when array structures are accessed through loops. However, stride prefetching uses an associative hardware table which normally contains 64 entries; each entry contains around 64 bits. This hardware table is accessed whenever a load instruction is detected.

2.2.1.3 Pointer Prefetching

Stride prefetching has been shown to be effective for array- intensive scientific programs. However, for general-purpose programs, which are pointer-intensive or

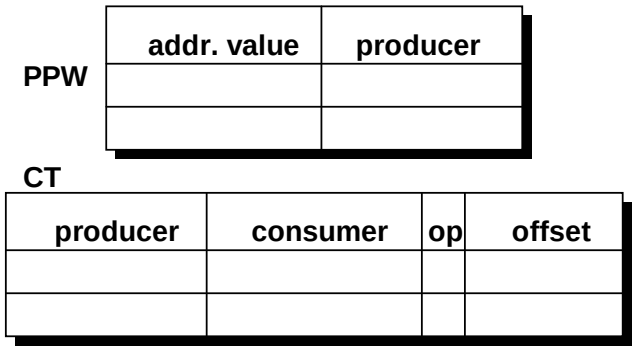


Figure 2.2. Structure of Potential Producer Window (PPW) and Correlation Table (CT) used for dependence-based prefetching

contain a large number of dynamic data structures, no constant strides can be easily found. Thus stride prefetching is not as effective for these applications.

One scheme for hardware-based prefetching on pointer structures, called *dependence-based prefetching*, is proposed by Roth et al. [80]. Like stride prefetching, this scheme uses hardware tables to record the most recently executed load instructions. The difference is that this table is used to detect dependencies between load instructions rather than establishing reference patterns for single instructions.

The hardware tables used by dependence-based prefetching are shown in Figure 2.2. The Correlation Table (CT) is the component responsible for storing dependence information. Each correlation represents a dependence between a load instruction that produces an address (producer) and a subsequent load that uses that address (consumer). In addition to producer and consumer identities (PCs), each correlation also contains an operator and offset field to represent the address generation template which is used to calculate future prefetching addresses. The Potential Producer Window (PPW) records the most recent loaded values and the corresponding instructions. When a load commits, its base address value is checked against the entries in the PPW, with a correlation created on a match. This correlation is added to the CT as an entry containing dependence information.

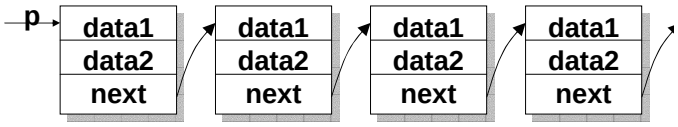


Figure 2.3. Dependence-based prefetching on linked data structures

Such address dependencies are common in linked list processing, i.e., where the next pointer of one list element is used as the base address for the fields of the subsequent list element, as shown in Figure 2.3. Once CT establishes these dependencies, prefetches are triggered by the execution of those load instructions that produce base addresses. For example, once the address of $p \rightarrow next$ is known, $p \rightarrow next \rightarrow data1$ and $p \rightarrow next \rightarrow data2$ can be prefetched. A prefetch of $p \rightarrow next \rightarrow next$ can also be used to initiate further prefetching.

PPW and CT are typically 64-128 entries containing addresses and program counters; each entry may contain 64 or more bits. The hardware cost is around twice that for stride prefetching. This scheme improves performance on many of the Olden [79] pointer-intensive benchmarks.

2.2.1.4 Combined Stride and Pointer Prefetching

One contribution of our work is a combined stride and pointer prefetching technique [39]. Our objective is to evaluate a technique that is beneficial for applications containing both array and pointer based accesses.

We will show that the combined technique performs consistently better than the individual techniques on two benchmark suites with different characteristics. However, the hardware cost of this approach is higher since we need to use the hardware tables from both stride and dependence-based prefetching.

2.2.2 Software Prefetching

Software prefetching techniques rely on compiler analysis to predict prefetching addresses and generate explicit prefetching instructions. We implemented two software prefetching techniques on both array-based structures and pointer-intensive programs [61, 69].

2.2.2.1 Software Prefetching for Arrays

The first technique we implemented is a *Compiler-based prefetching* technique similar to Todd Mowry's work [69].

A compiler analysis is implemented to analyze the array access pattern within loops and determine the possible cache misses and therefore prefetch addresses. The main steps are as follows:

- For each reference, determine the accesses that are likely to be cache misses and therefore need to be prefetched.
- Isolate the predicted cache miss instances through loop splitting. This avoids the overhead of adding conditional statements to the loop bodies.
- Performance stride and reuse analysis on the predicted cache miss instances to determine the stride between two accesses.
- Calculate prefetch addresses for the cache misses instances based on the stride information.

Prefetch addresses will be calculated and explicit prefetch instructions are inserted in the binaries afterwards.

2.2.2.2 Software Prefetching for Pointers

Compiler-based prefetching on Linked Data Structures is proposed by Luk and Mowry [61] to prefetch pointer structures.

The greedy approach in [61] analyzed recursive data structure types and insert prefetching instructions to fetch the next data element in the linked structures. The greedy approach issues prefetches for all the elements within a linked data structure once a link to the data structure is accessed. This scheme has the following main steps:

- Identify large linked data structures during static compiler analysis.
- Determine data type information for each memory access via type analysis.
- Based on the type information obtained, calculate prefetch addresses if the accessed data is one of the linked data structures identified during the first step.
- Issue explicit prefetch instructions into the binaries.

For software prefetching, prefetch instructions in the binaries will be identified in the hardware and prefetching requests will be issued in the same way as we do for hardware prefetching. The only difference is that prefetching addresses are provided by the instructions, instead of calculated by the hardware.

2.3 Experimental Assumptions

Before we discuss the evaluation of performance/energy aspects of the described data prefetching techniques, we first present in this section the experimental framework, the energy modeling of data prefetching, and our methods for energy calculation. The experimental framework and energy modeling methods will also be used in all the following sections as well.

Table 2.1. Baseline parameters

Processor speed	1GHz
Issue	4-way, out-of-order
L1 D-cache	32KB, CAM-tag, 32-way, 32bytes cache line
L1 I-cache	32KB, 2-way, 32bytes cache line
L1 cache latency	1 cycle
L2 cache	unified, 256KB, 4-way, 64bytes cache line
L2 cache latency	12 cycle
Memory latency	100 cycles latency + 10 cycles/word

2.3.1 Experimental Framework

We implement the hardware-based data prefetching techniques by modifying the SimpleScalar [18] simulator[2]. The binaries input to the SimpleScalar simulator are created using a native Alpha assembler. The parameters we use for the simulations are listed in Table 2.1.

The benchmarks evaluated are listed in Table 2.2. The SPEC2000 benchmarks [2] use mostly array-based data structures, while the Olden benchmark suite [79] contains pointer-intensive programs that make substantial use of linked data structures. We randomly select a total of ten benchmark applications, five from SPEC2000 and five from Olden. For SPEC2000 benchmarks, we fast forward the first one billion instructions and then simulate the next 100 million instructions. The Olden benchmarks are simulated to completion except for *perimeter*, since they complete in relatively short time.

2.3.2 Energy Modeling

To accurately estimate power and energy consumption in the L1 and L2 caches, we perform circuit-level simulations using HSPICE. We base our design on a recently

[2] "sim-outorder" from SimpleScalar version 3.0vd.

Table 2.2. SPEC2000 & Olden benchmarks

Benchmark	Description
SPEC2000	
181.mcf	Combinatorial Optimization
197.parser	Word Processing
179.art	Image Recognition / Neural Nets
256.bzip2	Compression
175.vpr	Versatile Place and Route
Olden	
bh	Barnes & Hut N-body Algorithm
em3d	Electromagnetic Wave Propagation
health	Colombian Health-Care Simulation
mst	Minimum Spanning Tree
perimeter	Perimeters of Regions in Images

proposed low-power circuit [13] that we simulated in 70-nm BPTM technology. Our L1 cache includes the following low-power features: low-swing bitlines, local word-line, CAM-based tags, separate search lines, and a banked architecture. The L2 cache we evaluate is based on a banked RAM-tag design.

As we expect that implementations in 70-nm technology would have significant leakage, we apply a recently proposed circuit-level leakage reduction technique called asymmetric SRAM cells [11]. This is necessary because otherwise our conclusions would be skewed due to very high leakage power. The *speed enhanced cell* in [11] has been shown to reduce L1 data cache leakage by 3.8X for SPEC2000 benchmarks with no impact on performance. For L2 caches, we use the *leakage enhanced cell* which increases the read time by 5%, but can reduce leakage power by at least 6X. In our evaluation, we assume speed-enhanced cells for L1 and leakage enhanced cells for L2 data caches, by applying the different asymmetric cell techniques respectively.

The power consumption numbers of our L1 and L2 caches are shown in Table 2.3. We show the configurations of L1 and L2 and their dynamic and leakage power numbers respectively. If an L1 miss occurs, energy is consumed not only in L1 tag-

Table 2.3. Cache configuration and power consumption

Parameter	L1	L2
size	32KB	256KB
tag array	CAM-based	RAM-based
associativity	32-way	4-way
bank size	2KB	4KB
# of banks	16	64
cache line	32B	64B
Power (mW)		
tag	6.5	6.3
read	9.5	100.5
write	10.3	118.6
leakage	3.1	23.0
reduced leakage	0.8	1.5

Table 2.4. Prefetch hardware table and power consumption

Table implementation	64×64 CAM-array
P_update (including lookup)	11.5mW
P_lookup	11.3mW

lookups, but also when writing the requested data back to L1. L2 accesses are similar, except that an L2 miss goes to off-chip main memory. Leakage energy is consumed for all processor cycles no matter an L1 or L2 cache access occurs or not.

An off-chip memory access consumes a significant amount of processor power. Rather than picking a single design-point, we choose a range of energy costs ranging from optimistic to pessimistic. We express the L2 miss energy as a function of L1 hit energy. We assume that an L2 cache miss consumes 32X to 512X single-word read energy of our L1 cache. A similar assumption has been made in [111]. The actual power consumed depends on how many bits are in transition and on the actual implementation/packaging choices.

Each prefetching history table is implemented as a 64×64 fully-associated CAM-array. The power consumption for each lookup is 7.3mW and each update to the table costs 7.4mW based on HSPICE simulation. The power numbers are shown in Table 2.4. The leakage energy of these hardware tables are very small compared to L1 and L2 caches due to their small area.

For software prefetching, the cost of the execution of a prefetch instruction includes an access to the L1 instruction cache by the prefetch instruction, and the pipeline cost of instruction fetching, decoding, and the calculation of prefetching addresses. These extra costs will increase the total energy consumption. Each L1 instruction cache access consumes about the same energy as an L1 data cache access, and the rest of the execution costs is generally comparable to an L1 data cache access [17]. Thus we assume that each prefetch instruction executed would consume an extra cost of roughly two times the L1 cache read energy cost.

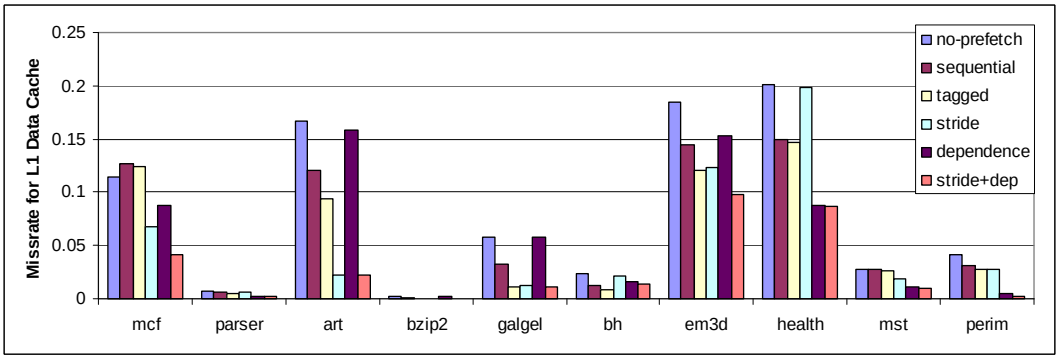
2.4 Analysis of Hardware Data Prefetching

2.4.1 Performance Speedup

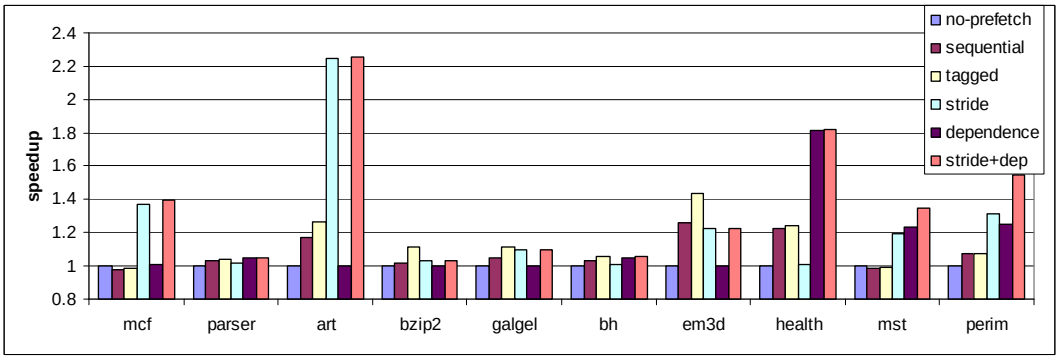
Performance speedup is the original and still the primary goal of prefetching. Fig. 2.4 shows the performance results of different prefetching schemes. The first five benchmarks are array-intensive SPEC2000 benchmarks, and the last five are pointer-intensive Olden benchmarks. Fig. 2.4(a) shows the reduction of DL1 miss-rate, and Fig. 2.4(b) shows actual speedup based on simulated execution time.

As expected, the dependence-based approach does not work well on the five SPEC2000 benchmarks since pointers and linked data structures are not used frequently. But it still gets marginal speedup on three benchmarks (*parser* is the best with almost 5%).

Tagged prefetching (10% speedup on average) does slightly better on SPEC2000 benchmarks than the simplest sequential approach, which achieves an average speedup



(a)



(b)

Figure 2.4. Performance speedup for different prefetching schemes: (a) L1 miss rate reduction; (b) IPC speedup.

of 5%. Stride prefetching yields up to 124% speedup (for *art*), averaging just over 25%. Combined prefetching is the best, but gives on the average only about 1.5% speedup compared to the stride approach. The comparison between miss rate reduction in Fig. 2.4(a) and speedup in Fig. 2.4(b) matches our intuition that fewer cache misses means greater speedup.

As for the five Olden pointer-intensive benchmarks in Fig. 2.4, the dependence-based approach eliminates about half of all the L1 cache misses and achieves an average speedup of 27%. Stride prefetching (14% on average) does surprisingly well on this set of benchmarks and implies that even pointer-intensive programs contain significant constant-stride memory access sequences. The combined approach achieves an average of 40% performance speedup on the five Olden benchmarks.

In summary, for array-intensive programs, stride prefetching does reasonably well and dependence-based pointer prefetching is not very effective. However, for pointer-intensive programs, both stride and dependence-based approaches do well. On the 10 benchmarks simulated, the combined approach achieves the best performance speedup due to prefetching. In general, the combined technique is useful for general purpose programs which contain both array and pointer structures.

2.4.2 Memory Traffic Increase

Memory traffic is increased because not all the data we prefetch from the next level are useful (i.e., the data are not always actually used by a later access before they are replaced from the cache). In most cases, some useless data is prefetched into the higher levels of the memory hierarchy; these are a major source of power/energy consumption added by the prefetching schemes. Apart from memory traffic increases, power is also consumed when we attempt to prefetch the data that already exists in the higher level cache. In this case, the attempt to locate the data (e.g., cache-tag lookup) consumes power.

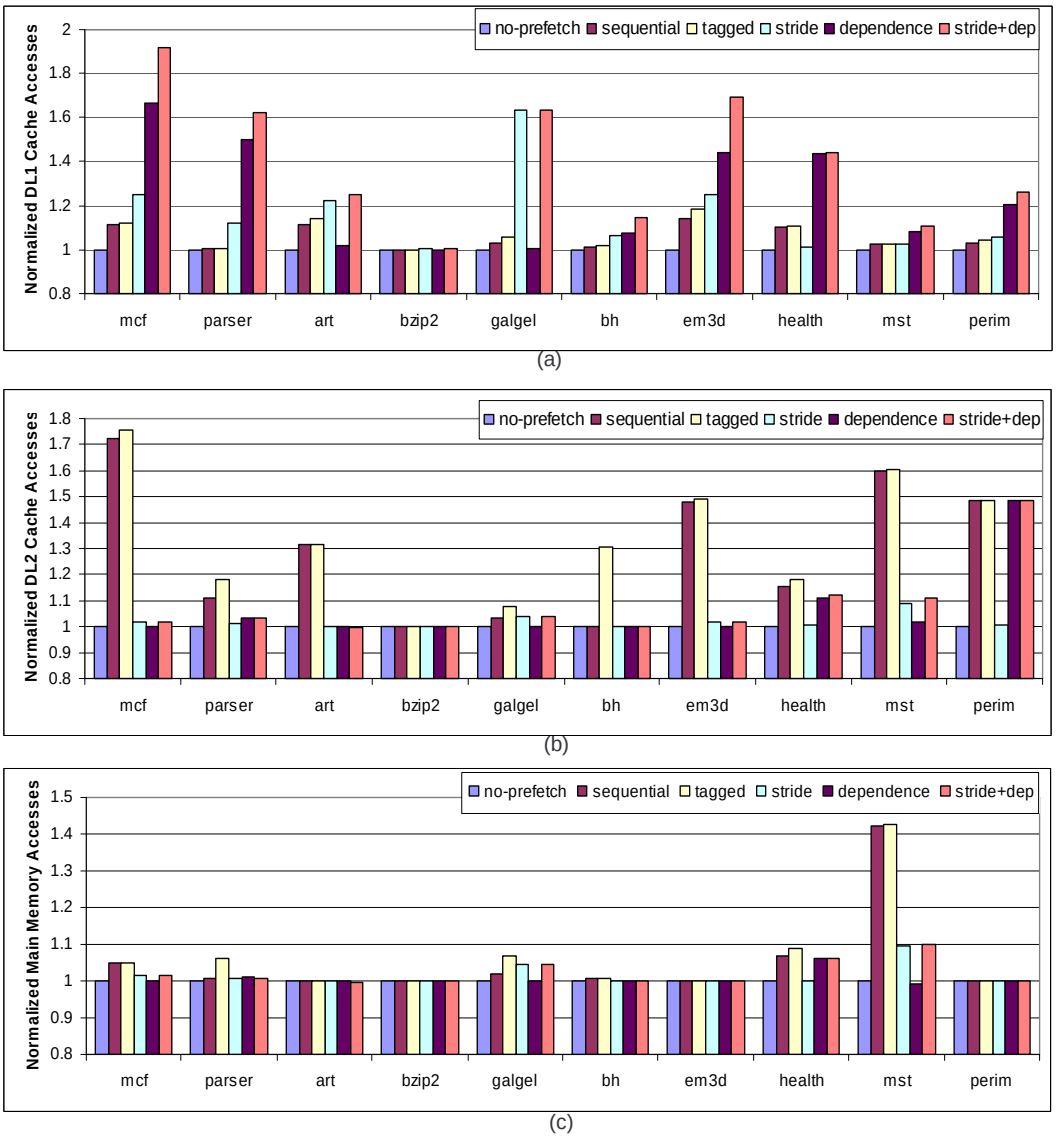


Figure 2.5. Memory traffic increase for different prefetching schemes. (a) Number of accesses to L1 data cache, including extra cache-tag lookups to L1; (b) Number of accesses to L2 data cache; (c) Number of accesses to main memory.

Fig. 2.5 shows the number of accesses going to different levels in the memory hierarchy. The numbers are normalized to the baseline with no prefetching. On average, the number of accesses to L1 D-cache increases almost 40% with the most aggressive prefetching scheme. However, the accesses to L2 only increase by 8% for the same scheme, showing that most of the L1 cache accesses are only cache-tag lookups trying to prefetch data already present in L1.

Sequential prefetching techniques (both prefetch-on-miss and tagged schemes) show completely different behavior as they increase the L1 access for only about 7% while resulting in a more than 30% average increase on L2→L1 traffic. The explanation for this is that sequential prefetching always tries to prefetch the next cache line which has a much greater chance to miss in L1. Main memory accesses are largely unaffected in the last three techniques, and only increased by 5-7% for sequential prefetching.

As L1 accesses increase significantly for the three most effective techniques, we break down the number of L1 accesses into three parts: regular L1 accesses, L1 prefetch misses and L1 prefetch hits, shown in Fig. 2.6. The L1 prefetch misses are those prefetching requests that go to L2 and actually bring cache lines from L2 to L1, while the L1 prefetch hits stand for those prefetching requests that hit in L1 and no real prefetching occurs.

From Fig. 2.6, we can see that L1 prefetching hits account for most of the increases in L1 accesses. On average, 70-80% of all the increases come from extra L1 prefetching hits, which may result in significant energy overhead, while being almost useless for performance speedup. The extra L1 accesses will obviously translate into unnecessary energy consumption.

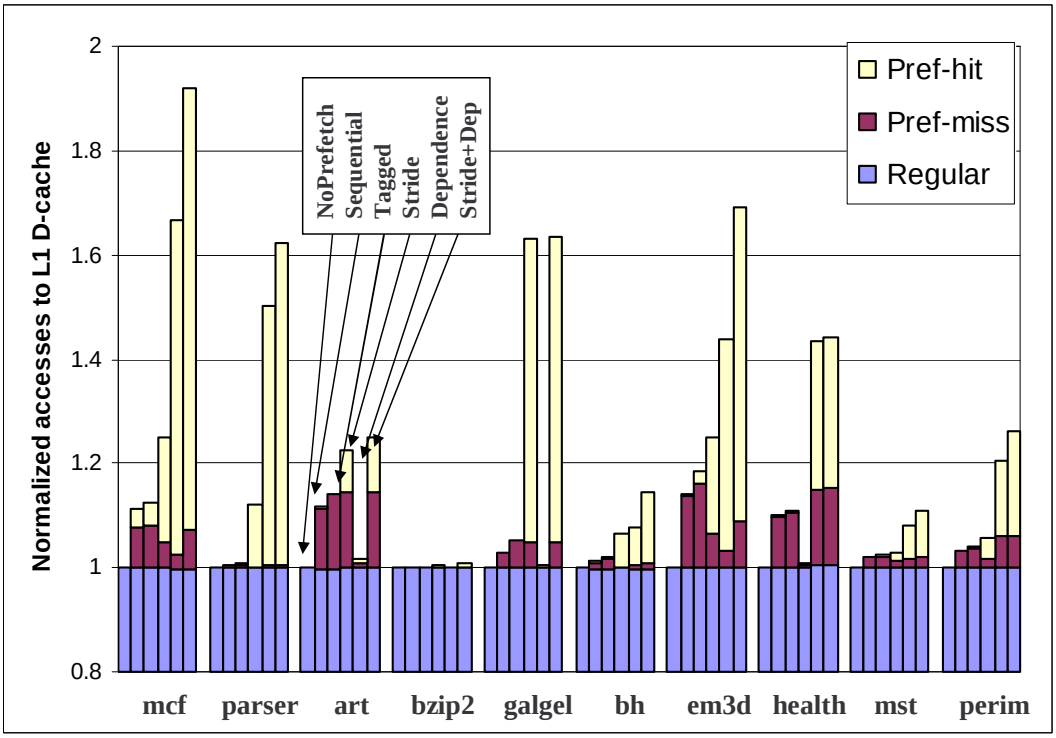


Figure 2.6. Breakdown of L1 Accesses, all numbers normalized to L1 cache accesses of baseline with no prefetching.

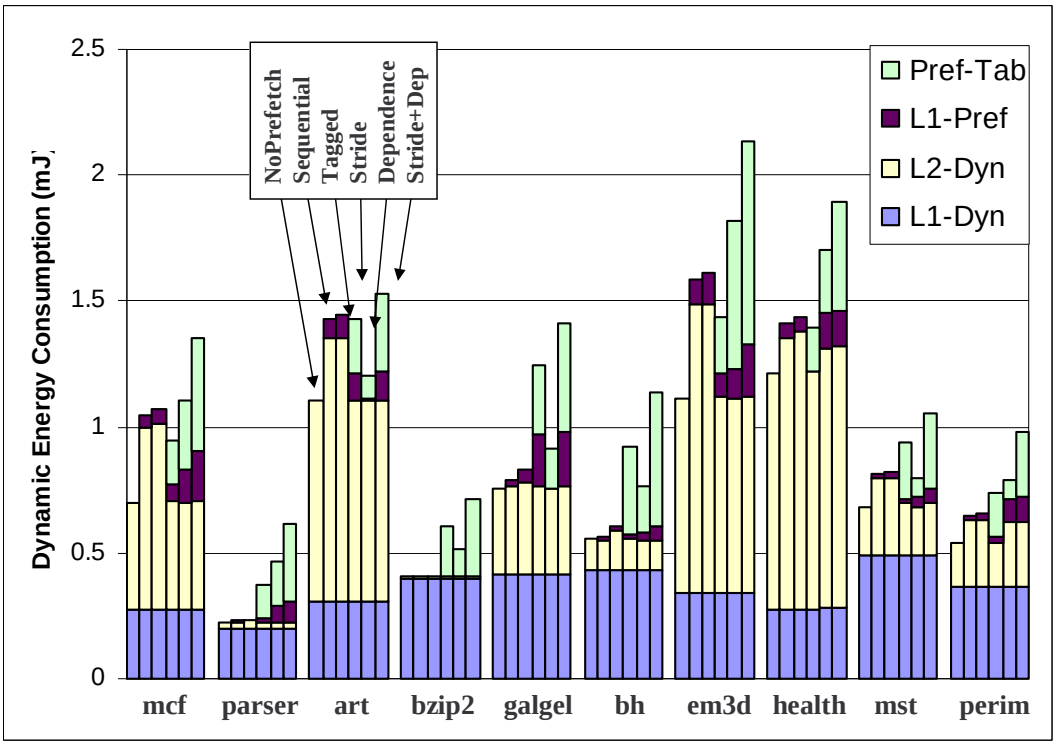


Figure 2.7. Total cache energy consumption without considering leakage energy.

2.4.3 Energy Consumption Overhead

We use the power numbers shown in Section 2.3 to calculate the energy consumption.

2.4.3.1 Cache energy consumption

Fig. 2.7 shows the dynamic energy consumption for L1 and L2 caches and prefetching tables. For most of the benchmarks, the L1 dynamic energy (excluding prefetching overhead) is not affected significantly. The L2 dynamic energy is increased in proportion to the L2 memory traffic increase shown in Fig. 2.5(b). Prefetching related energy overhead on L1 cache is quite small for sequential prefetching, but more significant for the other three prefetching approaches. This part of the energy overhead is proportional to the prefetch-related L1 access increase shown in Fig. 2.6.

Energy consumption for the hardware tables is very significant for all the three prefetching techniques using hardware tables. On average, the hardware tables consume almost the same amount of energy as regular L1 caches accesses for the combined prefetching. Typically this portion of energy accounts for 60-70% of all the dynamic energy overhead that results from combined prefetching. The reason is that prefetch tables are frequently looked up and are also highly associative. Their power consumption is similar to a tag lookup in a banked low power cache.

For the most aggressive combined prefetching approach, the prefetching energy overhead almost doubles the total dynamic energy (baseline with no prefetching) for some applications (such as *mcf* and *em3d*), and is 76% on the average. For the other prefetching techniques, there is a 25% increase for sequential prefetching, and about 38% for both stride and dependence schemes. This shows that while complicated prefetching algorithms can achieve greater speedups, they can significantly increase the overall energy consumption.

Fig. 2.8 shows the total cache energy consumption with leakage energy also accounted. Leakage energy is proportional to program runtime and thus decreases linearly with speedup: higher speedup will reduce the leakage energy consumption.

In this figure, the total energy consumption for caches is dominated by L2 leakage because of the large size (256KB) of the L2 cache. As we can see, for most of the applications, the relative prefetching overhead shown in Fig. 2.7 has been significantly reduced after the leakage energy is taken into account.

With no leakage optimization, sequential prefetching saves on average about 10% of the total energy, stride prefetching about 17% and the combined approach results in almost 24% energy savings. The results show that prefetching schemes which have a better performance speedup also save energy when leakage energy increases to a certain level in deep sub-micron technologies.

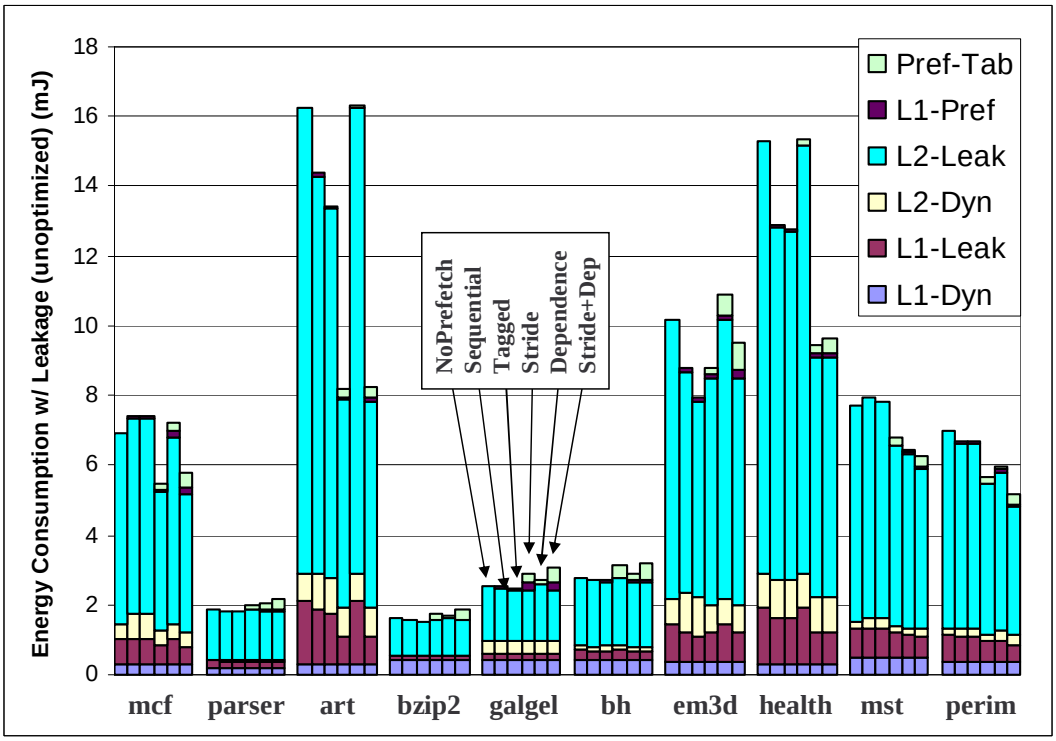


Figure 2.8. Total cache energy consumption with unoptimized leakage energy accounted.

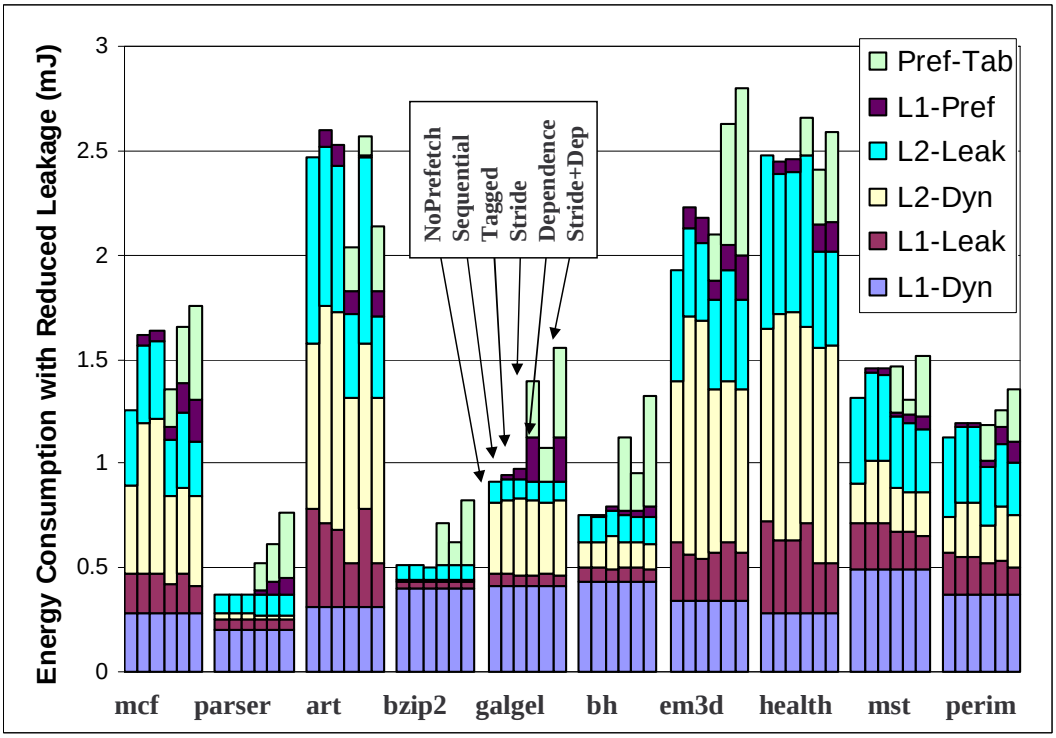


Figure 2.9. Total cache energy consumption with leakage reduction techniques applied.

However, leakage energy could be reduced significantly by techniques such as asymmetric SRAM cells [11]. Fig. 2.9 shows the total cache energy after applying the above leakage reduction techniques. The dynamic hit energy dominates some of the benchmarks with higher IPC; however, the leakage energy still dominates in some programs, such as *art*, which have a higher L1 miss rate and thus a longer running time. Although both L1 and L2 cache access energy are significantly increased due to prefetching, the static (leakage) energy reduction due to performance speedup can compensate for at least some portion of the increase in dynamic energy consumption.

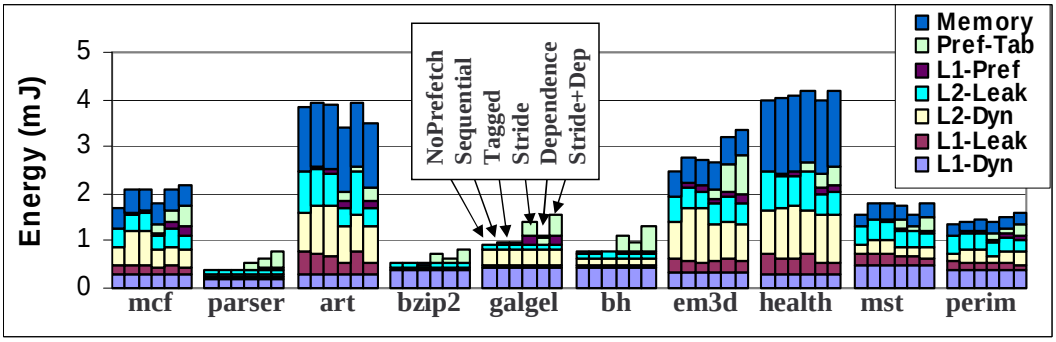
The results in Fig. 2.9 show that on average, the prefetching schemes still cause relatively significant energy consumption overhead when leakage consumption is reduced to a reasonable level. The average increase of the combined approach is more than 26%, and about 11% increase for stride prefetching.

2.4.3.2 Energy cost for off-chip accesses

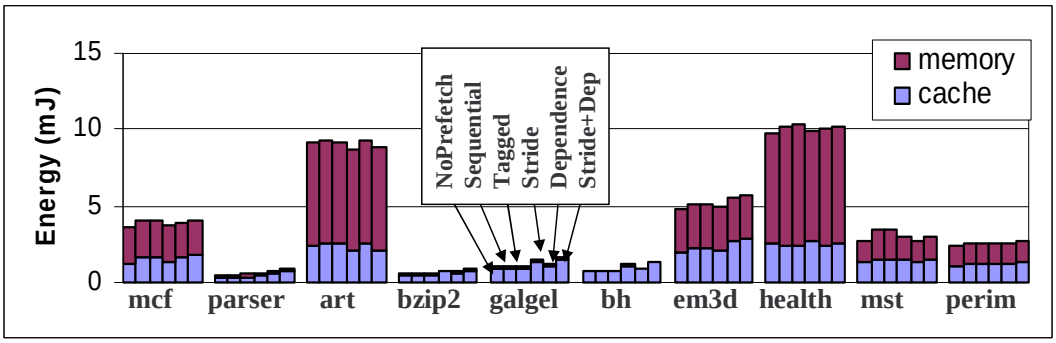
To estimate the energy consumption within the processor for driving off-chip memory accesses, we use similar assumptions as in [111]. We assume that an L2 cache miss consumes 32-512X single-word read energy of the L1 D-cache. Our results, including energy consumption for both caches and off-chip memory access related power, are shown in Fig. 2.10.

Fig. 2.10(a) shows the situation where L2 miss energy cost is 32X of L1 hit energy. The prefetching energy overhead is quite significant for many applications, averaging 7% for sequential prefetching, 8% for stride prefetching and more than 20% for the combined approach.

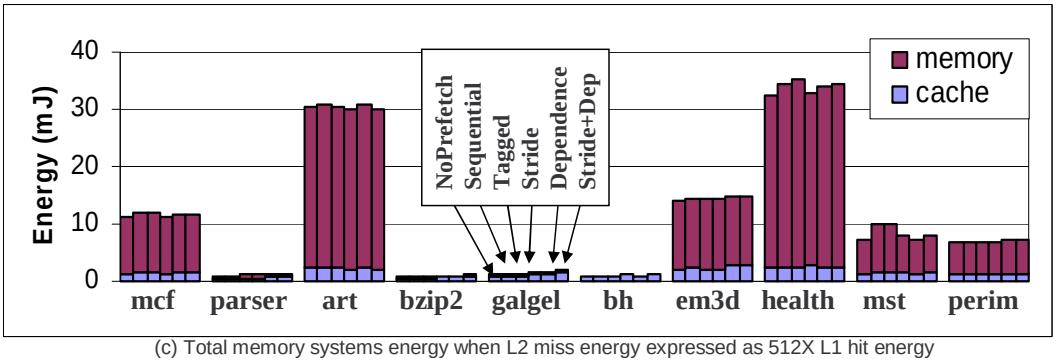
When the off-chip memory costs goes up to 128X, as shown in Fig. 2.10(b) the prefetching overhead stays at 7% for sequential techniques, but drops to almost half for the last three schemes, averaging about 11% for combined prefetching. If the



(a) Total memory systems energy when L2 miss energy expressed as 32X L1 hit energy

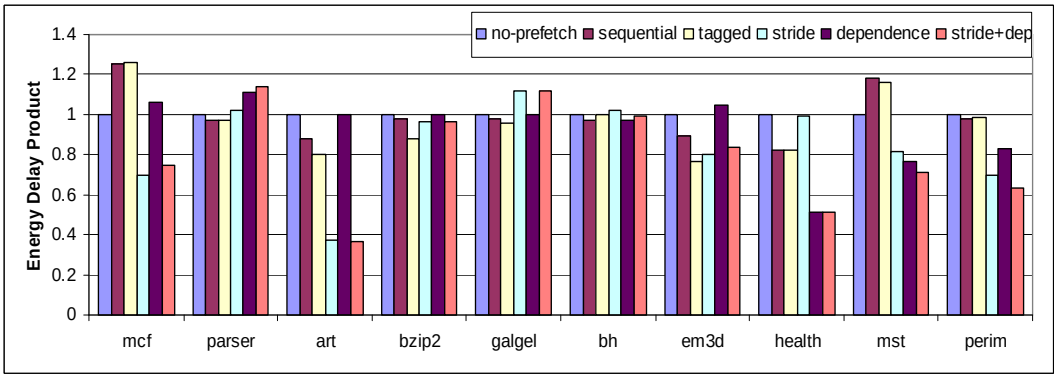


(b) Total memory systems energy when L2 miss energy expressed as 128X L1 hit energy

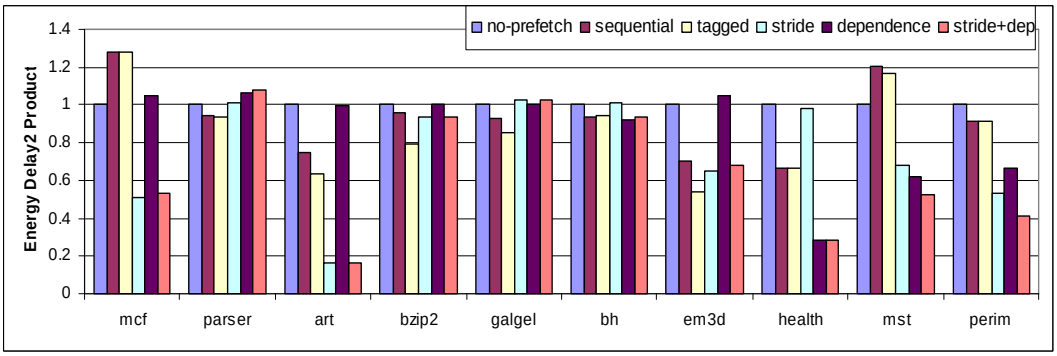


(c) Total memory systems energy when L2 miss energy expressed as 512X L1 hit energy

Figure 2.10. Total energy consumption for memory systems with varying L2 miss energy cost.



(a)



(b)

Figure 2.11. Energy-delay product for different prefetching techniques. (1) Energy-delay product; (2) Energy-delay^2 product. In both figures, we assume that the leakage reduction techniques are applied and the off-chip memory energy cost is 32X of the L1 hit energy.

off-chip memory costs were to increase to a pessimistic 512X as shown in Fig. 2.10(c), the energy overhead of prefetching drops to less than 5%.

2.4.3.3 Energy-delay product

Finally, we show in Fig. 2.11 the energy-delay and energy-delay^2 product normalized to the baseline (no prefetching) using the assumption that L2 miss energy is 32X L1 hit energy.

In most cases, we note that both energy-delay and energy-delay^2 products improve with effective prefetching techniques that achieve a large enough performance speedup. The energy-delay product improves by more than 20% for the combined

prefetching, while the energy-delay^2 improves by almost 35%. This is important since by choosing a design point with lower voltage, this could be converted into energy efficiency. Nevertheless, we believe that more energy-focused prefetching algorithms and architectures should be developed to achieve energy efficiency even at unchanged voltage levels.

According to this figure, extra energy cost by complicated prefetching techniques are worthwhile for some applications such as the combined prefetching approach on *mcf* and *em3d*.

2.5 Comparison on Hardware/Software Prefetching

The comparison between hardware and software prefetching techniques will be presented in this section. We provide experimental results for the following five prefetching techniques:

- *Stride prefetching* [12] - Focuses on array-like structures, it catches constant strides in memory accesses and prefetches using the stride information;
- *Dependence-based prefetching* [80] - Designed to prefetch on pointer-intensive programs containing linked data structures where no constant strides can be found;
- A *combined* stride and dependence-based approach - Focuses on general-purpose programs, which often use both array and pointer structures, to achieve benefits from both stride and pointer prefetching.
- *Compiler-based prefetching* similar to [69] - Use the compiler to insert prefetch instructions for strided array accesses.
- *Compiler-based prefetching on Linked Data Structures* - Uses the greedy approach in [61] to prefetch pointer structures.

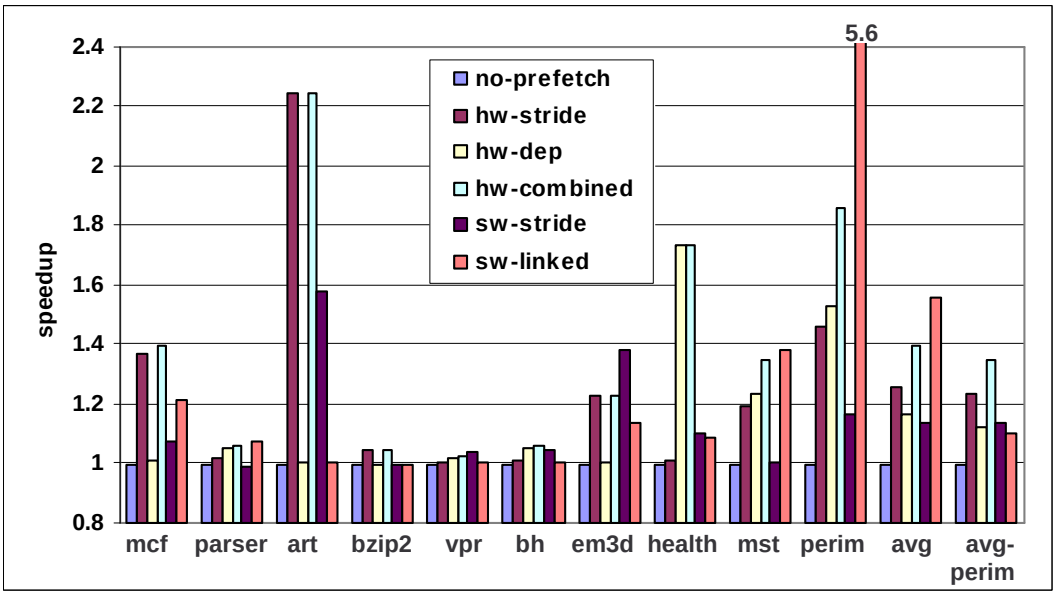


Figure 2.12. Performance speedup for different prefetching schemes.

The first three techniques are hardware-based and they require the help of one or more hardware history tables to trigger prefetches. The last two are software-based techniques which use compiler analysis to decide what addresses should be prefetched and where in the program to insert the prefetch instructions.

The performance improvement of the five prefetching techniques is shown in Figure 2.12. The first five benchmarks are from SPEC2000 benchmarks; the last five are Olden benchmarks which contains many pointers and linked data structures.

Stride prefetching does very well on performance for SPEC2000 benchmarks, averaging just over 25% speedup across the five applications studied. In contrast, the dependence-based approach achieves an average speedup of 27% on the five Olden benchmarks. The combined approach achieves the best performance speedup among the three hardware techniques, averaging about 40%. In general, the combined technique is the most effective approach for general-purpose programs (which typically contain both array and pointer structures).

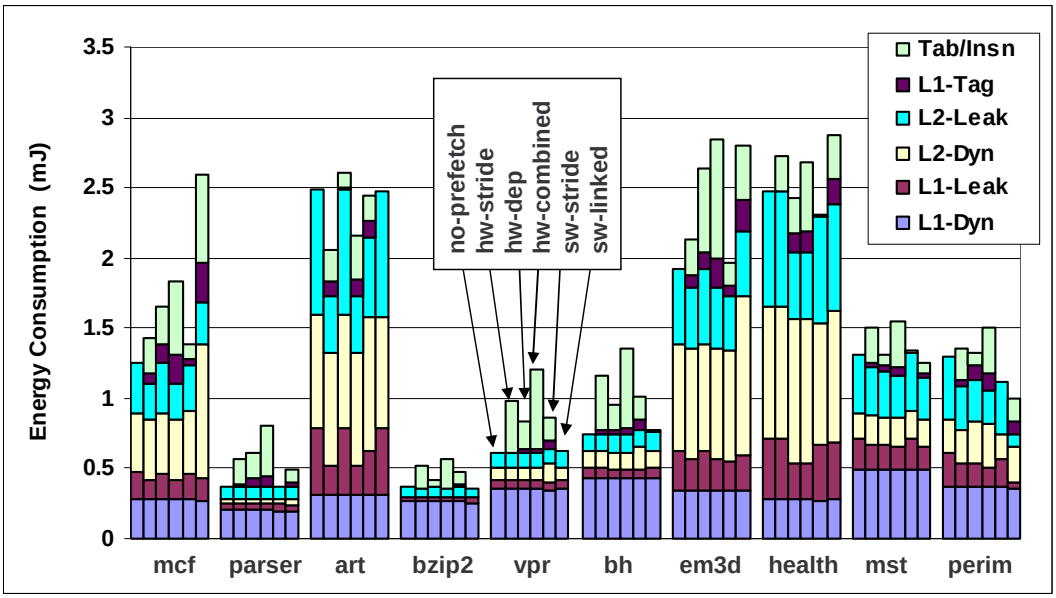


Figure 2.13. Total cache energy consumption.

For the two software techniques, the compiler-based technique for strided accesses achieves almost 60% speedup on *art* and about 40% on *em3d*, with an average of 16% in performance speedup. The scheme for linked data structures yields an average of 55%, but it does extremely well on *perim*(a speedup of 5.6x). Without *perim*, the average speedup goes down to just 10%.

We calculated the total energy consumption in the memory system for each prefetching technique based on HSPICE. The results are shown in Figure 2.13. In the figure, we show the energy breakdown for (from bottom to top for each bar) L1 dynamic energy, L1 leakage, L2 dynamic energy, L2 leakage, L1 tag lookups due to prefetching, and prefetch hardware table accesses for hardware prefetching or prefetch instruction overhead for software prefetching.

The results in Figure 2.13 show that the three hardware-based prefetching schemes result in a significant energy consumption overhead, especially in the combined prefetching approach. The average increase for the combined approach is more than 28%, which is mainly due to the prefetch table accesses and the extra L1 tag lookups

due to prefetching. Software prefetching also increases energy consumption for most of the benchmarks, especially in *mcf* and *em3d*. However, compared to the combined hardware prefetching, software prefetching techniques are more energy-efficient for most of the benchmarks.

Considering both performance and energy-efficiency, it seems that there is no single prefetching solution which would yield the best performance and at the same time consume the least energy consumption. Based on our observation, the combined hardware-based technique outperforms others in terms of speedup for most benchmarks although it consumes considerably more energy than the other four techniques. The question is: can we make the combined hardware prefetching more energy-efficient without sacrificing its performance benefits? We will address the issue in the next chapter.

2.6 Chapter Summary

This chapter studies the energy consumption issues related to data prefetching. In deep-submicron process technologies, memory system energy is dominated by the leakage component unless effective leakage reduction techniques are used. As feature sizes continue to decrease, leakage power will constitute an increasing fraction of the total energy consumption, favoring aggressive prefetching techniques. However, with successful leakage control, the problem shifts back to tuning the level of prefetch aggressiveness; otherwise the energy cost of prefetching will be dominated by the overhead from the prefetching hardware energy consumption and from extra L1 lookups when prefetching requests resolve at L1 Cache.

Clearly, for low-power processors, designing the appropriate prefetching technique with good speedup and less energy overhead will be very important. New power-aware prefetching techniques are needed to reduce the energy overhead without decreasing

the performance benefits of data prefetching. We will focus on how to achieve this goal in the rest of this dissertation.

CHAPTER 3

ENERGY-AWARE PREFETCH FILTERING

This chapter proposes several filtering techniques to make hardware-based data prefetching more energy-efficient. Our proposed techniques include three compiler-based approaches which make the prefetch predictor more power efficient. The compiler identifies the pattern of memory accesses in order to selectively apply different prefetching schemes depending on predicted access patterns and to filter out unnecessary prefetches. We also propose a hardware-based filtering technique to further reduce the energy overhead due to prefetching in the L1 cache. Our experiments show that the proposed techniques reduce the prefetching-related energy overhead by close to 40% without reducing its performance benefits.

3.1 Introduction

Our experiments in the last chapter on five hardware-based data prefetching techniques show that while aggressive prefetching techniques often help to improve performance, in most of the applications, they increase memory system energy consumption by as much as 30%. In many systems [38, 67], this is equivalent to more than 15% increase in chip-wide energy consumption.

Aggressive hardware prefetching is beneficial in many applications as it helps to hide memory-system related performance costs. By doing that, however, it often significantly increases energy consumption in the memory system. The memory system consumes a large fraction of the total chip-energy and it is therefore a key area targeted for energy optimizations. Our experimental results in the previous chapter

show that most of the energy degradation is due to the prefetch-hardware related energy costs and unnecessary L1 data-cache lookups related to prefetches that hit in the L1 cache.

This chapter proposes several power-aware prefetch filtering techniques for hardware data prefetching to reduce the energy overheads stated above. The techniques include:

- A *compiler-based selective filtering (CBSF)* approach which reduces the number of accesses to the prefetch hardware by only searching the prefetch hardware tables for selected memory accesses that are identified by the compiler;
- A *compiler-assisted adaptive prefetching(CAAP)* mechanism, which utilizes compiler information to selectively apply different prefetching schemes depending on predicted memory access patterns;
- A compiler-driven filtering technique using a *runtime stride counter(SC)* designed to reduce prefetching energy consumption on memory access patterns with very small strides; and
- A hardware-based filtering technique using a *prefetch filter buffer (PFB)* applied to further reduce the L1 cache related energy overhead due to prefetching.

Compiler-based prefetch filtering requires static compiler analysis to provide program information in order to apply respective filtering technique. The compiler passes are implemented by extending the SUIF infrastructure [99]. To apply the filtering techniques on pointer-intensive applications, a pointer analysis pass called runtime-biased pointer analysis [40] is first applied to generate the location-set information of all the pointers. The necessary information for filtering is then acquired based on array structure information collected through regular SUIF functions and the location-set information for pointers.

The SimpleScalar [18] simulation tool has been modified to implement the different prefetching techniques and collect statistics on performance as well as switching activity in the memory system. To estimate power consumption in the memory system, we use state-of-the-art low-power cache circuits and simulate them using HSPICE. Our experiments show that the proposed techniques successfully reduce the prefetching-related energy overheads by 40% on average, without reducing the performance benefits of prefetching.

The rest of this chapter is organized as follows. An overview of the energy-aware prefetching solutions are presented in Section 3.2. Section 3.2 provides an introduction to the key compiler analysis passes used in our application. Detailed descriptions on compiler-based filtering are presented in Section 3.4 and hardware filtering is presented in Section 3.5. We summarize this chapter with Section 3.6.

3.2 Filtering Overview

Our experimental results show that most of the energy overhead due to prefetching comes from two areas. The major part is from the prefetching prediction phase: when we search/update the prefetch history table to find potential prefetching opportunities; Another significant part of the energy overhead comes from the extra L1 tag-lookups. This is because many unnecessary prefetches are issued by the prefetch engine.

Fig. 3.1 shows the modified combined prefetching architecture including four energy-saving components. The first three techniques are compiler-based approaches used to reduce prefetch-table related costs and some extra L1 tag lookups due to prefetching. The last one is a hardware-based approach designed to reduce the extra L1 tag lookups. The techniques proposed, as numbered in Fig. 3.1, are:

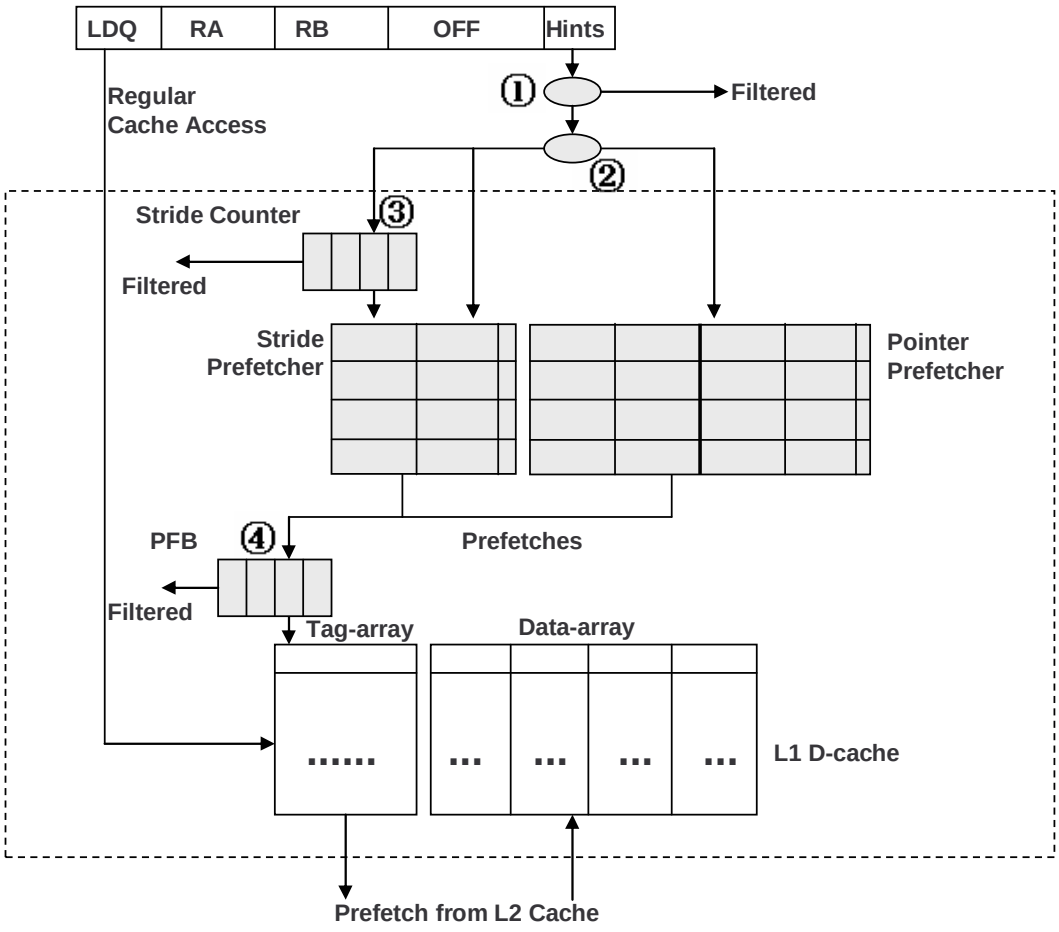


Figure 3.1. Power-aware prefetching architecture for general-purpose programs

1. A *compiler-based selective filtering (CBSF)* approach which reduces the number of accesses to the prefetch hardware by only searching the prefetch hardware tables for selected memory accesses that are identified by the compiler;

2. A *compiler-assisted adaptive prefetching(CAAP)* mechanism, which utilizes compiler information to selectively apply different prefetching schemes depending on predicted memory access patterns;

3. A compiler-driven filtering technique using a *runtime stride counter(SC)* designed to reduce prefetching energy consumption on memory access patterns with very small strides; and

4. A hardware-based filtering technique using a *prefetch filter buffer (PFB)* applied to further reduce the L1 cache related energy overhead due to prefetching.

The compiler-based approaches help make the prefetch predictor more selective based on program information extracted. With the help of the compiler hints, the energy-aware prefetch engine performs much fewer searches in the prefetch hardware tables and issues fewer prefetches, which results in less energy overhead being consumed in L1 cache tag-lookups.

Fig. 3.2 shows the compiler passes in our approach. Prefetch analysis is the process where we generate the prefetching hints, including whether or not we will do prefetching, which prefetcher to choose, and the stride information. A speculative pointer and stride analysis approach [40] is applied to help analyze the programs and generate the information we need for prefetch analysis. Compiler-assisted techniques require the modification of the instruction set architecture to encode the prefetch hints generated by the compiler analysis. These hints could be accommodated by reducing the number of offset bits. We will discuss how to perform the analysis for each of the techniques in detail later.

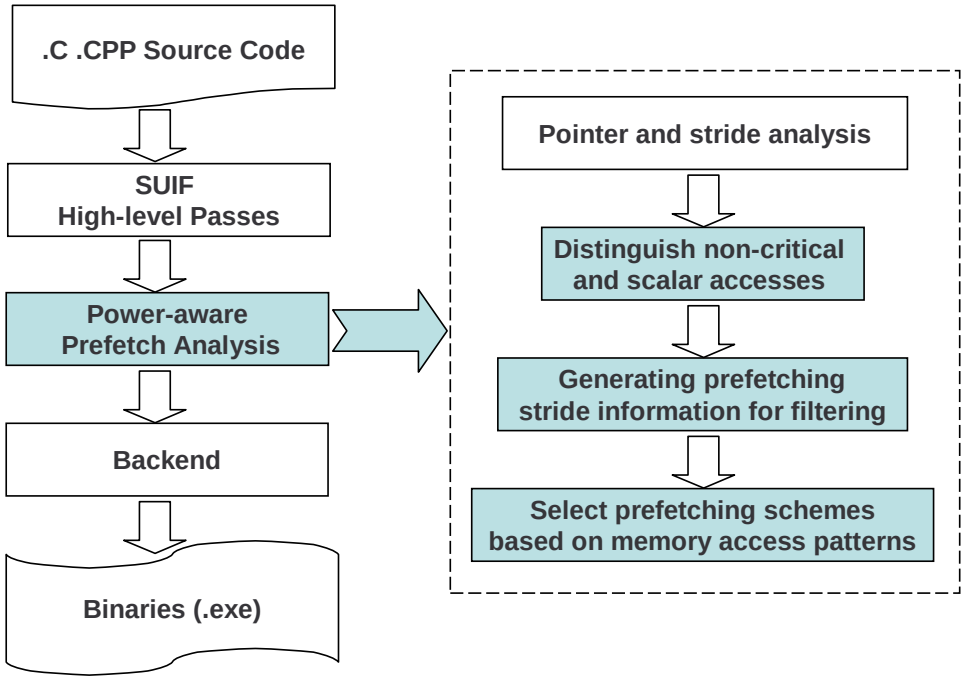


Figure 3.2. Compiler analysis used for power-aware prefetching

In addition, our hardware-based filtering technique utilizes the temporal and spatial locality of prefetching requests to filter out the requests trying to prefetch the same cache line as prefetched recently. The technique is based on a small hardware buffer called the Prefetch Filtering Buffer (PFB).

Next, we will first present the key compiler analysis steps used in the compiler-based filtering before we present the detailed filtering techniques.

3.3 Runtime-Biased Pointer Reuse Analysis

Many researchers have focused on program locality/reuse analysis for array-based memory accesses [68, 102, 103]. In general, array accesses are more regular than pointer-based memory accesses because arrays are normally accessed sequentially while pointers typically have more complicated behavior. As a result, array based accesses are also relatively easy to deal with as type information is available to guide the analysis.

Table 3.1. Pointer analysis results using a context- and flow-sensitive state-of-the-art pointer analysis tool.

Benchmark	Analysis result
em3d, allroots, backprop, eks, bintr	succeeded
mcf, mst, vor	failed, because of pointer cast conversion from integer
art, equake, parser, vpr, bh, bisort, perimeter, power, treeadd, tsp, vor, health, bisect, ks, main, trie, qbsort, football	failed, because of undefined global pointers from libraries

Intensive use of pointers makes however program analysis difficult since a pointer may point to different locations during execution time; the set of all locations a pointer can access at runtime is typically referred to as the *location set*. This difficulty is further accentuated in the context of large and/or complex programs. Precise dataflow-based implementations of pointer analysis often cannot complete the analysis when used for large programs or when special constructs such as pointer based calls, recursion, or library calls are found in the program. Our experiments with a recently developed flow- and context-sensitive state-of-the-art analysis [83] show that it would not complete for several of our tested programs(see Table 3.1). This, however, is not a limitation of a particular implementation: it is, rather, a fundamental limitation of any conservative program analysis technique. Clearly, the explanation is that conservative analysis could not be completed when provable correctness could not be guaranteed at compile time.

Our objective is to develop new techniques to capture pointer behavior in complex applications with no restrictions, applicable in optimizations where absolute correctness of the information extracted is not necessary for guaranteeing correctness of execution. *The idea is to determine pointer behavior by capturing the frequent locations for each pointer rather than all the locations as conservative analysis would do.* Predicted pointer reuse information is therefore runtime biased and

speculative in the sense that the possible targets for each pointer access are statically predicted/speculated based on the likelihood of their occurrence at runtime.

A key difference between traditional pointer analysis and our RB pointer analysis is that the targets captured are not guaranteed at compile time. While the analysis can give incorrect targets or omit targets, it has the advantage that it always completes and often omits targets that are anyway seldom used at runtime. These targets would have little effect anyway when used to control energy consumption in energy-aware techniques. Besides memory accesses with good reuse/locality, the technique also identifies irregular accesses that typically result in energy and performance penalties no matter how they are managed in an energy-aware context.

The approach presented here is applicable in all architecture optimizations that use some kind of compiler-exposed speculation hardware and when absolute correctness of static information leveraged is not necessary. Beside compiler managed prefetching, the potential techniques include for example compiler managed energy-aware memory systems, and speculative parallelization and synchronization - these applications by their design benefit from precise memory behavior information but could tolerate occasional incorrect static control information.

We will first give a brief overview of traditional pointer analysis mechanisms in the next section, then present the key steps in the runtime-biased pointer reuse analysis. More detailed information on the runtime-biased pointer analysis can be found in our previous work [40].

3.3.1 Traditional Pointer Analysis

Pointer analysis attempts to statically determine the possible values a pointer will take at runtime. This is a relatively mature field [26, 33, 56, 82, 83, 86, 93, 100], and the existing techniques can be classified by two major properties: flow-sensitivity and context-sensitivity.

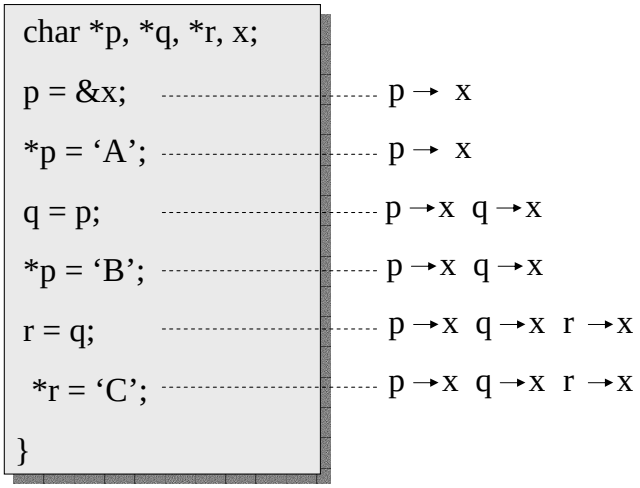


Figure 3.3. A simple points-to graph.

Flow-sensitive analysis makes use of a procedure's control-flow information, producing a solution for each program point [26, 33, 56, 82, 100]. Flow-insensitive analysis, on the other hand, produces one solution for either the whole program or for each individual procedure [7, 86, 93]. Because of this, flow-insensitive analysis techniques are typically more efficient than flow-sensitive analysis techniques, however they are also less precise [94].

In context-sensitive analysis, the calling context is considered when analyzing a procedure. Thus, a result is generated for each different calling context of a procedure. In context-insensitive analysis, only one result is generated for each procedure. This result is typically found by merging information from different call sites. Like flow-sensitive analysis, context-sensitive analysis is generally less efficient than context-insensitive analysis but is considered to be more accurate even though, in this case, it is not clear whether this belief is true or not [82, 100].

During pointer analysis, the *location sets*, or the sets of locations each pointer may point to, are represented as a *points-to graph* (PTG). Nodes in a PTG correspond to program variables and edges are used to represent each possible points-to relation. Figure 3.3 shows a simple program segment and a points-to graph that describes it.

Without points-to analysis, we would not be able to draw the conclusion that pointers p, q, and r are pointing to the same location at compile-time. By using the points-to information generated through pointer analysis, it is clear that these pointers will all point to the same location in memory and thus potentially the same cache line.

3.3.2 Overview - RB Compiler Analyses

The runtime biased (RB) compiler analyses can be separated into a series of three steps: RB pointer analysis, RB distance analysis, and RB reuse analysis. Instead of basing our presentation on a formal description such as dataflow equations, we focus on some of the key insights and steps that make our analysis unique.

RB Pointer Analysis is first applied in order to gather basic pointer information needed to predict pointer access patterns. A flow-sensitive dataflow scheme is used in our implementation. Flow-sensitive analysis maintains high precision (i.e., the location set of each pointer access is determined in a flow-sensitive manner even if based on the same variable). Our analysis is guided by re-evaluating, at each pointer dereference point, the (likely) runtime frequency of each location a pointer can point to. For example, possible locations that are from definitions in outer loop-nests are marked or not included when the pointer is dereferenced in inner loops and if at least one new location has been defined in the inner loop. Conventional analysis would not distinguish between these locations.

RB Distance Analysis gathers stride information for pointers changing across loop iterations. The stride information is used to predict pointer-based memory access patterns, and speculation is performed whenever the stride is not fixed. As strides could change in function of the paths taken in the Control-Flow Graph (CFG) of the loop body, only the most likely strides (based on static branch prediction) are considered.

RB Reuse Analysis attempts to discover those pointer accesses that have cache line reuse property, i.e., refer to the same cache line. Reuse analysis uses the information provided by the previous analyses to decide whether two pointer accesses might refer to the same cache line. Based on the reuse patterns, pointer accesses are partitioned into *reuse equivalence classes (RECs)*. Each REC can be either *regular*, which means that memory accesses belonging to the REC have high cache line reuse, or *irregular*, which means that memory accesses in the REC are not reusing cache lines frequently.

3.3.3 RB Pointer Analysis

There are two ways to give pointer information: (1) through program-point information, and (2) through global information. Figure 3.4 shows a simple C program and illustrates the difference between these representations. Program point information for example would show that at the end of the program segment in Figure 3.4, pointer p points to location set $\{y,z\}$, a more precise information, compared to the global information case where p points to location set $\{x,y,z\}$. Although global information can be extracted with much more efficient analysis algorithms, it gives less precise results. Our analysis is based on program point information that combined with the program CFG forms the point-to graph (PTG) of a program.

Nodes in a PTG correspond to program variables and edges represent points-to relations. A points-to relation connects two variables and means that a pointer variable can take the value of another variable (or location) during execution.

Precise conventional pointer analysis usually requires that the program includes all its source codes, for all the procedures, including static libraries. Otherwise, the analysis cannot be performed. The analysis is often used in program optimizations where conservative assumption must be made - any speculation could result in incorrect execution.

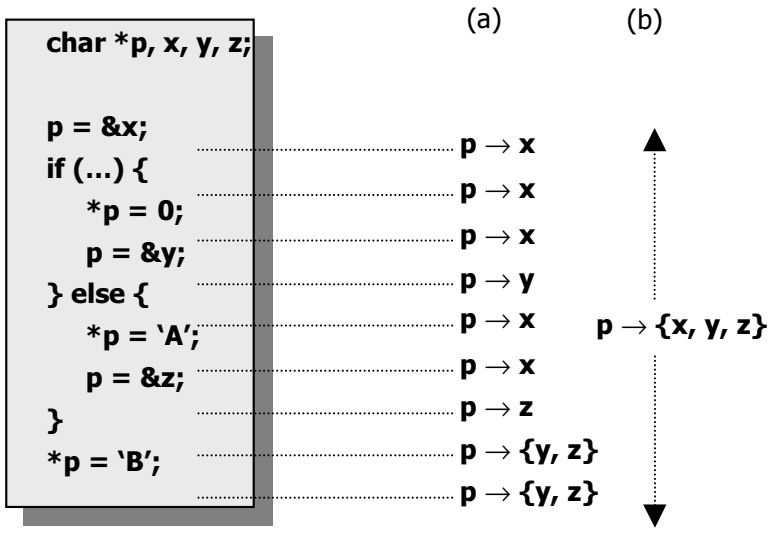


Figure 3.4. Pointer information representations: (a) program-point representation, (b) through global information.

In contrast, our approach does not require the same type of strict correctness. If the behavior of a specific pointer cannot be inferred precisely, we can speculate or just ignore its effect. For example, if a points-to relation (or location) cannot be inferred statically, we speculatively consider only the other locations gathered in the pointer's location set. We mark the location as undefined. When assigning location sets for the same pointer at a later point in the CFG, one could safely ignore/remove the *undefined location* in the set, if the probability of the pointer accessing that location, at the new program point, is low.

For the example in Figure 3.5, after pointer analysis, the pointer p at the position $*p \mathrel{+}= 5$ may point to either a or b. With only this information, we cannot decide exactly where p points during runtime. However, if we consider how frequently each location will be accessed during execution, e.g., with static branch prediction, we can predict that the probability for p to point to a during runtime is only 0.1%. Since p points to b for most of the time, we will conclude that p always points to b by speculation. The idea is that although we may make some incorrect predictions, the

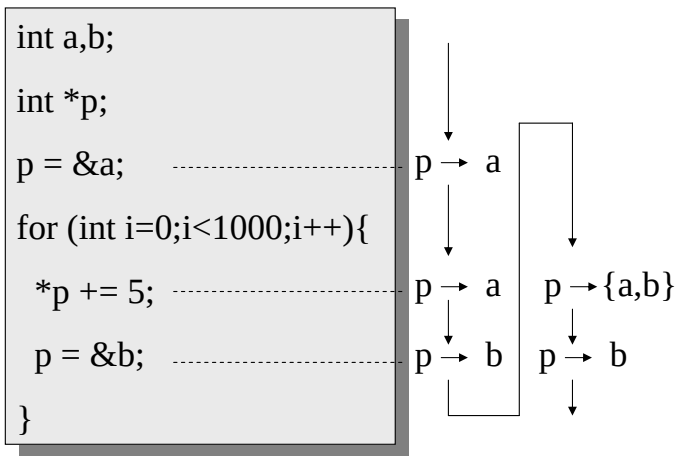


Figure 3.5. Location set optimization example based on static branch prediction. After the first iteration we will conclude that p always points to b by speculation.

penalties caused by these mispredictions are small compared to the energy benefits we can often achieve.

The main steps of our RB pointer analysis algorithm are as follows: (1) build a control-flow graph (CFG) of the computation, (2) analyze each basic block in the CFG gradually building a PTG, (3) at the beginning of each basic block merge location set information from previous basic blocks, (4) mark locations in the location sets that are unlikely to occur at runtime, at the current program point, as less frequent, (5) mark undefined locations or point-to relations; (6) repeat steps 2-5 until the PTG graph does not change (i.e., full convergence is reached) or until the allowed number of iterations are reached.

Library calls that may modify pointer values and for which source codes are not available are currently speculatively ignored in our implementation. If a pointer is passed in as an argument, its location set after the call-point in the caller procedure will be marked as speculative, signaling that the location set of the pointer might be incomplete after the call. In none of the programs we have analyzed we have found library-modified pointer behavior to be a considerable factor.

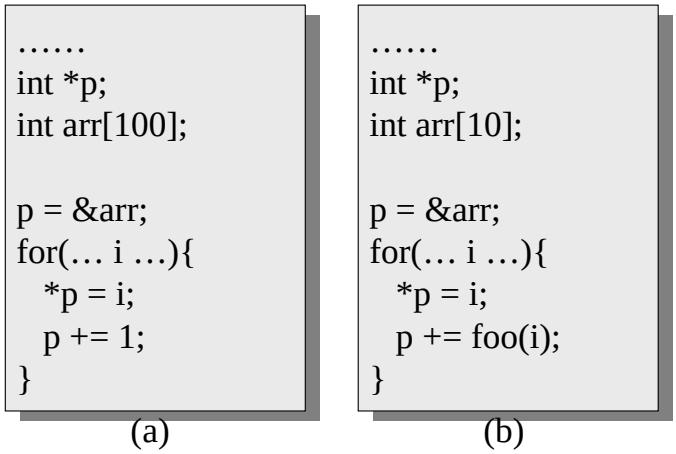


Figure 3.6. Distance analysis examples: (a) static stride (b) variable stride

3.3.4 RB Distance Analysis

Focusing on loop-based pointers, we introduce distance analysis which helps us to capture the changing pattern if a pointer variable in the loop is modified for each loop iteration. In the example shown in Figure 3.6(a), the value of pointer p changes after each iteration. In general, there are two ways to deal with this situation if implemented as part of pointer analysis. Each element in the array structure could be treated as a different location, or, another approach would be to treat the whole array *arr* as a single location. The former is too complicated for compiler analysis while the latter is not precise enough.

In our approach, as shown in Figure 3.6(a), we first find the initial location for p. Then, when we find out that p is changing for each iteration, we calculate the distance (stride) between the current location and the location after modification. If the distance is a constant, we will use both the initial location and distance to describe the behavior of the pointer.

Extracting stride information is not always easy. In Figure 3.6(a), we could easily calculate that the stride for pointer p is 4 bytes. However, for the example in Figure 3.6(b), the stride for pointer p is variable since we do not know what

value procedure *foo()* will return. In this case, we can use speculation based on static information related to the location set to estimate the stride. For example, the information we do know is (1) p points to array *arr* and (2) the size of array *arr* is small. Based on this information, we can speculate that the stride of p is small although we do not know the exact number.

Another example of stride prediction, as also mentioned earlier, is ignoring strides that are less likely to occur at runtime based on static branch prediction. Clearly, depending on which path is executed at runtime, the stride of a pointer might change across loop iterations, as not all the possible paths leading to that pointer access are equally likely to occur.

3.3.5 RB Reuse Analysis

Reuse analysis is used to decide which memory accesses are likely to access the same cache line, which is the information needed for our compiler-managed data prefetching. Reuse analysis for array-based accesses has been studied and used in [68, 102, 103] for data prefetching. For pointer-intensive programs, we use a similar classification scheme, but we redefine it specifically in the context of pointer-based accesses. Four types of reuse patterns are identified in our analysis:

1. *Temporal Reuse*: This is the case when a pointer is not changing during loop iterations. This is the simplest case for loop-based accesses, as shown in Figure 3.7(a). In this example, the pointer access *p->result* is never changed during different loop iterations, so the cache line it refers to will be reused during each loop iteration.

2. *Self-Spatial Reuse*: If a pointer is changing using a constant stride and the stride is small enough, two or more consecutive accesses will refer to the same cache line. As shown in Figure 3.7(b), the pointer p is changing after each iteration,

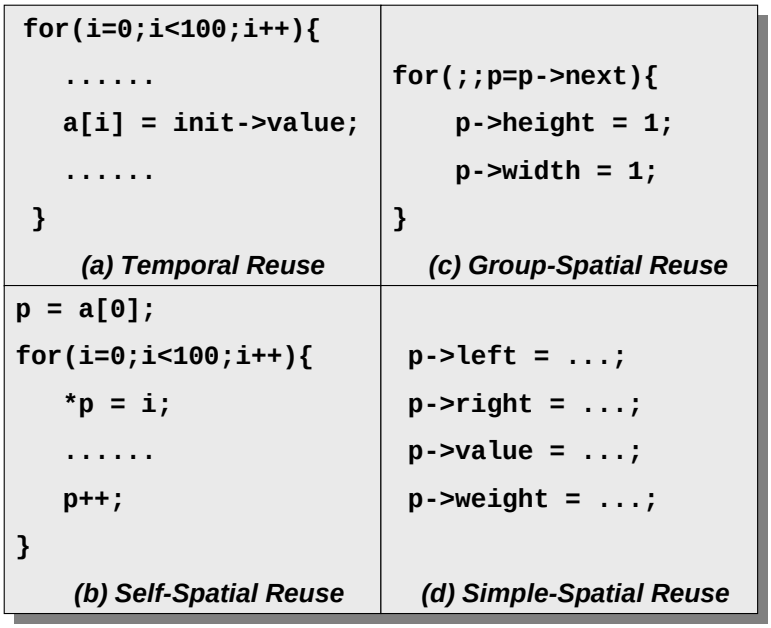


Figure 3.7. Reuse Classification

however it is changing regularly. Although p is pointing to a different address each time, there is still great possibility for it to access the same cache line.

3. *Group-Spatial Reuse*: A group of pointers can share the same cache line during each loop iteration even when they do not exhibit self-spatial reuse. In the example of Figure 3.7(c), p->*height* and p->*width* are always pointing to the same cache line for each loop iteration, although the cache line may be different for different iterations.

4. *Simple-Spatial Reuse*: This exists between two pointers that refer to the same cache line but do not belong to any loop. In Figure 3.7(d), we can see that all pointer accesses will probably access the same cache line, so there is a reuse relationship among them. Simple-spatial reuse is added as a new reuse category because we found that this situation is important for pointer-based programs although it is not as important for array-based programs. The reason is that array structures are typically accessed using loops, while pointer-based data

structures are often accessed using recursive functions. When a small recursive function is called frequently, the simple-spatial reuse cases within the function will become significant.

Based on the reuse classification above and strides information gathered by distance analysis, pointer-based memory accesses are partitioned into different *reuse equivalence classes (RECs)*. Each REC contains a group of memory accesses with reuse equivalence relations among them. A *reuse equivalence relation* between two memory accesses exists when they have a good chance to access the same cache line during runtime.

Pointers with different properties will be treated differently during the reuse equivalence classification process. First, loop-based accesses are scanned to identify temporal and self-spatial reuses using the following criteria:

1. Static loop-based accesses which do not change will be regarded as temporal reuse;

2. Loop-based accesses with static stride will be categorized as self-spatial reuse if the stride is small enough (for example, less than half of the cache line size);

3. Loop-based accesses with unknown (dynamic) strides will also be treated as self-spatial reuse if their stride can be speculated as small from distance analysis.

Each of these temporal reuse or self-spatial reuse accesses will be assigned to an REC containing only the memory access itself.

Next, non-loop accesses and loop-based accesses that do not have temporal or self-spatial reuse properties will be scanned by a compiler pass for group-spatial and simple-spatial reuses. The algorithm used here is an intra-procedural analysis using a recursive function which is shown in Figure 3.8. Before we analyze each procedure, all RECs are initialized to empty. A reuse address list is also maintained for each REC

that contains the most recent memory access assigned to the REC. The analysis is performed recursively for each basic block in the Control-Flow Graph (CFG). During the analysis, the address of every memory access is compared to the addresses in the address list in order to find the closest REC. If its distance to the closest REC is smaller than half of the cache line size, it will be assigned to the same REC; otherwise the memory access will be assigned to the next available REC. The process continues recursively with the following basic blocks until all the basic blocks in the procedure are analyzed.

Among all the RECs identified during this process, an REC containing loop-based memory accesses will be identified as group-spatial, while an REC containing memory accesses in recursive functions will be identified as simple-spatial. All the memory accesses in the remaining RECs will be regarded as irregular accesses.

In group-spatial and simple-spatial reuse cases, the number of accesses contained in an REC reflects the reuse possibility (accessing the same cache line) of this particular class. In our applications, the size of an REC should be at least three or four to allow frequent cache line reuse and compensate the penalties that might occur because of the cold-start miss for each REC. If the size of the REC is too small to allow frequent cache line use to occur, all the memory accesses belonging to this REC will also be considered as irregular accesses.

After the reuse analysis pass, each memory access has been assigned to an REC. All the RECs with temporal and self-spatial reuses will be regarded as *regular* accesses, as well as RECs with group-spatial and simple-spatial reuses if their sizes are large enough. The remaining RECs will be treated as *irregular* as we mentioned above.

Each REC can be either *regular*, which means that memory accesses belonging to the REC have high cache line reuse, or *irregular*, which means that memory accesses in the REC are not reusing cache lines frequently. The regular memory accesses

```
/* For each procedure, start with the first basic block */
FOR each procedure DO
  B = first basic block;
  FOR REC index i: 1 to MAX DO
    REC[i] = {}; /* initialize each REC */
    access[i] = NULL; /* most recent access for REC[i] */
  END FOR
  call Analyze_Block B;
END FOR

/* recursive function to analyze a block X */
PROCEDURE Analyze_Block X
  FOR each access A in X DO
    distance = INFINITE;
    /* find the closest REC for access A */
    FOR all non-empty REC index: i from 1 to n DO
      IF (distance_between(A, access[i]) < distance) THEN
        closest = i and update distance;
      END IF
    END FOR
    /* if the smallest distance is close enough, assign A to the REC */
    IF distance <= CacheLineSize/2 THEN
      add A to REC[closest];
      access[closest] = A;
    ELSE
    /* otherwise assign A to the next available REC */
      assign A to the next available REC;
      access[next] = A;
    END IF
  END FOR

  workList = successors of X in CFG;

  WHILE !empty(workList) DO
    B = next basic block in workList;
    IF B.analyzed THEN
      continue;
    END IF
    /* Traverse through the CFG by making recursive calls */
    call Analyze_Block B;
    B.analyzed = true;
  END WHILE
END PROCEDURE
```

Figure 3.8. Compiler analysis for group-spatial and simple-spatial reuses

have good cache line reuse possibility, while the remaining irregular accesses are not reusing cache lines frequently. These information will be very important while making prefetching decisions (as used in [68]) as well as energy-performance tradeoffs (as used in this dissertation).

3.4 Compiler-Assisted Filtering

3.4.1 Compiler-Based Selective Filtering (CBSF)

One of our observations is that not all load instructions are useful for prefetching. Some instructions, such as scalar memory accesses, have no access patterns and cannot anyway trigger useful prefetches when fed into the prefetcher.

We use the compiler to distinguish memory accesses useful for prefetching from those which my have no benefit. Only those useful load instructions, selected by the compiler, are fed into the prefetcher. Instructions identified with "no prefetching potential" will not be added to the prefetch history table. Thus, these instructions will not contribute to the energy consumption overhead.

The compiler identifies the following memory accesses as having "no prefetching potential":

- *Non-critical accesses*: Memory accesses within a loop or a recursive function are regarded as critical accesses. Because prefetching schemes are anyway designed to capture the memory access patterns in critical program phases, we can safely filter out the non-critical accesses before they reach the prefetcher.

- *Scalar accesses*: Scalar accesses do not have any pattern and will not contribute to the prefetcher if fed into the prefetcher. Only memory accesses to array structures and linked data structures will be sent to the prefetcher to make prefetching decisions.

The instructions selected by the compiler are annotated with "no prefetching potential" and are filtered out before they are fed into the prefetcher. This optimization could eliminate on average as much as 8% of all the prefetch table accesses, as we will show later.

3.4.2 Compiler-Assisted Adaptive Prefetching (CAAP)

Another compiler approach focuses on how to help the prefetch predictor choose one of the prefetching schemes in the combined prefetching approach.

One important aspect of the combined approach is that it uses two techniques independently and prefetches based on the memory access patterns for all memory accesses. As we know, stride prefetching works better on array-based accesses and dependence-based prefetching is more appropriate for pointer-based structures. One obvious approach is therefore to distinguish these two types of accesses.

Distinguishing between pointers and non-pointer accesses is difficult during execution time. However, we can distinguish them easily during compilation. Array accesses and pointer accesses are annotated using hints written into the instructions. During runtime, the prefetch engine can identify the hints and apply different prefetching mechanisms.

We have found that simply splitting the array and pointer structures is not very effective and affects the performance speedup (which is the primary goal of prefetching techniques). Instead, we use the following heuristic to decide whether we should use stride prefetching or pointer prefetching:

- Memory accesses to an array which does not belong to any larger structure (e.g., fields in a C struct) are only fed into the stride prefetcher;
- Memory accesses to an array which belongs to a larger structure are fed into both stride and pointer prefetchers;

- Memory accesses to a linked data structure with no arrays are only fed into the pointer prefetcher;
- Memory accesses to a linked data structure that contains arrays are fed into both prefetchers.

The above heuristic is able to preserve the performance speedup benefits of the aggressive prefetching scheme. We can filter out up to 20% of all the prefetch-table accesses and up to 10% of the extra L1 tag lookups due to prefetching, by applying this technique.

3.4.3 Compiler-Hinted Filtering Using a Runtime Stride Counter (SC)

Another part of prefetching energy overhead comes from memory accesses with small strides. Accesses with very small strides (compared to the cache line size of 32 bytes we use) could result in frequent accesses to the prefetch table and issuing more prefetch requests than needed. For example, if we have an iteration on an array with a stride of 4 bytes, we will access the hardware table at least 8 times before we reach the point where we can issue a useful prefetch to get a new cache line. The overhead not only comes from the extra prefetch table accesses; 8 different prefetch requests are also issued to prefetch the same cache line during the 8 iterations.

Software prefetching would be able to avoid the penalty by doing loop unrolling. In our approach, we use hardware to accomplish loop unrolling with assistance from the compiler. The compiler predicts as many strides as possible based on static information. Stride analysis is applied not only for array-based memory accesses, but we also predict strides for pointer accesses with the help of pointer analysis. Detailed information on how to do the pointer and stride analysis could be found in our previous work [40].

Strides predicted as larger than half of the cache line size (16 bytes) will be considered as large enough since they will be able to reach a different cache line after

each iteration. Strides smaller than the half of the cache line size will be recorded and passed to the hardware. This is a very small 8-entry buffer used to record the most recently used instructions with small strides. Each entry contains the program counter (PC) of the particular instruction and a stride counter. The counter is used to count how many times the instruction occurs after it was last fed into the prefetcher. The counter is initially set to a maximum value (decided by cache_line_size/stride) and is then decremented each time the instruction is executed. The instruction is only fed into the prefetcher when its counter is decreased to zero; then, the counter will be reset to the maximum value.

For example, if we have an array access (in a loop) with a stride of 4 bytes, the counter will be set to 8 initially. Thus, during eight occurrences of this load instruction, only once it is sent to the prefetcher.

This technique reduces 5% of all the prefetch table accesses as well as 10% of the extra L1 cache tag lookups, while resulting in less than 0.3% performance degradation.

3.5 Hardware Prefetch Filtering Using PFB

To further reduce the L1 tag-lookup related energy consumption, we add a hardware-based prefetch filtering technique. Our approach is based on a very small hardware buffer called the Prefetch Filtering Buffer(PFB).

When a prefetch engine predicts a prefetching address, it does not prefetch the data from that address immediately from the lower-level memory system (e.g., L2 Cache). Typically, tag lookups on L1 tag-arrays are performed. If the data to be prefetched already exists in the L1 Cache, the prefetch request from the prefetch engine is dropped. A cache tag-lookup costs much less energy compared to a full read/write access to the low-level memory system (e.g., the L2 cache). However, associative tag-lookups are still energy expensive.

To reduce the number of L1 tag-checks due to prefetching, we add a PFB to remember the most recently prefetched cache tags. We check the prefetching address against the PFB when a prefetching request is issued by the prefetch engine. If the address is found in the PFB, the prefetching request is dropped and we assume that the data is already in the L1 cache. When the data is not found in the PFB, we perform normal tag lookup and proceed according to the lookup results. The LRU replacement algorithm is used when the PFB is full. The prefetch filtering scheme using the PFB is shown in Fig. 3.1.

A smaller PFB costs less energy per access, but can only filter out a smaller number of useless prefetches. A larger PFB can filter out more useless prefetches, but each access to the PFB costs more energy. To find out the optimal size of the PFB, we simulated a set of benchmarks with PFB sizes of 1 to 16. We will show in Chapter 5 that an 8-entry PFB is large enough to accomplish the prefetch filtering task with very small performance overhead.

PFBs are not always correct in predicting whether the data is still in L1 since the data might have been replaced although its address is still present in the PFB. We call this case a PFB misprediction. High PFB mispredictions would result in performance loss because useful prefetches are dropped. Fortunately, as we will show later, the PFB misprediction rate is very low (close to 0).

3.6 Chapter Summary

This chapter explores the energy-efficiency aspects of data-prefetching techniques and proposes several filtering techniques to make prefetching energy-aware. Our proposed techniques include three compiler-based approaches which help to make the prefetch predictor more selective and filter out unnecessary prefetches based on static program information. We also propose a hardware based filtering technique to further reduce the energy overheads due to prefetching in the L1 cache.

A new compiler analysis called runtime-biased pointer reuse analysis is applied to support the compiler-based filtering mechanisms. A key difference between traditional pointer analysis and our runtime-biased pointer analysis is that the targets captured are not guaranteed at compile time. While the analysis can give incorrect targets or omit targets, it has the advantage that it always completes and often omits targets that are anyway seldom used at runtime. These targets would have little effect anyway when used to control energy consumption in energy-aware techniques. Besides memory accesses with good reuse/locality, the technique also identifies irregular accesses that typically result in energy and performance penalties no matter how they are managed in an energy-aware context.

The approach presented here is applicable in all architecture optimizations that use some kind of compiler-exposed speculation hardware and when absolute correctness of static information leveraged is not necessary. Beside compiler managed prefetching, the potential techniques include for example compiler managed energy-aware memory systems, and speculative parallelization and synchronization - these applications by their design benefit from precise memory behavior information but could tolerate occasional incorrect static control information.

We will first give a brief overview of traditional pointer analysis mechanisms in the next section, then present the key steps in the runtime-biased pointer reuse analysis. More detailed information on the runtime-biased pointer analysis can be found in our previous work [40].

CHAPTER 4

LOCATION-SET DRIVEN DATA PREFETCHING

In this chapter, we present location-set driven prefetching using PARE, a Power-Aware pRefetching Engine that uses a newly designed indexed hardware history table. Compared to the conventional single table design, the new prefetching table consumes 7-11X less power per access. With the help of compiler-based location-set analysis, we show that the proposed PARE design improves energy consumption by as much as 40% in the data memory systems in 70-nm processor designs.

4.1 Introduction

The filtering techniques presented in the previous section are capable of eliminating a significant portion of unnecessary or useless prefetching attempts. However, we have found that the energy overhead of prefetching is still pretty high, mainly because that it consumes significant power accessing the hardware table used for combined data prefetching.

As presented in previous chapters, hardware prefetching requires the help of a history table to record recent memory access instructions and set up relationships between them in order to make prefetching decisions and calculate prefetching addresses. The history tables are usually pretty large (normally 64-128 entries) [12, 80]. When implemented as a fully-associative CAM table, the energy cost of each table lookup or update operation could cost the amount of energy which is comparable to a read operation of a low-power cache. To make accurate prefetching decisions, the history table is accessed very frequently to update the recent information on

all relevant load instructions, which makes this part of the energy overhead very significant.

In this chapter, we introduce a new data prefetching scheme: *location-set driven data prefetching*, which uses *PARE - a new Power-Aware pRefetching Engine* with a novel design of an indexed hardware history table. Instead of reducing the number of accesses to the prefetch hardware as proposed in the previous chapter, the proposed scheme is focused on improving the power-efficiency of the prefetch hardware itself. We present the detailed design of PARE and compare its power dissipation with the conventional fully-associative table design. We show that with the help of compile-time location-set analysis [83], we can divide the memory accesses into different relationship groups, with each group consisting of memory accesses visiting only closely related location-sets. The compiler generated group numbers allow us to use the indexed history table in PARE.

In the proposed prefetch history table, we divide the total entries into multiple (e.g., 16 or more) smaller tables. Each memory access will be directed to one of the tables upon entering the prefetching engine according to their group numbers provided by the compiler. The prefetching engine will update the information within the group and will make prefetching decisions solely based on the information within this group. The compile-time location-set analysis is utilized to ensure that no information will be lost due to the partitioning of memory accesses. We can reduce the power consumption of each access to the prefetching tables by 7-11X with the proposed technique based on our HSPICE simulation.

To estimate power consumption in the memory system, we use state-of-the-art low-power cache circuits and simulate them using HSPICE. The SimpleScalar [18] simulation tool has been modified to implement the hardware prefetching technique and collect statistics on performance as well as switching activity in the memory system. The compiler passes are implemented using the SUIF infrastructure [99].

Our experiments show that the proposed technique improves the energy consumption by as much as 40% in the data memory system for a set of general-purpose programs. Our evaluation is based on 70-nm BPTM technology node and accounts for both active and leakage power.

The rest of this chapter is organized as follows. The PARE power-aware prefetching engine is presented in Section 4.2. Section 4.3 gives an overview of the location set based group analysis. We conclude with Section 4.4.

4.2 PARE: a Power-Aware Prefetching Engine

As we mentioned earlier, the combined stride and pointer prefetching technique [39] integrates the mechanisms from both stride prefetching [12] and dependence-based prefetching [80].

Stride prefetching captures the static strides between memory accesses (mainly array accesses), and requires a history table to record the address of the instruction (PC), previously accessed address, and the predicted stride. In comparison, the dependence-based prefetching requires two history tables to record the potential candidates of instructions and the correlations which include PC, previously generated addresses, and predicted offset values.

As we will show later, the tables for both techniques could be combined together into a single table, each entry attached with one bit to indicate the prefetching type. We will also use two bits to indicate the prefetching status, which will help us track whether the relationship is steady (status>1) or not. Prefetching requests will be issued only after the relationship is established, i.e., it is *steady*.

Next, we will show the design of our baseline prefetching history table, which is a 64-entry fully-associative table that already uses many circuit-level low-power features. Following that we present the design of the proposed indexed history table

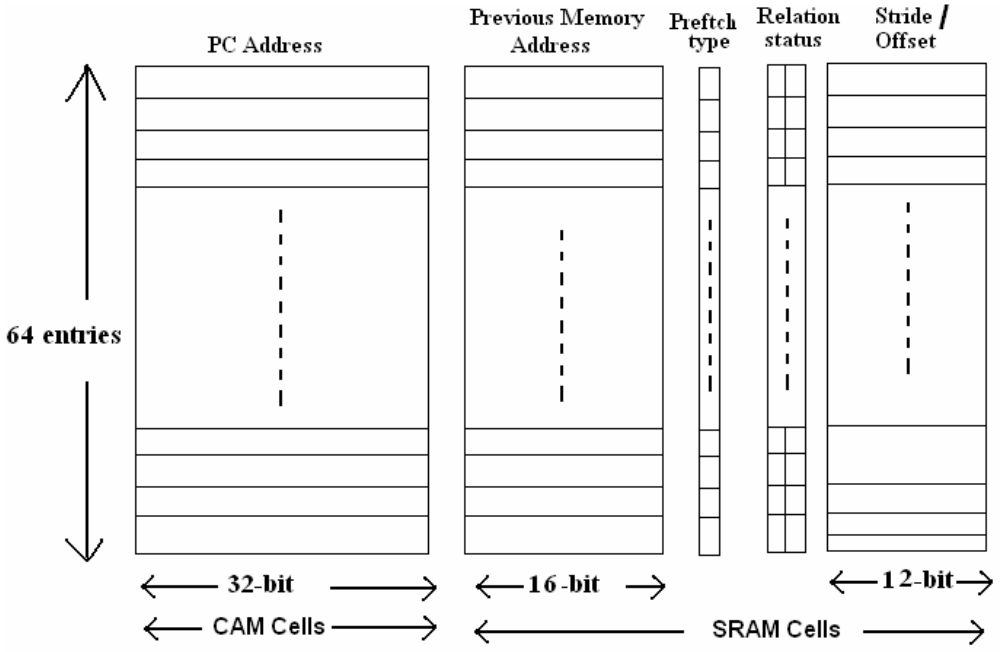


Figure 4.1. The baseline design of hardware prefetch table.

for PARE, and compare the power dissipation, including both dynamic and leakage power, of the two designs.

4.2.1 Baseline History Table Design

The baseline prefetching table design is a 64-entry fully-associative table shown in Figure 4.1. In each table entry, we store a 32-bit program counter (the address of the instruction), the lower 16 bits of the previously used memory address (we do not need to store the whole 32 bits because of the locality property in prefetching). We also use one bit to indicate the prefetching type and two bits for status, as mentioned previously. Finally, each entry also contains the lower 12 bits of the predicted stride/offset value.

In our design, we use Content Addressable Memory (CAM) for the PCs in the table, because CAM provides a fast and power-efficient data search function, accessing data by its content rather than its memory location.

The memory array of CAM cells logically consists of 64 by 32 bits. The rest of the history table is implemented using SRAM arrays. During a search operation, the reference data are driven to and compared in parallel with all locations in the CAM array. Depending on the matching tag, one of the wordlines in the SRAM array is selected and read out.

The prefetching engine will update the table for each load instruction and check whether steady prefetching relationships have been established. If there exists a steady relation, the prefetching address will be calculated according to the relation and data stored in the history table. A prefetching request will be issued in the following cycle.

4.2.2 PARE History Table Design

Each access to the table in Figure 4.1 still consumes significant power because all 64 CAM entries are activated during a search operation. We could reduce the power dissipation in two ways: reducing the size of each entry and partitioning the large table into multiple smaller tables.

First, because of the program locality property, we do not need the whole 32 bits PC to distinguish between different memory access instructions. If we use only the lower 16 bits of the PC, we could reduce roughly half of the power consumed by each CAM access.

Next, we break up the whole history table into 16 smaller tables, each containing only 4 entries, as shown in Figure 4.2. Each memory access will be directed to one of the smaller tables according to their group numbers provided by the compiler when they enter the prefetching engine . The prefetching engine will update the information within the group and will make prefetching decisions solely based on the information within this group. The compile-time location-set analysis is utilized to ensure that no information will be lost due to the partitioning of memory accesses. The group

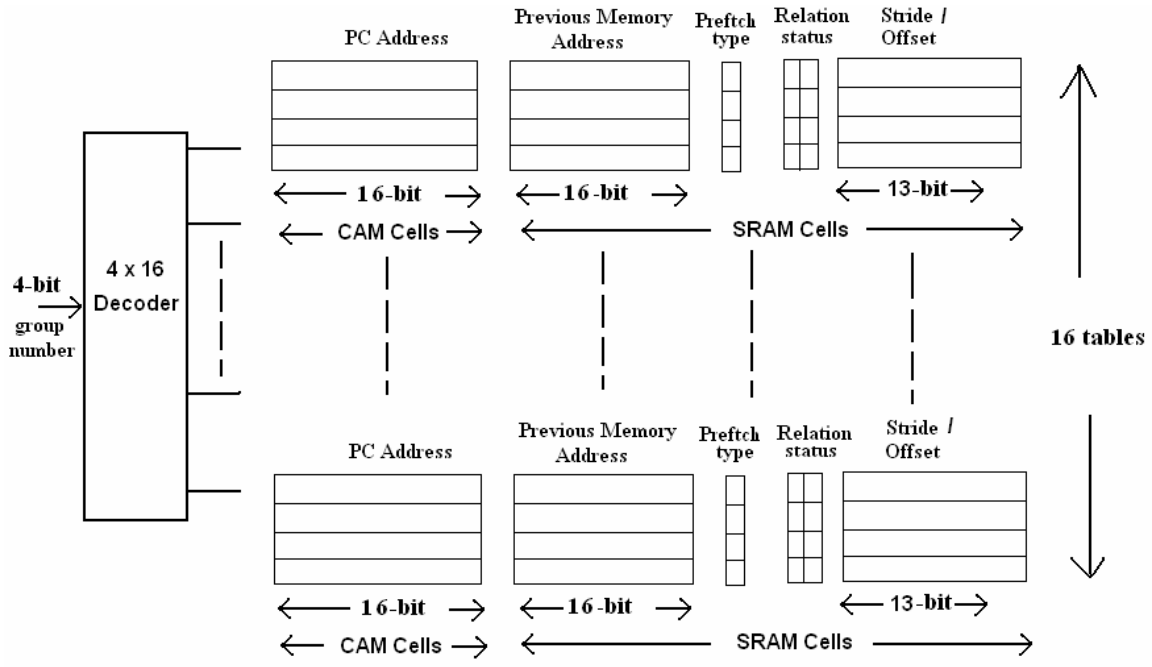


Figure 4.2. The overall organization of our hardware prefetch table.

number can be accommodated in future ISAs that target energy efficiency and can be added easily in VLIW/EPIC type of designs. We also expect that many optimizations that would use compiler hints could be combined to reduce the impact on the ISA. The approach can reduce power significantly even with fewer tables (requiring fewer bits in the ISA) and could also be implemented in current ISAs by using some bits from the offset. Embedded ISAs like ARM that have 4 bits for predication in each instruction could trade off less predication bits (or none) with perhaps more bits used for compiler inserted hints. The compiler analysis will be presented in the next section.

In the PARE history table shown in Figure 4.2, during a search operation, only one of the 16 tables will be activated based on the group number provided by the compiler. We only perform the CAM search within the activated table, which is a fully-associative 4-entry CAM array.

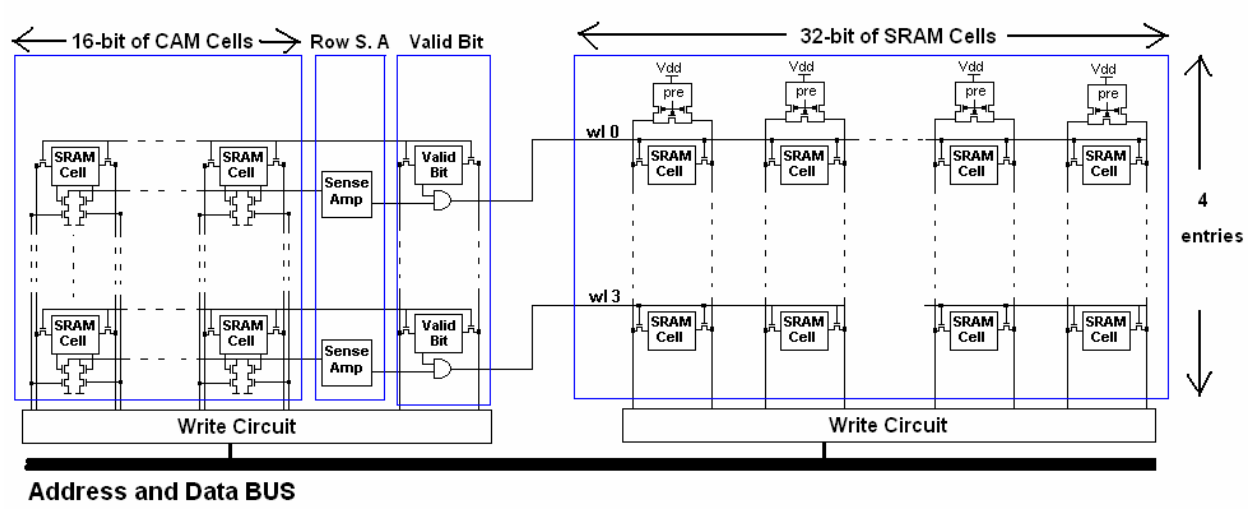


Figure 4.3. The schematic for each small history table.

The schematic of each small table is shown in Figure 4.3. Each small table consists of a 4x16 bits CAM array containing the program counter, a sense amplifier and a valid bit for each CAM row, and the SRAM array on the right which contains the data.

We use one of the most power-efficient CAM cell designs proposed in [111]. The cell uses ten transistors that contain an SRAM cell and a dynamic XOR gate used for comparison. It separates search bitlines from the write bitlines in order to reduce the capacitance switched during a search operation.

For the row sense amplifier, we are using a single-ended alpha latch to sense the match line during the search in the CAM array. The activation timing of the sense amplifier was determined with the case where only one bit in the word has a mismatch state.

Each word has the valid bit which indicates whether the data stored in the word will be used in search operations. A match line and a single ended sense amplifier are associated with each word. A hit/miss signal is also generated: its *high* state indicating a *hit* or *multiple hits* and the *low* state indicating *no hits* or *miss*.

Finally, the SRAM array is the memory block that holds the data. Low-power memory designs typically use a six-transistor (6T) SRAM cell. Writes are performed differentially with full rail voltage swings.

The power dissipation for each successful search is the power consumed in the decoder, CAM search and SRAM read. The power consumed in a CAM search includes the power in the match lines and search lines, the sense amplifiers and the valid bits.

The new hardware prefetch table has the following benefits compared to the baseline design:

- The dynamic power consumption is dramatically reduced because of the partitioning into 16 smaller tables;
- The CAM cell power is also reduced because we use only the lower 16 bits of the PC instead of the whole 32 bits;
- Another benefit of the new table is that since the table is very small (4-entry), we do not need a column sense amplifier. This also helps to reduce the total power consumed.

However, some overhead is also introduced by the new design. First, we need an address decoder to select one of the 16 tables. The total leakage power is increased (in a relative sense only) because while one of the smaller tables is active, the remaining 15 tables will be leaking. Fortunately, as we will show next, the PARE design overcomes all these disadvantages.

4.3 Location-set Based Group Analysis

As we mentioned above, compiler analysis help is required in order to partition the memory accesses into different groups such that we can use the new proposed PARE history table design.

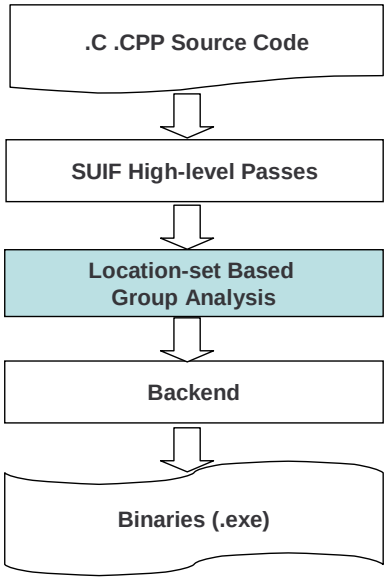


Figure 4.4. The flow diagram for the compiler passes.

Figure 4.4 shows the flow diagram of our compiler procedures. SUIF compiler infrastructure [99] is used to perform the analysis on intermediate files. Our location-set analysis pass is performed after the high-level SUIF passes.

Location-set analysis is a compiler analysis similar to pointer alias analysis [83]. By specifying locations for each memory object allocated by the program, a location set is calculated for each memory instruction. A key difference in our work is that we use an approximative runtime-biased analysis [40] that has no restrictions in terms of complexity or type of applications. Each location set contains the set of possible memory locations which could be accessed by the instruction. Detailed information of the runtime-biased compiler analysis has been presented in Section 3.3.

The location-sets for all the memory accesses are grouped based on their relationships and their potential effects on the prefetching decision-making process. The reason why we can group the memory accesses while not losing the accuracy of prefetching is because of the regular properties of the prefetching techniques: stride prefetching is based on the relationship within an array structure, while dependence-based pointer prefetching is based on the relationship between linked data structures.

The results of the location-set analysis, along with type information captured during SUIF analysis, give us the ability to group the memory accesses which relate during the prefetching decision-making process into the same group. For example, memory instructions which access the same location-set will be put in the same group, while the instructions accessing the same pointer structure will also be put in the same group.

In our analysis, group numbers are assigned within each procedure, and will be reused on a round-robin basis if necessary. The group numbers then will be written as annotations to the instructions and transferred to the SimpleScalar simulator via the binaries.

The essential steps of the location-set based group analysis are performed in the following order:

1. Runtime-biased pointer analysis is applied first to generate location set information for each pointer (including array accesses).

2. Type information for each memory access is identified through SUIF intermediate files. Then we identify the data type for each memory access. When a pointer is accessed, the type information of its target is also captured.

3. For array accesses, stride information is calculated based on array types (size of each element) and the pattern of array accesses (stride between elements).

4. For pointer structures, we identify all linked data structures connected to each other. The pattern will be the same pattern captured through hardware-based dependence-based data prefetching for linked data structures.

5. Group numbers are then assigned for each procedure based on the following criteria:

- Related array accesses are assigned to the same group, while non-related array accesses are each assigned into a single group.
- Accesses to each linked data structures (based on actual data objects, instead of based on types) are assigned to the same group.
- Memory accesses do not belong to any types identified are assigned into one single group by themselves.
- Group numbers are reused in a round-robin format if more than 16 groups are identified.

6. All group numbers are then transferred to the assembly code after applying the Machine SUIF [91] pass. Perl scripts are used to help insert the group number into the binaries for each memory access instruction.

After the location-set based compiler analysis, each memory access is assigned a group number. During the execution phasee, the architectural-level simulator will use the group number to direct the memory access to the corresponding entry in the new PARE table.

4.4 Chapter Summary

This chapter proposes a new prefetching scheme using a power-aware prefetching engine called PARE to improve the power-efficiency of hardware-based data prefetching mechanisms.

We show that with the help of compile-time location-set analysis, we can divide the memory accesses into different relationship groups, with each group consisting of memory accesses visiting only closely related location-sets. The compiler generated group numbers allow us to use the indexed history table in PARE.

Although we have implemented PARE on one specific data prefetching technique, we believe that PARE could be applied on other hardware prefetching techniques as

well. For example, stride prefetching will be a perfect candidate for PARE when we are dealing with scientific applications only.

CHAPTER 5

RESULTS AND ANALYSIS

We simulated the proposed energy-saving techniques and evaluated their impact on energy consumption as well as performance speedup. All the techniques are applied to the combined stride and dependence-based pointer prefetching. We first show the results by applying each of the techniques individually; next, we apply them together.

5.1 Compiler-Based Filtering

Fig. 5.1 shows the results for the three compiler-based techniques, first individually and then combined. The results shown are normalized to the baseline, which is the combined stride and pointer prefetching scheme without any of the new techniques.

Fig. 5.1(a) shows the number of prefetch table accesses. The compiler-based selective filtering (CBSF) works best for *parser*: more than 33% of all the prefetch table accesses are eliminated. On average, CBSF achieves about 7% reduction in prefetch table accesses. The compiler-assisted adaptive prefetching (CAAP) achieves the best reduction for *health*, about 20%, and on average saves 6%. The stride counter filtering (SC) technique removes 12% of prefetch table accesses for *bh*, with an average of over 5%. The three techniques combined filter out more than 20% of the prefetch table accesses for five out of ten benchmarks, with an average[1] of 18% across all applications.

Fig. 5.1(b) shows the extra L1 tag lookups due to prefetching. CBSF reduces the tag lookups by more than 8% on average; SC removes about 9%. CAAP does not

[1] All average numbers in this dissertation are calculated as arithmetic means.

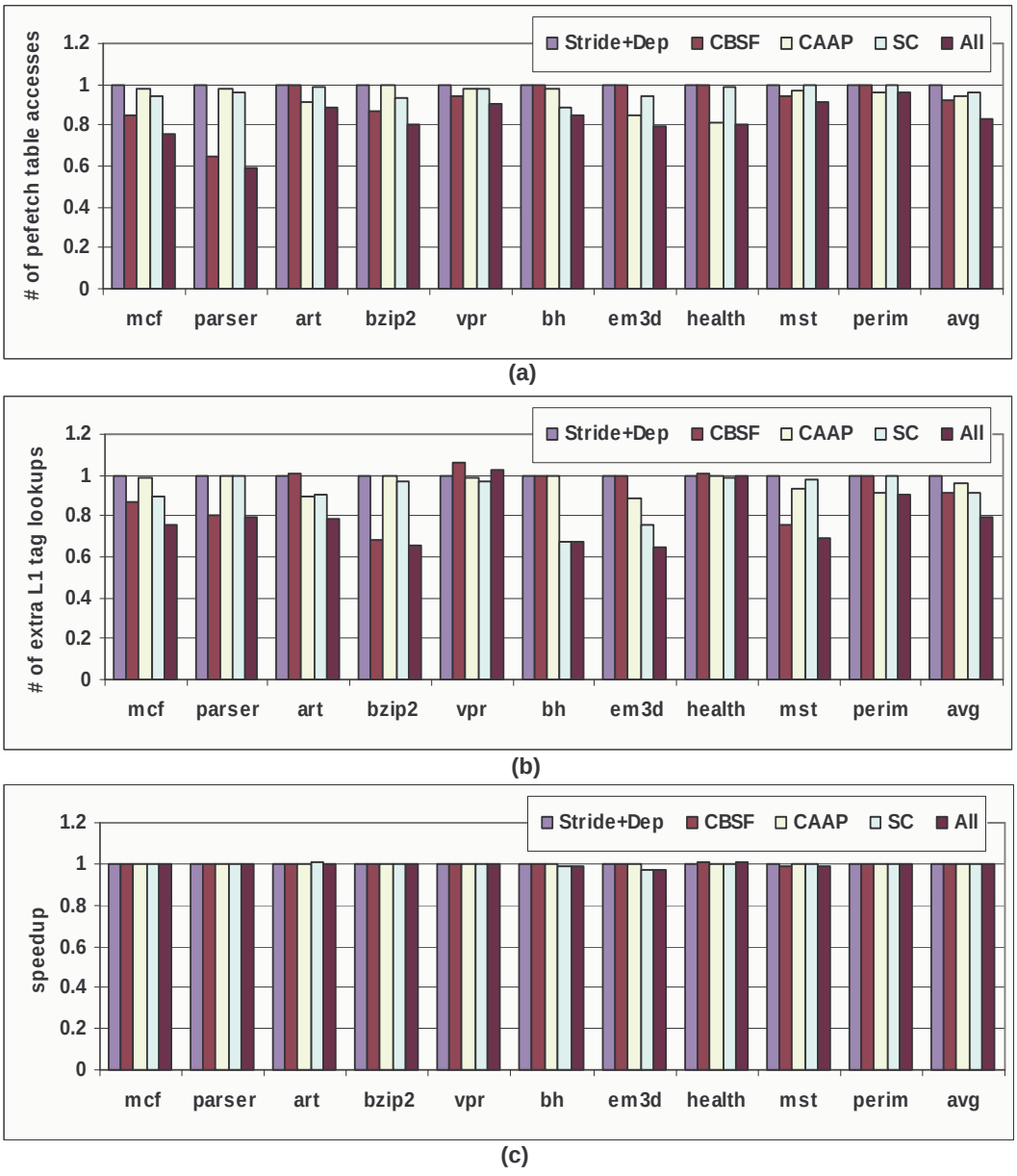


Figure 5.1. Simulation results for the three compiler-based techniques: (a) normalized number of the prefetch table accesses; (b) normalized number of the L1 tag lookups due to prefetching; and (c) impact on performance.

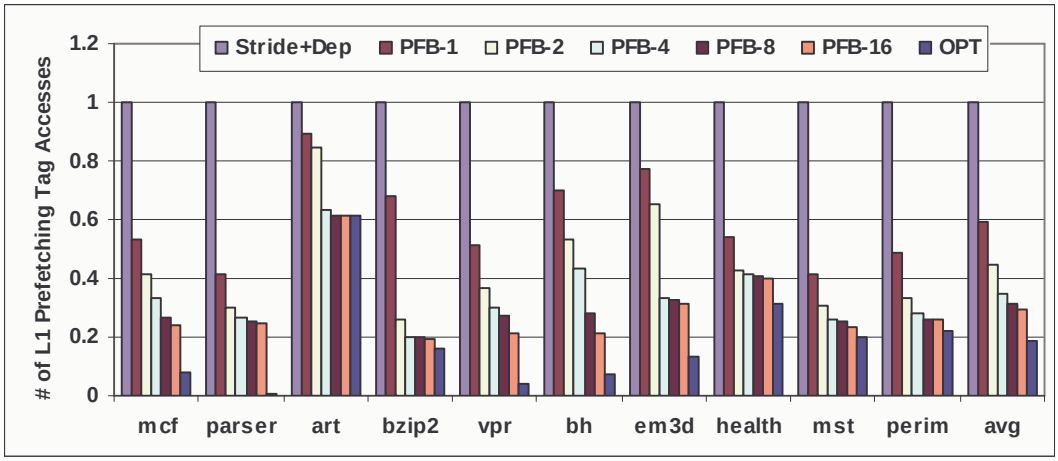


Figure 5.2. The number of L1 tag lookups due to prefetching after applying the hardware-based prefetch filtering technique with different sizes of PFB.

show a lot of savings, averaging just over 4%. The three techniques combined achieve tag-lookup savings of up to 35% for *bzip2*, averaging 21% compared to the combined prefetching baseline.

The performance penalty introduced by the three techniques is shown in Fig. 5.1(c). As shown, the performance impact is negligible. The only exception is *em3d*, which has less than 3% of performance degradation, due to filtering using SC.

5.2 Hardware Filtering Using PFB

Prefetch filtering using PFB will filter out those prefetch requests which would result in L1 cache hits if issued. We simulated different sizes of PFB to find out the best PFB size, considering both performance and energy consumption aspects. Fig. 5.2 shows the number of L1 tag lookups due to prefetching after applying the PFB prefetch filtering technique with PFB sizes ranging from 1 to 16.

As we can see from the figure, even a 1-entry PFB can filter out about 40% of all the prefetch tag accesses (on average). An 8-entry PFB can filter out over 70% of tag-checks with almost 100% accuracy. Increasing the PFB size to 16 does

Table 5.1. The number of PFB mispredictions during the whole run with different sizes of PFBs

Bench	PFB-1	PFB-2	PFB-4	PFB-8	PFB-16
mcf	0	0	0	1	9
parser	0	0	0	0	0
art	0	0	0	0	0
bzip2	0	0	0	0	0
vpr	0	0	0	0	0
bh	0	0	0	0	0
em3d	0	0	0	0	0
health	0	0	0	0	1
mst	0	0	11	11	11
perimeter	0	0	0	0	0

not increase the filtering percentage significantly. The increase is about 2% on the average compared to an 8-entry PFB, while the energy cost per access doubles.

We also show the ideal situation (OPT in the figure), where all the prefetch hits are filtered out. For some of the applications, such as *art* and *perim*, the 8-entry PFB is already very close to the optimal case. This shows that an 8-entry PFB is a good enough choice for this type of prefetch filtering.

As we stated before, PFB predictions are not always correct: it is possible that a prefetched address still resides in the PFB but it does not exist in the L1 cache (it has been replaced). The number of PFB mispredictions during the complete run of each application is shown in Table 5.1. Although the number of mispredictions increases with the size of the PFB, an 8-entry PFB makes almost perfect predictions and does not affect performance.

5.3 PARE Results

The prefetch hardware history table proposed was designed using the 70-nm BPTM technology and simulated using HSPICE with a supply voltage of 1V. Both leakage and dynamic power are measured. Figure 5.3 summarizes our results showing

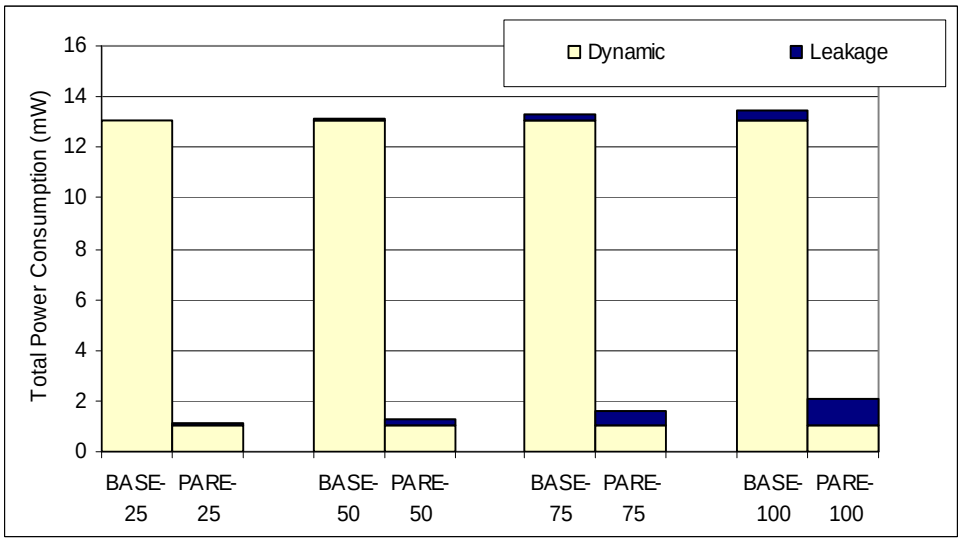


Figure 5.3. Power consumption for each history table access for PARE and baseline designs at different temperatures(℃).

the breakdown of dynamic and leakage power at different temperatures for both baseline and PARE history table designs.

From the figure, we can see that leakage power is very sensitive to temperature. The leakage power, which is initially 10% of the total power for the PARE design at room temperature (25℃), increases up to 50% as the temperature goes up to 100℃. This is because scaling and higher temperature cause subthreshold leakage currents to become a large component of the total power dissipation.

The new PARE table design proves to be much more power efficient than the baseline design. Although the leakage power consumption of PARE has more than doubled compared to the baseline design (this is because a smaller fraction of transistors are switching and a larger fraction are idle), the dynamic power of PARE is reduced dramatically, from 13mW to 1.05mW. The total power consumption is reduced by 7-11X. For our simulation, we used the power consumption result at 75℃, which is the typical temperature of a chip.

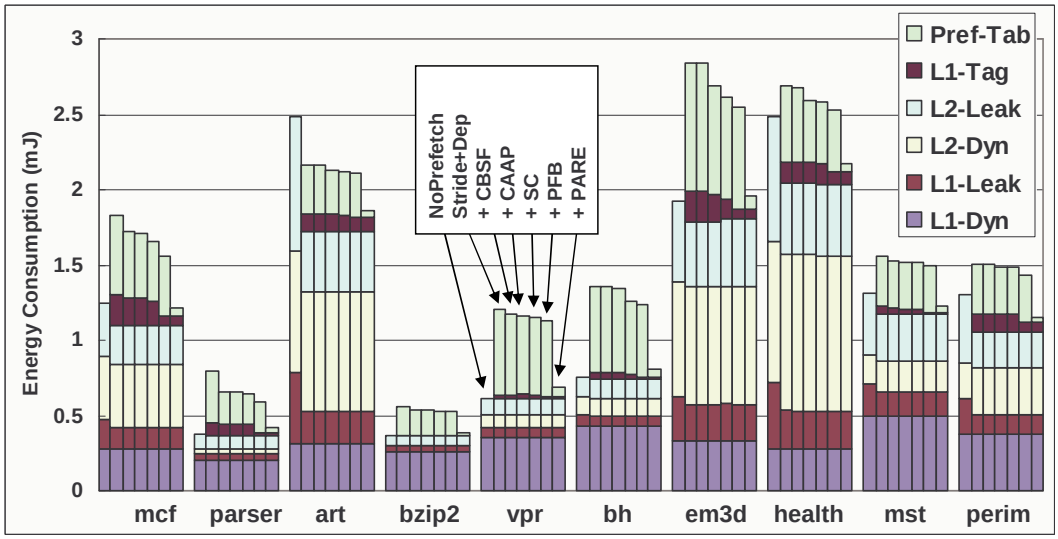


Figure 5.4. Energy consumption in the memory system after applying different energy-aware prefetching schemes.

5.4 Energy Savings

We apply the techniques in the following order CBSF, CAAP, SC, PFB, and PARE. We show the energy savings after each technique is added in Fig. 5.4.

Compared to the combined stride and pointer prefetching, the compiler-based selective filtering (CBSF) shows good improvement for *mcf* and *parser*, with an average reduction of total memory system energy of about 3%.

The second scheme, compiler-assisted adaptive prefetching (CAAP), reduces the energy consumed by about 2%, and shows good improvement for *health* and *em3d* (about 5%).

The stride counter approach is then applied. It reduces the energy consumption for both prefetch hardware tables and L1 prefetch tag accesses. It improves the energy consumption consistently for almost all benchmarks, achieving an average of just under 4% savings on the total energy consumption.

The hardware filtering technique is applied with an 8-entry PFB. The PFB reduces more than half of the L1 prefetch tag lookups and improves the total energy consumption by about 3%.

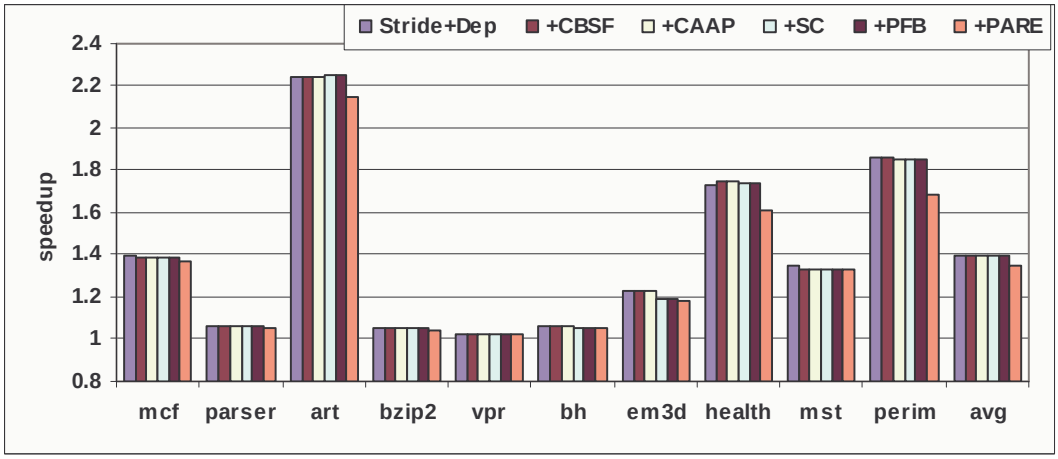


Figure 5.5. Performance speedup after applying different energy-aware prefetching schemes.

Overall, the four filtering techniques together reduce by almost 40% the energy overhead of the combined prefetching approach: the energy overhead due to prefetching is reduced from 28% to 17%. This is about 11% of the total memory system energy (including L1, L2 caches and prefetch tables).

Finally, we replace the prefetching hardware with the new PARE design and achieve energy savings of up to 8X for the prefetching table related energy (the topmost bar). After the application of PARE, we can see that the prefetching energy overhead becomes very small, and combined with the effect of leakage reduction due to performance improvement, half of the applications studied even show a total energy decrease after energy-aware data prefetching techniques applied.

5.5 Performance Degradation

Fig. 5.5 shows the performance statistics after applying each of the five techniques proposed, one after another. We can see that there is little performance impact for the four prefetch filtering techniques. On average, the three compiler-based filtering and PFB only affect the performance by less than 0.4%.

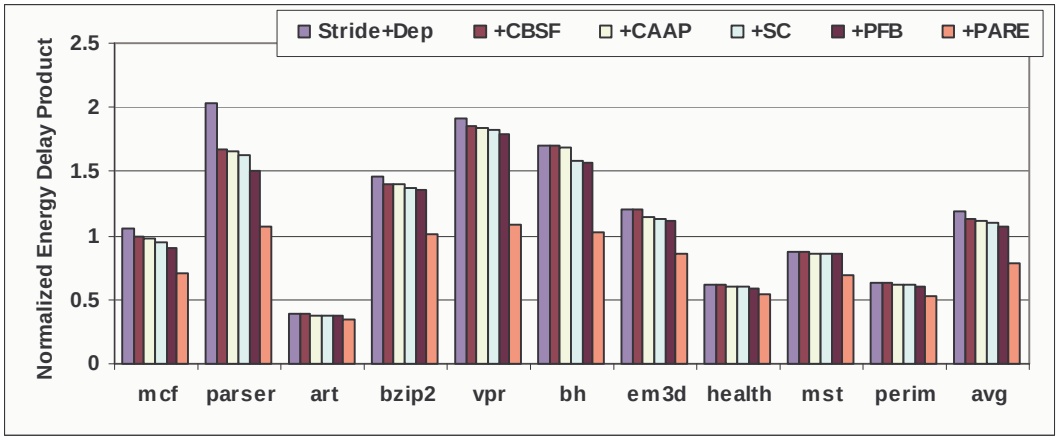


Figure 5.6. Energy-delay product with different energy-aware prefetching schemes.

The new prefetch engine PARE causes obvious performance degradation on most benchmarks, with *perim* suffers the largest degradation of 19% while still maintaining close to 70% speedup compared to no prefetching. On average, PARE causes a 5% performance impact, which is not negligible. However, the energy savings we achieve from PARE are clearly much more significant compared to the performance degradation, as demonstrated by the energy-delay product numbers presented next, and much of the prefetching-related speedup is still preserved.

5.6 Energy-Delay Product

Energy-delay product (EDP) is normally used as one of the important metrics to evaluate the effectiveness of an energy saving technique. A lower EDP indicates that the energy saving technique evaluated can be considered worthwhile because the energy saving is larger than the performance degradation (if any).

The EDP numbers of the proposed energy-aware techniques are shown in Fig. 5.6. All numbers are normalized to the case where no prefetching techniques are used. Compared to the combined stride and pointer prefetching, the EDP improves by almost 48% for *parser*. On average, the four power-aware prefetching techniques combined improve the EDP by about 33%.

Compared to the case where no prefetching is used, only four out of ten applications have a normalized EDP higher than 1 after all energy saving techniques are applied, with the average EDP being 21% lower than no prefetching. The EDP results show that data prefetching, if implemented with energy-aware schemes and hardware, could be very beneficial for both energy and performance.

5.7 Sensitivity Analysis

As the CMOS technology progresses rapidly, smaller and smaller transistors will be manufactured in the near future. With the technology scaling down, both dynamic and leakage power will be affected; as a result, energy consumption will also change significantly.

To demonstrate how our proposed techniques would be affected by future technologies, we run our simulation with new parameters based on the Predictive Technology Model (PTM) [1] at both 45-nm and 32-nm. A new set of results on the proposed energy-aware techniques are recalculated and presented next.

5.7.1 Power Modeling for Future Technologies

In order to estimate the energy consumption for next-generation technology, we first run simulations to capture the power numbers for all energy consuming components, including caches and hardware history tables, using HSPICE based on the PTM provided parameters. The power numbers at 45-nm and 32-nm PTM technologies are shown in Table 5.2.

With the technology scaling down, dynamic power goes down while leakage power goes up. We can clearly see this trend from the table. Leakage power, especial for L2 cache, goes up significantly, thus it will increase the total power consumption in our experiments. The power consumption for the prefetch hardware table is reduced at

Table 5.2. Power consumption at different technology nodes.

Hardware	Operation	70-nm	45-nm	32-nm
L1 Cache	Tag	6.5	5.7	5.2
	Read	9.5	10.2	11.9
	Write	10.3	11.1	12.7
	Leakage	3.1	4.5	6.7
	Reduced Leakage	0.8	1.2	1.8
L2 Cache	Tag	6.3	4.9	3.8
	Read	100.5	107.0	103.4
	Write	118.6	125.5	120.5
	Leakage	23.0	32.1	41.3
	Reduced Leakage	1.5	2.1	2.7
Baseline Table	Search	11.3	10.4	9.7
	Update	11.5	10.6	9.8
PARE Table	Search	1.41	1.30	1.21
	Update	1.44	1.33	1.23

smaller technology because the leakage for the small table is not significant enough compared to the decrease of its dynamic power.

5.7.2 45-nm Technology

We recalculated the energy consumption based on the power numbers presented above. Figure 5.7 shows the energy consumption at 45-nm PTM technology in the memory systems for cases both before and after applying the proposed energy-aware prefetching schemes. Compared to the power numbers in Figure 5.4 at 70-nm, the energy consumption typically goes up for all applications, mainly because of the increase of L1 and L2 leakage energy. However, the percentage of energy savings remains almost the same when we apply the energy savings techniques. On average, we can achieve about 24% total energy savings after applying all energy saving techniques compared to the prefetching baseline with no energy-aware optimizations.

When we consider the energy-delay product (EDP) numbers (shown in Figure 5.8), it shows that the relative (normalized to the baseline with no prefetching) EDP numbers obtained are even lower than what we achieved at 70-nm. The reason is that

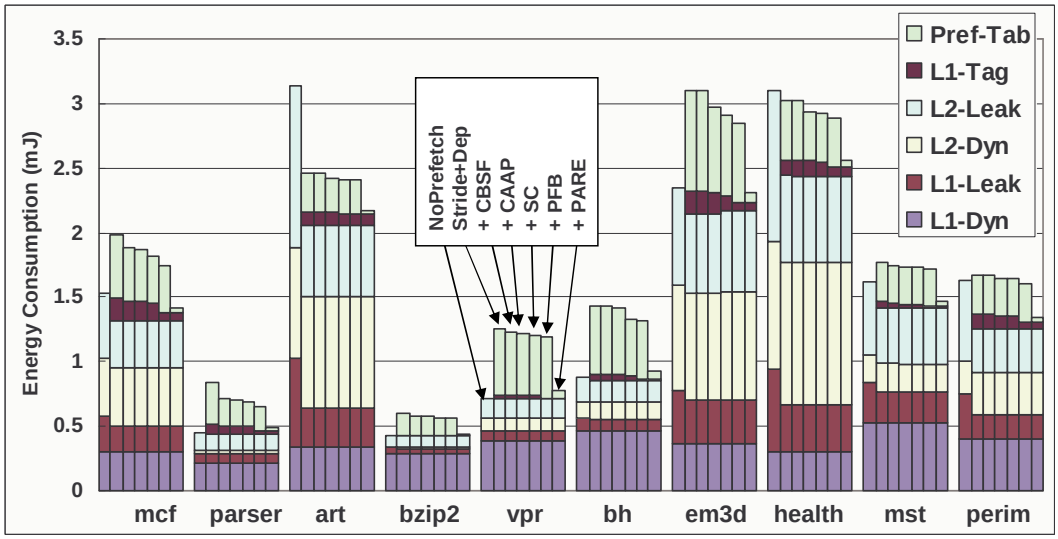


Figure 5.7. Energy consumption at 45-nm PTM technology in the memory system after applying different energy-aware prefetching schemes.

even without energy-aware optimizations, the reduction on leakage energy becomes larger at 45-nm compared to 70-nm due to the execution time reduction. On average, EDP numbers are improved by 30% compared to the prefetching baseline and 24% compared to no prefetching.

5.7.3 32-nm Technology

Figure 5.9 shows the energy consumption at 32-nm PTM technology in the memory systems for the applications studied. The total energy consumption goes up even further for all applications compared to 45-nm, mainly because of the increase of L1 and L2 leakage energy. The energy savings after applying all energy-aware techniques are about 20% on average compared to the prefetching baseline with no energy-aware optimizations.

Figure 5.10 shows the energy-delay product (EDP) numbers. It is worthwhile to point out that even with no energy-aware techniques applied, the average EDP with data prefetching is 4% better compared to no prefetching because leakage is even higher at 32-nm compared to 45-nm. On average, EDP numbers are improved by

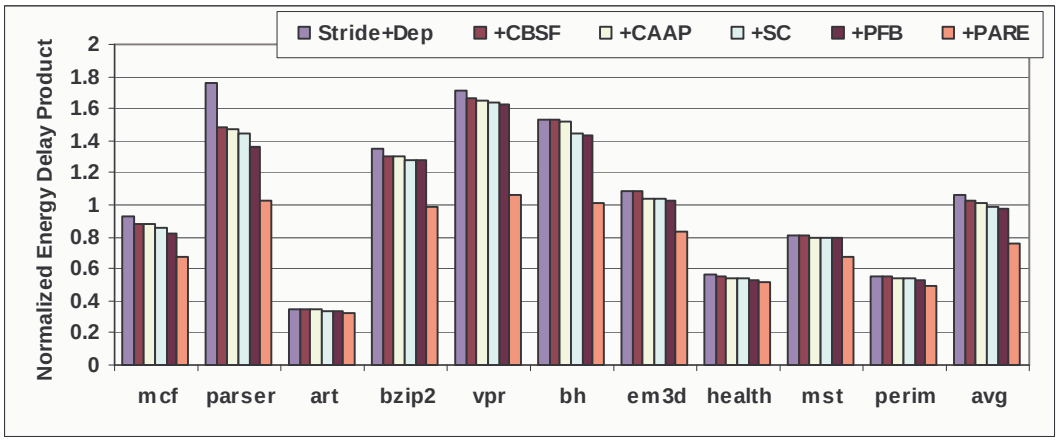


Figure 5.8. Energy-delay product at 45-nm PTM technology with different energy-aware prefetching schemes.

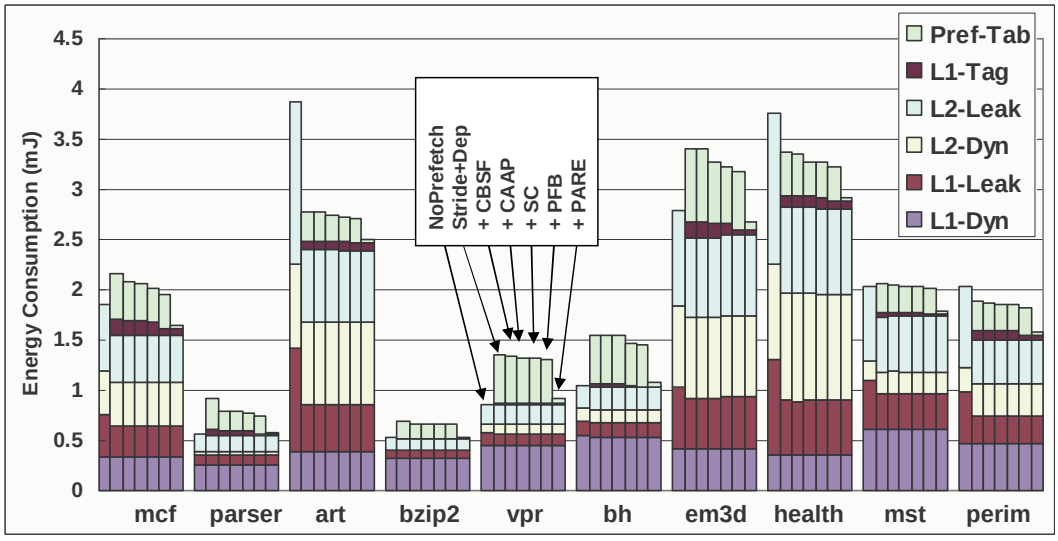


Figure 5.9. Energy consumption at 32-nm PTM technology in the memory system after applying different energy-aware prefetching schemes.

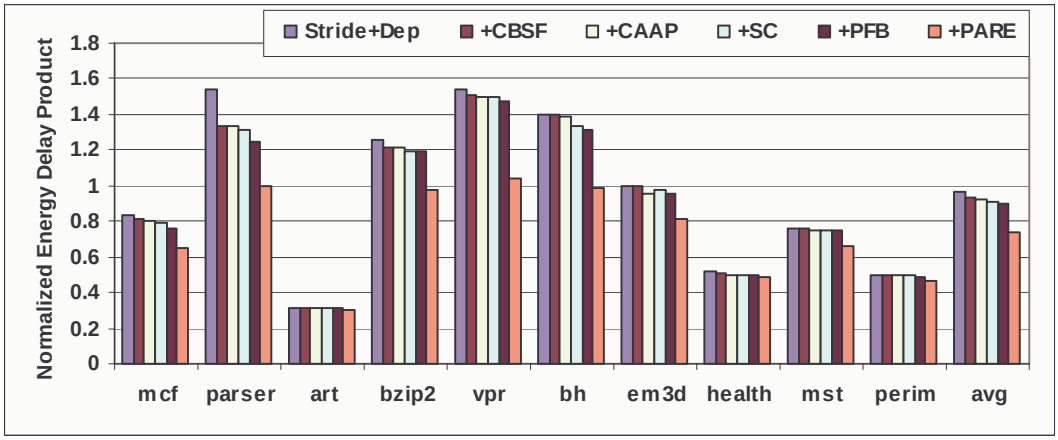


Figure 5.10. Energy-delay product at 32-nm PTM technology with different energy-aware prefetching schemes.

23% after all energy-aware optimizations compared to the prefetching baseline and 26% lower compared to no prefetching.

5.8 Chapter Summary

This chapter presents detailed experimental results for the proposed energy-aware data prefetching techniques. The energy-aware prefetch filtering techniques can reduce about 40% of the prefetch-related energy overhead. The location-set driven data prefetching works even better, reducing the prefetch hardware power cost by 7-11X. The techniques combined could overcome the energy overhead due to prefetching, improving the energy-delay product by 33% on average at the 70-nm PTM technology. Sensitivity analyses at smaller technologies of 45-nm and 32-nm show that our techniques will also work well for future-generation technologies.

CHAPTER 6

CONCLUSION

This dissertation explores the energy-efficiency aspects of data-prefetching techniques and proposes several new techniques to make prefetching energy-aware. Our proposed techniques include three compiler-based approaches which help to make the prefetch predictor more selective and filter out unnecessary prefetches based on static program information. We also propose a hardware based filtering technique to further reduce the energy overheads due to prefetching in the L1 cache. Our experiments show that the proposed techniques combined reduce the prefetching-related energy overheads by 40%, with almost no impact on performance.

We have achieved the following results:

- We implement a number of data prefetching techniques and provide detailed simulation results on both performance and energy consumption. The simulation results show that although aggressive prefetching techniques help to improve performance, in most of the applications they increase energy consumption by up to 30%. In designs implemented in deep-submicron 70-nm BPTM process technology, cache leakage dominates the energy consumption. We have found that if cache leakage is optimized with recently-proposed circuit-level techniques, most of the energy overhead is due to prefetch hardware related cost and unnecessary L1 data cache lookups related to prefetches that hit in the L1 cache.

- We propose several *energy-aware filtering* techniques for hardware data prefetching to reduce the energy overheads. The techniques include three compiler-based

filtering techniques and a hardware filter. These techniques are applied on one of the hardware prefetching techniques, which achieves the best performance speedup but also suffers the worst energy degradation. Our experiments show that the proposed techniques successfully reduce the prefetching-related energy overheads, by 40% on average, without reducing the performance benefits of data prefetching.

- To further reduce the energy overheads, we develop a new data prefetching technique called *location-set driven data prefetching*. A *power-aware prefetch engine* called PARE with a novel design of an indexed hardware history table is proposed at the circuit level. Compared to the conventional single-table design, the new prefetching table consumes 7-11X less power per access. With the help of compiler-based location-set analysis, we show that the proposed prefetching scheme improves energy consumption by as much as 40% in the data memory system.

Our experiments show that the proposed techniques if combined can eliminate the prefetching-related energy overheads, even turning data prefetching into an energy saving technique in many cases, with very small impact on performance compared to a baseline with prefetching but no energy-efficient techniques.

This dissertation demonstrates that, with proper energy-aware techniques, data prefetching has the potential to become an energy reduction technique in addition to (as traditionally regarded) a performance speedup technique. We believe our work has shown promising results in this direction and will help encourage more interests towards commercial implementation of hardware-based data prefetching.

6.1 Future Work

We identify the following potential future work closely related to the topics of this dissertation.

- *Hardware-software cooperative data prefetching.* This dissertation investigates how to make hardware-based data prefetching more energy-efficient with the help of compiler provided information. On the other hand, it will also be an interesting direction to investigate how to make software-based data prefetching more accurate and more energy-efficient with hardware support. In that case, software prefetching will insert explicit prefetching instructions to the binaries, however, the hardware could make decision on whether to do the actual prefetching based on hardware provided history information. This will have the potential to improve software prefetching in terms of both energy consumption and performance.

- *Data prefetching for embedded systems.* Current data prefetching techniques, especially hardware-based prefetching, are focused on general-purpose applications. For embedded systems, data prefetching will be quite different because it will be impossible to invest huge prefetching hardware in embedded processors because of the high hardware cost and power/energy restriction. However, simpler and lower-cost data prefetching techniques could be developed for embedded systems as the potential energy cost could be controlled based on what we have demonstrated in this dissertation.

- *Energy-aware instruction prefetching.* Although most of prefetching efforts have been focused on data prefetching, instruction prefetching is also an important component in contemporary processors. It would be interesting to see how different instruction prefetching techniques affect energy consumption and whether any technique could be applied to make it more energy-aware.

- *Implementation and experiments with commercial microprocessors.* Further efforts could be spent on investigating the effect of energy-efficient data prefetching techniques on a real commercial microprocessor as a case study. The study will provide more insights on the problem and the feedbacks would help make the techniques more practical.

BIBLIOGRAPHY

[1] Predictive Technology Model (PTM). http://www.eas.asu.edu/ ptm/.

[2] SPEC CPU2000 Benchmarks. The Standard Performance Evaluation Corporation, 2000. http://www.spec.org.

[3] Aamodt, T. M, Marcuello, P, Chow, P, González, A, Hammarlund, P, Wang, H, and Shen, J. P. A framework for modeling and optimization of prescient instruction prefetch. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (New York, NY, USA, 2003), ACM Press, pp. 13–24.

[4] Adl-Tabatabai, A.-R, Hudson, R. L, Serrano, M. J, and Subramoney, S. Prefetch injection based on hardware monitoring and object metadata. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation* (New York, NY, USA, 2004), ACM Press, pp. 267–276.

[5] Akkary, H, and Driscoll, M. A. A dynamic multithreading processor. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture* (Los Alamitos, CA, USA, 1998), IEEE Computer Society Press, pp. 226–236.

[6] Anacker, W, and Wang, C. P. Performance evaluation of computing systems with memory hierarchies. *IEEE Transactions on Computers*, 6 (December 1967), 764–773.

[7] Andersen, L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[8] Annavaram, M, Patel, J. M, and Davidson, E. S. Data prefetching by dependence graph precomputation. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture* (New York, NY, USA, 2001), ACM Press, pp. 52–61.

[9] Ashok, R, Chheda, S, and Moritz, C. A. Cool-mem: Combining statically speculative memory accessing with selective address translation for energy efficiency. In *ASPLOS* (2002).

[10] Ashok, R, Chheda, S, and Moritz, C. A. Coupling compiler-enabled and conventional memory accessing for energy efficiency. *ACM Trans. on Computer Systems 22*, 2 (2004), 180–213.

[11] Azizi, N, Moshovos, A, and Najm, F. N. Low-leakage asymmetric-cell sram. In *Proceedings of the 2002 international symposium on Low power electronics and design* (2002), ACM Press, pp. 48–51.

[12] Baer, J. L, and Chen, T. F. An effictive on-chip preloading scheme to reduce data access penalty. In *Supercomputing 1991* (Nov. 1991), pp. 179–186.

[13] Bennaser, M, and Moritz, C. A. A step-by-step design and analysis of low power caches for embedded processors. In *Boston Area Architecture Workshop (BARC-2005)* (Jan. 2005).

[14] Bergeron, R. D, Lipsa, D. R, and Rhodes, P. J. Iteration aware prefetching for scientific data. In *ACM-SE 43: Proceedings of the 43rd annual Southeast regional conference* (New York, NY, USA, 2005), ACM Press, pp. 216–217.

[15] Bernstein, D, Cohen, D, Freund, A, and Maydan, D. E. Compiler techinques for data prefetching on the PowerPC. In *International Conference on Parallel Architectures and Compilation Techniques* (June 1995), pp. 19–26.

[16] Bernstein, D, Cohen, D, and Freund, A. Compiler techniques for data prefetching on the powerpc. In *PACT '95: Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques* (Manchester, UK, UK, 1995), IFIP Working Group on Algol, pp. 19–26.

[17] Brooks, D, Tiwari, V, and Martonosi, M. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (June 2000), pp. 83–94.

[18] Burger, D. C, and Austin, T. M. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.

[19] Cahoon, B, and McKinley, K. S. Simple and effective array prefetching in java. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande* (New York, NY, USA, 2002), ACM Press, pp. 86–95.

[20] Cantin, J. F, Lipasti, M. H, and Smith, J. E. Stealth prefetching. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ACM Press, pp. 274–282.

[21] Chan, K. K, Hay, C. C, Keller, J. R, Kurpanek, G. P, Schumacher, F. X, and Zheng, J. Design of the HP PA 7200 CPU. *Hewlett-Packard Journal: technical information from the laboratories of Hewlett-Packard Company 47*, 1 (Feb. 1996), 25–33.

[22] Chen, T.-F, and Baer, J.-L. A performance study of software and hardware data prefetching schemes. In *ISCA '94: Proceedings of the 21ST annual international symposium on Computer architecture* (Los Alamitos, CA, USA, 1994), IEEE Computer Society Press, pp. 223–232.

[23] Chen, W. Y, Mahlke, S. A, Chang, P. P, and Hwu, W.-M. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *The 24th Annual International Symposium on Microarchitecture* (Nov. 1991), pp. 69–73.

[24] Chi, C.-H, and Cheung, C.-M. Hardware-driven prefetching for pointer data references. In *ICS '98: Proceedings of the 12th international conference on Supercomputing* (New York, NY, USA, 1998), ACM Press, pp. 377–384.

[25] Chilimbi, T. M, and Hirzel, M. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI-02)* (June 2002), Cindy Norris and Jr. James B. Fenwick, Eds., pp. 199–209.

[26] Choi, J.-D, Burke, M, and Carini, P. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, Jan. 10–13, 1993), ACM Press, pp. 232–245.

[27] Choi, S, Kohout, N, Pamnani, S, Kim, D, and Yeung, D. A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching. *ACM Trans. Comput. Syst. 22*, 2 (2004), 214–280.

[28] Collins, J, Sair, S, Calder, B, and Tullsen, D. M. Pointer cache assisted prefetching. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture* (Los Alamitos, CA, USA, 2002), IEEE Computer Society Press, pp. 62–73.

[29] Cooksey, R, Jourdan, S, and Grunwald, D. A stateless content-directed data prefetching mechanism. In *Tenth international conference on architectural support for programming languages and operating systems(ASPLOS-X)* (2002), ACM Press, pp. 279–290.

[30] Doshi, G, Krishnaiyer, R, and Muthukumar, K. Optimizing software data prefetches with rotating registers. In *International Conference on Parallel Architectures and Compilation Techniques* (Sept. 2001), pp. 257–267.

[31] Drach, N. Hardware implementation issues of data prefetching. In *ICS '95: Proceedings of the 9th international conference on Supercomputing* (New York, NY, USA, 1995), ACM Press, pp. 245–254.

[32] Dwyer, H, and Torng, H. C. An out-of-order superscalar processor with speculative execution and fast, precise interrupts. In *Proceedings of the 25th Annual International Symposium on Microarchitecture* (Portland, Oregon, Dec. 1–4, 1992), IEEE Computer Society TC-MICRO and ACM SIGMICRO, pp. 272–281.

[33] Emami, M, Ghiya, R, and Hendren, L. J. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conference on Programming Language Design and Implementation* (1994), pp. 242–256.

[34] Flautner, K, Kim, N. S, Martin, S, Blaauw, D, and Mudge, T. Drowsy caches: simple techniques for reducing leakage power. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture* (Washington, DC, USA, 2002), IEEE Computer Society, pp. 148–157.

[35] Ganusov, I, and Burtscher, M. Future execution: A prefetching mechanism that uses multiple cores to speed up single threads. *ACM Trans. Archit. Code Optim. 3*, 4 (2006), 424–449.

[36] Ghose, K, and Kamble, M. B. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design* (New York, NY, USA, 1999), ACM Press, pp. 70–75.

[37] Gornish, E. H, and Veidenbaum, A. An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. *Int. J. Parallel Program. 27*, 1 (1999), 35–70.

[38] Gowan, M. K, Biro, L. L, and Jackson, D. B. Power considerations in the design of the alpha 21264 microprocessor. In *Proceedings of the 1998 Conference on Design Automation (DAC-98)* (June 1998), pp. 726–731.

[39] Guo, Y, Chheda, S, Koren, I, Krishna, C. M, and Moritz, C. A. Energy characterization of hardware-based data prefetching. In *International Conference on Computer Design (ICCD'04)* (Oct. 2004).

[40] Guo, Y, Chheda, S, and Moritz, C. A. Runtime biased pointer reuse analysis and its application to energy efficiency. In *Workshop on Power-Aware Computer Systems(PACS) at Micro-36* (Dec. 2003), pp. 1–15.

[41] Harizopoulos, S, and Ailamaki, A. Improving instruction cache performance in oltp. *ACM Trans. Database Syst. 31*, 3 (2006), 887–920.

[42] Hinton, G, Sager, D, Upton, M, Boggs, B, Carmean, D, Kyker, A, and Roussel, R. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q1 (Feb. 2001), 13.

[43] Hu, Z, Kaxiras, S, and Martonosi, M. Let caches decay: reducing leakage energy via exploitation of cache generational behavior. *ACM Trans. Comput. Syst. 20*, 2 (2002), 161–190.

[44] Hughes, C. J, and Adve, S. V. Memory-side prefetching for linked data structures for processor-in-memory systems. *J. Parallel Distrib. Comput. 65*, 4 (2005), 448–463.

[45] Hur, I, and Lin, C. Memory prefetching using adaptive stream detection. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 397–408.

[46] Inagaki, T, Onodera, T, Komatsu, K, and Nakatani, T. Stride prefetching by dynamically inspecting objects. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI-03)* (June 2003), pp. 269–277.

[47] Joseph, D, and Grunwald, D. Prefetching using markov predictors. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture* (New York, NY, USA, 1997), ACM Press, pp. 252–263.

[48] Kadayif, I, Kandemir, M, and Li, F. Prefetching-aware cache line turnoff for saving leakage energy. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation* (New York, NY, USA, 2006), ACM Press, pp. 182–187.

[49] Kadayif, I, Kandemir, M, and Li, F. Prefetching-aware cache line turnoff for saving leakage energy. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation* (New York, NY, USA, 2006), ACM Press, pp. 182–187.

[50] Kessler, R. E. The Alpha 21264 microprocessor. *IEEE Micro* (March/April 1999), 24–36.

[51] Ki, A, and Knowles, A. E. Adaptive data prefetching using cache information. In *ICS '97: Proceedings of the 11th international conference on Supercomputing* (New York, NY, USA, 1997), ACM Press, pp. 204–212.

[52] Kim, N. S, Blaauw, D, and Mudge, T. Leakage power optimization techniques for ultra deep sub-micron multi-level caches. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design* (Washington, DC, USA, 2003), IEEE Computer Society.

[53] Kim, N. S, Flautner, K, Blaauw, D, and Mudge, T. Drowsy instruction caches: leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture* (Los Alamitos, CA, USA, 2002), IEEE Computer Society Press, pp. 219–230.

[54] Klaiber, A. C, and Levy, H. M. An architecture for software-controlled data prefetching. In *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture* (New York, NY, USA, 1991), ACM Press, pp. 43–53.

[55] Kumar, R, and Ravikumar, C. P. Leakage power estimation for deep submicron circuits in an asic design environment. In *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design* (Washington, DC, USA, 2002), IEEE Computer Society, p. 45.

[56] Landi, W, and Ryder, B. G. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Notices 27*, 7 (July 1992), 235–248. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*.

[57] Li, J.-J, and Hwang, Y.-S. Snug set-associative caches: reducing leakage power while improving performance. In *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design* (New York, NY, USA, 2005), ACM Press, pp. 345–350.

[58] Li, Y, Parikh, D, Zhang, Y, Sankaranarayanan, K, Stan, M, and Skadron, K. State-preserving vs. non-state-preserving leakage control in caches. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe* (Washington, DC, USA, 2004), IEEE Computer Society, p. 10022.

[59] Lipasti, M. H, Schmidt, W. J, Kunkel, S. R, and Roediger, R. R. SPAID: software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th annual international symposium on Microarchitecture* (Nov. 1995), pp. 231–236.

[60] Lu, J, Chen, H, Fu, R, Hsu, W.-C, Othmer, B, Yew, P.-C, and Chen, D.-Y. The performance of runtime data cache prefetching in a dynamic optimization system. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2003), IEEE Computer Society.

[61] Luk, C.-K, and Mowry, T. C. Compiler-based prefetching for recursive data structures. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)* (Oct. 1996), pp. 222–233.

[62] Luk, C.-K, and Mowry, T. C. Cooperative prefetching: compiler and hardware support for effective instruction prefetching in modern processors. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture* (Los Alamitos, CA, USA, 1998), IEEE Computer Society Press, pp. 182–194.

[63] Luk, C.-K, Muth, R, Patil, H, Weiss, R, Lowney, P. G, and Cohn, R. Profile-guided post-link stride prefetching. In *Proceedings of the 16th International Conference on Supercomputing (ICS-02)* (June 2002), pp. 167–178.

[64] M. Weiser, B. Welch, A. J. D, and Shenker, S. Scheduling for reduced cpu energy. In *Operating Systems Design and Implementation* (1994), pp. 13–23.

[65] Meng, K, and Joseph, R. Process variation aware cache leakage management. In *ISLPED '06: Proceedings of the 2006 international symposium on Low power electronics and design* (New York, NY, USA, 2006), ACM Press, pp. 262–267.

[66] Mohyuddin, N, Bhatti, R, and Dubois, M. Controlling leakage power with the replacement policy in slumberous caches. In *CF '05: Proceedings of the 2nd conference on Computing frontiers* (New York, NY, USA, 2005), ACM Press, pp. 161–170.

[67] Montanaro, J, Witek, R. T, Anne, K, Black, A. J, Cooper, E. M, Dobberpuhl, D. W, Donahue, P. M, Eno, J, Hoeppner, G. W, Kruckemyer, D, Lee, T. H, Lin, P. C. M, Madden, L, Murray, D, Pearce, M. H, Santhanam, S, Snyder, K. J, Stephany, R, and Thierauf, S. C. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *Digital Technical Journal of Digital Equipment Corporation 9*, 1 (1997).

[68] Mowry, T. *Tolerating Latency Through Software Controlled Data Prefetching.* PhD thesis, Dept. of Computer Science, Stanford University, Mar. 1994.

[69] Mowry, T. C, Lam, M. S, and Gupta, A. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 1992), pp. 62–73.

[70] Patterson, R. H, and Gibson, G. A. Exposing I/O concurrency with informed prefetching. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems* (Sept. 1994), pp. 7–16.

[71] Pering, T, Burd, T, and Brodersen, R. The simulation and evaluation of dynamic voltage scaling algorithms. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design* (New York, NY, USA, 1998), ACM Press, pp. 76–81.

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

[73] Pillai, P, and Shin, K. G. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2001), ACM Press, pp. 89–102.

[74] Pinter, S. S, and Yoaz, A. Tango: a hardware-based data prefetching technique for superscalar processors. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture* (Washington, DC, USA, 1996), IEEE Computer Society, pp. 214–225.

[75] Porterfield, A. K. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, May 1989.

[76] Puzak, T. R, Hartstein, A, Emma, P. G, and Srinivasan, V. When prefetching improves/degrades performance. In *CF '05: Proceedings of the 2nd conference on Computing frontiers* (New York, NY, USA, 2005), ACM Press, pp. 342–352.

[77] Ramamoorthy, C. V, and Kim, K. H. Pipelining – the generalized concept and sequencing strategies. In *Proceedings AFIPS National Computer Conference (NCC)* (Montvale, NJ, 1974), vol. 43, AFIPS Press, pp. 289–297.

[78] Reinman, G, Calder, B, and Austin, T. Fetch directed instruction prefetching. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture* (Washington, DC, USA, 1999), IEEE Computer Society, pp. 16–27.

[79] Rogers, A, Carlisle, M. C, Reppy, J. H, and Hendren, L. J. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems 17*, 2 (Mar. 1995), 233–263.

[80] Roth, A, Moshovos, A, and Sohi, G. S. Dependence based prefetching for linked data structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (oct 1998), pp. 115–126.

[81] Roth, A, and Sohi, G. S. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th annual international symposium on Computer architecture* (1999), IEEE Computer Society Press, pp. 111–121.

[82] Ruf, E. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)* (La Jolla, California, 18–21 June 1995), pp. 13–22.

[83] Rugina, R, and Rinard, M. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, May 1–4, 1999), pp. 77–90.

[84] Sair, S, Sherwood, T, and Calder, B. A decoupled predictor-directed stream prefetching architecture. *IEEE Trans. Comput. 52*, 3 (2003), 260–276.

[85] Santhanam, V, Gornish, E. H, and Hsu, H. Data prefetching on the HP PA8000. In *24th Annual International Symposium on Computer Architecture* (May 1997).

[86] Shapiro, M, and Horwitz, S. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France, 15–17 Jan. 1997), pp. 1–14.

[87] Sharma, S, Beu, J. G, and Conte, T. M. Spectral prefetcher: An effective mechanism for l2 cache prefetching. *ACM Trans. Archit. Code Optim. 2*, 4 (2005), 423–450.

[88] Smith, A. J. Sequential program prefetching in memory bierarchies. *IEEE Computer 11*, 12 (Dec. 1978), 7–21.

[89] Smith, A. J. Cache memories. *ACM Computing Surveys (CSUR) 14*, 3 (1982), 473–530.

[90] Smith, J. E, and Hsu, W.-C. Prefetching in supercomputer instruction caches. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing* (Los Alamitos, CA, USA, 1992), IEEE Computer Society Press, pp. 588–597.

[91] Smith, M. Extending suif for machine-dependent optimizations. In *Proc. First SUIF Compiler Workshop* (Jan. 1996).

[92] Srinivasan, V, Tyson, G. S, and Davidson, E. S. A static filter for reducing prefetch traffic. Tech. Rep. CSE-TR-400-99, University of Michigan.

[93] Steensgaard, B. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)* (St. Petersburg, Florida, Jan. 21–24, 1996), ACM Press, pp. 32–41.

[94] Stocks, P. A, Ryder, B. G, Landi, W. A, and Zhang, S. Comparing flow and context sensitivity on the modification-side-effects problem. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA-98)* (New York, Mar. 2–5 1998), vol. 23,2 of *ACM Software Engineering Notes*, ACM Press, pp. 21–31.

[95] Unsal, O. S, Ashok, R, Koren, I, Krishna, C. M, and Moritz, C. A. Cool-cache for hot multimedia. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture* (2001), IEEE Computer Society, pp. 274–283.

[96] Unsal, O. S, Ashok, R, Koren, I, Krishna, C. M, and Moritz, C. A. Cool-cache: A compiler-enabled energy efficient data caching framework for embedded/multimedia processors. *ACM Trans. on Embedded Computing Systems 2*, 3 (2003), 373–392.

[97] Vanderwiel, S. P, and Lilja, D. J. Data prefetch mechanisms. *ACM Comput. Surv. 32*, 2 (2000), 174–199.

[98] Wang, Z, Burger, D, McKinley, K. S, Reinhardt, S. K, and Weems, C. C. Guided region prefetching: a cooperative hardware/software approach. In *Proceedings of the 30th Annual International Symposium on Computer Architecture* (June 2003), pp. 388–398.

[99] Wilson, R, French, R, Wilson, C, Amarasinghe, S, Anderson, J, Tjiang, S, Liao, S.-W, Tseng, C.-W, Hall, M. W, Lam, M, and Hennessy, J. L. SUIF: A parallelizing and optimizing research compiler. Tech. Rep. CSL-TR-94-620, Computer Systems Laboratory, Stanford University, May 1994.

[100] Wilson, R. P, and Lam, M. S. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)* (La Jolla, California, June 1995).

[101] Witchel, E, Larsen, S, Ananian, C. S, and Asanović, K. Direct addressed caches for reduced power consumption. In *Proceedings of the 34th Annual International Symposium on Microarchitecture* (Austin, Texas, Dec. 1–5, 2001), IEEE Computer Society TC-MICRO and ACM SIGMICRO, pp. 124–133.

[102] Wolf, M. E. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford University, Aug. 1992.

[103] Wolf, M. E, and Lam, M. S. A data locality optimizing algorithm. *SIGPLAN Notices 26*, 6 (June 1991), 30–44. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*.

[104] Wu, Y. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI-02)* (June 2002), Cindy Norris and Jr. James B. Fenwick, Eds., pp. 210–221.

[105] Xia, C, and Torrellas, J. Instruction prefetching of systems codes with layout optimized for reduced cache misses. In *ISCA '96: Proceedings of the 23rd annual international symposium on Computer architecture* (New York, NY, USA, 1996), ACM Press, pp. 271–282.

[106] Yang, C.-L, and Lebeck, A. R. Push vs. pull: data movement for linked data structures. In *ICS '00: Proceedings of the 14th international conference on Supercomputing* (New York, NY, USA, 2000), ACM Press, pp. 176–186.

[107] Yang, C.-L, Lebeck, A. R, Tseng, H.-W, and Lee, C.-H. Tolerating memory latency through push prefetching for pointer-intensive applications. *ACM Trans. Archit. Code Optim. 1*, 4 (2004), 445–475.

[108] Yeager, K. C. The mips r10000 superscalar microprocessor. *IEEE Micro 16*, 2 (1996), 28–40.

[109] Young, H. C, and Shekita, E. J. An inteliigent I-cache prefetch mechanism. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors* (Cambridge, MA, Oct. 1993), Edna Straub, Ed., IEEE Computer Society Press, pp. 44–53.

[110] Yuan, W, and Nahrstedt, K. Energy-efficient cpu scheduling for multimedia applications. *ACM Trans. Comput. Syst. 24*, 3 (2006), 292–331.

[111] Zhang, M, and Asanovic, K. Highly-associative caches for low-power processors. In *Kool Chips Workshop, 33rd International Symposium on Microarchitecture* (Dec. 2000).

[112] Zhang, W, Hu, J. S, Degalahal, V, Kandemir, M, Vijaykrishnan, N, and Irwin, M. J. Compiler-directed instruction cache leakage optimization. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture* (Los Alamitos, CA, USA, 2002), IEEE Computer Society Press, pp. 208–218.

[113] Zhang, W, Kandemir, M, Karakoy, M, and Chen, G. Reducing data cache leakage energy using a compiler-based approach. *ACM Trans. on Embedded Computing Sys. 4*, 3 (2005), 652–678.

[114] Zhang, Y, Haga, S, and Barua, R. Execution history guided instruction prefetching. In *ICS '02: Proceedings of the 16th international conference on Supercomputing* (New York, NY, USA, 2002), ACM Press, pp. 199–208.

[115] Zhang, Y, Haga, S, and Barua, R. Execution history guided instruction prefetching. *J. Supercomput. 27*, 2 (2004), 129–147.