Energy-Aware Data Prefetching for General-Purpose Programs

Yao Guo[1], Saurabh Chheda[2], Israel Koren[1], C. Mani Krishna[1], and Csaba Andras Moritz[1]

[1]*Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003*

[1]{*yaoguo, koren, krishna, andras*}*@ecs.umass.edu*

[2]*BlueRISC Inc., Hadley, MA 01035,* {*chheda@bluerisc.com*}

ABSTRACT

There has been intensive research on data prefetching focusing on performance improvement, however, the energy aspect of prefetching is relatively unknown. Our experiments show that although software prefetching tends to be more energy efficient, hardware prefetching outperforms software prefetching on most of the applications in terms of performance. This paper proposes several techniques to make hardware-based data prefetching power-aware. 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.

Keywords

Data Prefetching, Compiler Analysis, Energy Efficiency, Prefetch Filtering

1. INTRODUCTION

In recent years, energy and power efficiency have become key design objectives in microprocessors, in both embedded and general-purpose domains. Although considerable research [27, 3, 24, 25, 9, 21, 17, 18] has been focused on improving the performance of prefetching mechanisms, the impact of prefetching techniques on processor energy efficiency has not yet been fully investigated.

Our experiments [12] 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 [11, 20], this is equivalent to more than 15% increase in chip-wide energy consumption.

We implemented two software prefetching techniques [22, 18] to compare the performance and energy efficiency of hardware and software prefetching. The results show that in general software prefetching is more energy-efficient while hardware prefetching yields better performance for most applications. In this paper, we focus on making one of the hardware prefetching techniques (which yields the best performance speedup) more energy-efficient without sacrificing its performance benefits.

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 experiments 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.

We propose several power-aware techniques for hardware data prefetching to reduce the energy overheads stated above. The techniques include:

- A compiler-based prefetch filtering approach, which reduces energy consumption by only searching the prefetch hardware tables for selective memory instructions identified by the compiler;
- A compiler-assisted selective prefetching mechanism, which utilizes compiler supplied static information to selectively apply different prefetching schemes depending on predicted access patterns;
- A compiler-driven filtering technique using a runtime stride counter designed to reduce prefetching energy consumption wasted on memory access patterns with very small strides; and
- A hardware-based filtering technique applied to further reduce the L1 cache related energy overhead due to prefetching.

The SimpleScalar [6] 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. The compiler passes for both software prefetching and power-aware hardware prefetching are implemented using the SUIF infrastructure [30]. 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.

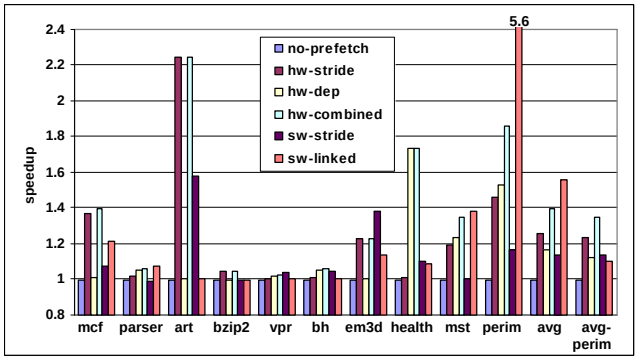

Figure 1: Performance speedup for different prefetching schemes.

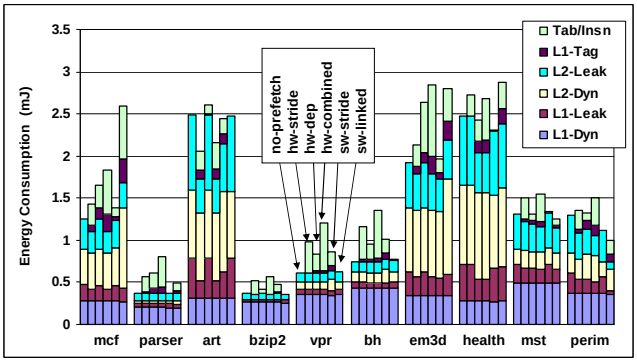

Figure 2: Total cache energy consumption.

The rest of this paper is organized as follows. Section 2 describes the energy overhead of data prefetching. The energy-aware prefetching solutions are presented in Section 3. Section 4 presents the experimental assumptions. Section 5 gives a detailed analysis of the results. The related work is presented in Section 6, and we conclude with Section 7.

2. MOTIVATION

Our previous work [12] has evaluated the energy perspective of hardware-based data prefetching techniques. In addition to the hardware techniques evaluated, we also implemented two software prefetching techniques [22, 18] and compare their performance and energy consumptions to the hardware mechanisms. More details and background on the this section can be found in [12].

To explore the energy aspects of data prefetching techniques, we provide experimental results for the following five prefetching techniques:

- *Stride prefetching* [3] - Focuses on array-like structures, it catches constant strides in memory accesses and prefetches using the stride information;
- *Dependence-based prefetching* [24] - 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 [22] - Use the compiler to insert prefetch instructions for strided array accesses.
- *Compiler-based prefetching on Linked Data Structures* - Uses the greedy approach in [18] to prefetch pointer structures.

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 1. The first five benchmarks are from SPEC2000 benchmarks; the last five are Olden benchmarks which contains many pointers and linked data structures.

As we expected, 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).

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%.

The memory system, including caches, consumes a significant fraction of the total processor energy. For example, the caches and translation look-aside buffers (TLB) consume 23% of the total power in the Alpha 21264 [11], and the caches alone use up 42% of the energy in the StrongARM 110 [20]. It is clear that the L1 and L2 caches, and the prefetching hardware tables could amount to about half of the total processor energy consumption.

We calculated the total energy consumption in the memory system for each prefetching technique based on HSpice; for more details, see Section 4. The results are shown in Figure 2. 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 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,

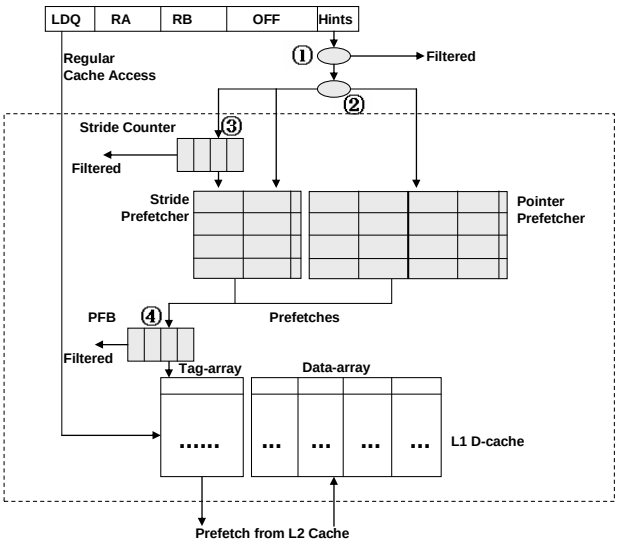

Figure 3: Power-aware prefetching architecture for general-purpose programs

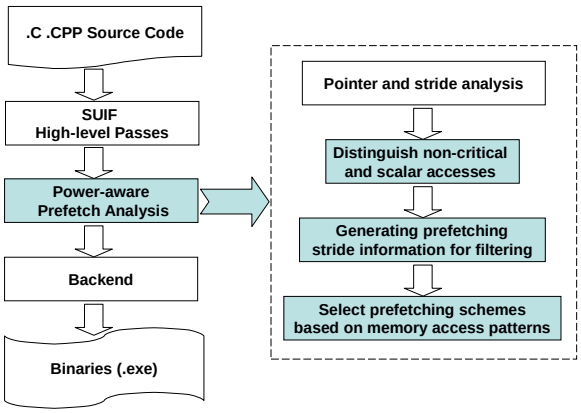

Figure 4: Compiler analysis used for power-aware prefetching

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?

3. ENERGY-AWARE PREFETCHING TECHNIQUES

In this section, we will discuss how to reduce the energy overhead for the most aggressive hardware prefetching scheme, the combined stride and pointer prefetching. This scheme gives the best performance speedup for general-purpose programs, but it is the worst in terms of energy efficiency.

3.1 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.

Figure 3 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 Figure 3, are:

1. A compiler-based prefetch filtering approach which reduces prefetch hardware energy cost by only searching the prefetch hardware tables for memory instructions selected by the compiler;

2. A compiler-assisted selective prefetching mechanism which utilizes the compiler supplied static information to selectively apply different prefetching schemes depending on predicted access patterns;

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

4. A hardware-based filtering technique 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.

Figure 4 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 [13] 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.

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).

3.2 Compiler-Based Prefetch Filtering (CBPF)

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.

3.3 Compiler-Assisted Selective Prefetching (CASP)

Another compiler approach focuses on how to help the prefetch predictor to 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 passes. 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 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 [13].

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 will be set to a maximal value (decided by cache_line_size/stride) and is then decremented by one 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 Figure 3.

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 Section 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).

4. EXPERIMENTAL ASSUMPTIONS

4.1 Experimental Framework

We implement the hardware-based data prefetching techniques by modifying the SimpleScalar [6] simulator. The software prefetching schemes are implemented using SUIF [30] and simulated with the modified SimpleScalar which can recognize prefetch instructions. We also use SUIF to implement the compiler passes for power-aware prefetching, generating annotations for all the prefetching hints which we later transfer to assembly codes. 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 1.

The benchmarks evaluated are listed in Table 2. The SPEC2000 benchmarks [1] use mostly array-based data structures, while the Olden benchmark suite [23] 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 one (perimeter), since they complete in relatively short time.

Table 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

Table 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

4.2 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 proposed low-power circuit [32] that we implemented in 100-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 100-nm technology would have significant leakage, we apply a recently proposed circuit-level leakage reduction technique called asymmetric SRAM cells [2]. This is necessary because otherwise our conclusions would be skewed due to very high leakage power. The *speed enhanced cell* in [2] 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 for our L1 and L2 caches are shown in Table 3.

If an L1 miss occurs, energy is consumed not only in L1 tag-lookup, 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.

Each prefetching history table is implemented as a 64×64 fully-associated CAM-array. The power consumption for

Table 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 4: Prefetch hardware table and power consumption

Table implementation	64×64 CAM-array
P_update (including lookup)	7.4mW
P_lookup	7.3mW

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 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 [5]. Thus we assume that each prefetch instruction executed would consume an extra cost of roughly two times the L1 cache read energy cost in Figure 2.

5. RESULTS AND ANALYSIS

We simulated each of the four 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 four techniques individually; and then, we apply them together in order.

5.1 Compiler-Based Techniques

Figure 5 shows the results for the three compiler-based techniques, first separately 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.

Figure 5(a) shows the number of prefetch table accesses. The compiler-based prefetching filtering (CBPF) works best

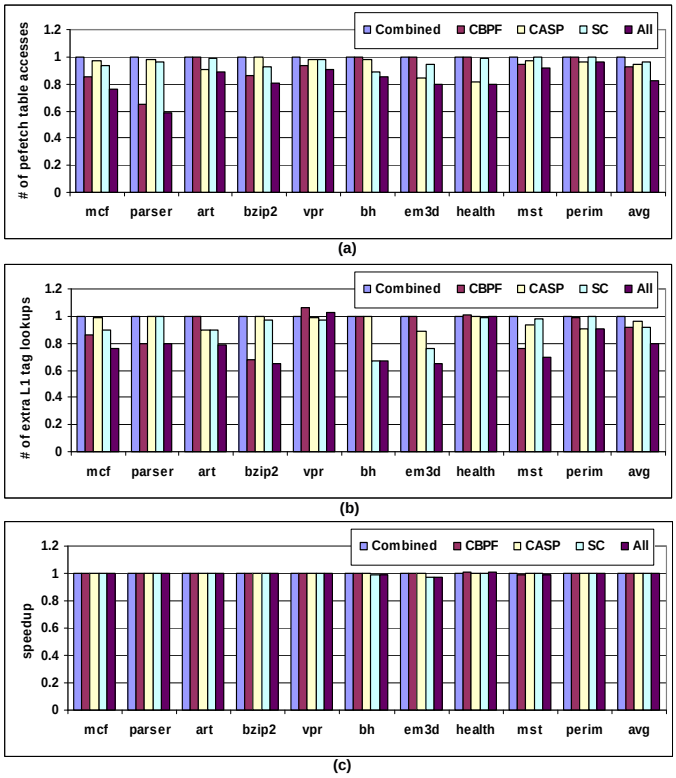

Figure 5: Simulation results for the three compiler-based techniques: (a) normalized number of the prefetch table accesses; (b) normalized number of the L1 tag lookups due to prefetching; and (c) impact on performance.

for *parser*: more than 33% of all the prefetch table accesses are eliminated. On average, CBPF achieves about 7% reduction in prefetch table accesses. The compiler-assisted selective prefetching (CASP) 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 of the ten benchmarks, with an average of 18% across all applications.

Figure 5(b) shows the extra L1 tag lookups due to prefetching. CBPF reduces the tag lookups by more than 8% on average; SC removes about 9%. CASP does not 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 Figure 5(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 Prefetch Filtering Using PFB

Prefetch filtering using PFB will filter out those prefetch requests which would result in a L1 cache hit if issued. We simulated different sizes of PFB to find out the best PFB

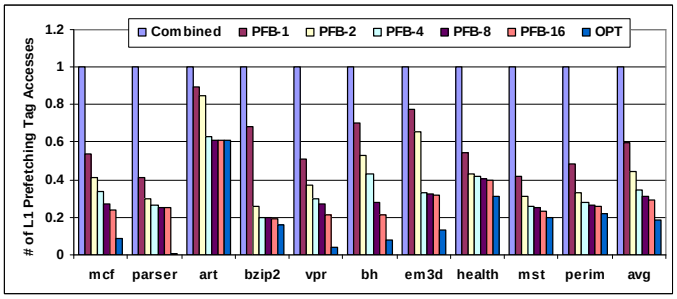

Figure 6: The number of L1 tag lookups due to prefetching after applying the hardware-based prefetch filtering technique with different sizes of PFB.

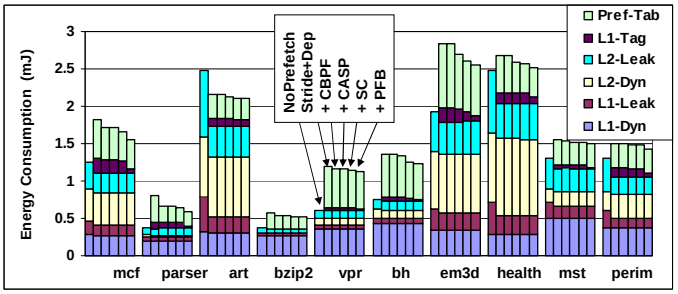

Figure 7: Energy consumption in the memory system after applying different energy-aware prefetching schemes.

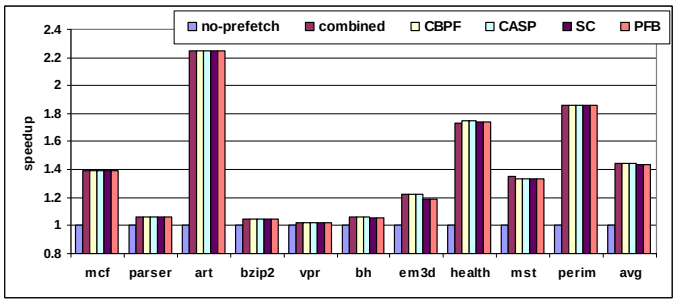

Figure 8: Performance speedup after applying different energy-aware prefetching schemes.

size, considering both performance and energy consumption. Figure 6 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 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 prefetch filtering.

Table 5: The number of PFB mispredictions for 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

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 is shown in Table 5. 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 Energy Savings

We apply the techniques in the following order CBPF, CASP, SC, and PFB. We show the energy savings after each technique is added in Figure 7.

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

The second scheme, compiler-assisted selective prefetching (CASP), 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.

Finally, the prefetch 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%.

Overall, the four power-saving 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).

5.4 Performance Degradation

Figure 8 shows the performance statistics associated with each of the four techniques.

We can see that there is no performance impact except for *em3d* where stride-filtering yields less than 3% speedup degradation. On average, the performance degradation is only 0.4%, while we achieve an average energy saving of

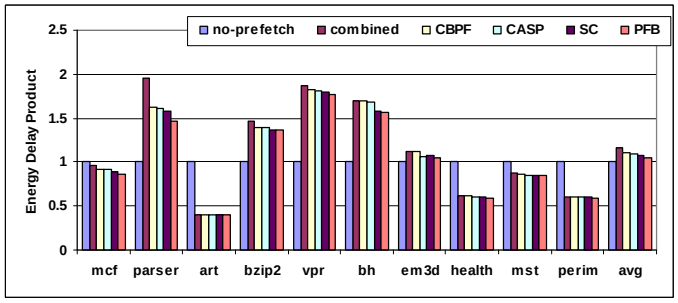

Figure 9: Energy-delay product with different energy-aware prefetching schemes.

11%.

5.5 Energy-Delay Product

Finally, the energy-delay product (EDP) is shown in Figure 9. The EDP is normalized to the case where no prefetching algorithms are used. Compared to the combined stride and pointer prefetching, the EDP improves by almost 25% for *parser*. On average, the four power-aware prefetching techniques combined improve the EDP by about 11%.

6. RELATED WORK

Prefetching has been an active area of research for a long time. Both hardware [27, 3, 24, 25, 9] and software [7, 22, 17, 18] techniques have been proposed for prefetching in recent years. Hardware-based techniques typically uses a hardware table to remember the recent load instructions and set up certain relations between the load instructions. These relations are used to predict future (potential) load addresses used for prefetching. Software prefetching techniques normally need the help of compiler analysis, inserting explicit prefetch instructions into the executables. The prefetch instructions are supported by most contemporary microprocessors [4, 10, 15, 26].

Many recent compiler-assisted prefetching techniques use profiling as an effective tool to recognize data access patterns used for making prefetch decisions. Luk *et al.* [19] uses profiling to analyze executable codes to generate post-link relations which can be used to trigger prefetches. Wu [31] proposes a technique which discovers regular strides for irregular code based on profiling information. Chilimbi *et al.* [8] use profiling to discover dynamic hot data streams which are used for predicting prefetches. Inagaki *et al.* [14] implemented a stride prefetching technique for Java objects. We did not compare to these techniques because our techniques does not needs the help of profiling.

Most of the current prefetching research work focuses on how to improve performance. Related to our paper, a static filter [28] 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.* [29] also propose a compiler-based prefetching filtering technique to reduce traffic resulting from unnecessary prefetches. Their prefetching mechanism is based on [16], which prefetches data blocks from main memory into the L2 cache instead of the typical prefetching from L2 to L1. Although the above two techniques have the potential to reduce prefetching energy overhead, there are no specific discussions or quantitative evaluation of the prefetching energy consumption.

7. CONCLUSION

This paper 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.

8. REFERENCES

[1] The standard performance evaluation corporation, 2000. http://www.spec.org.

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

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

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

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

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

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

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

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

[10] G. Doshi, R. Krishnaiyer, and K. Muthukumar. Optimizing software data prefetches with rotating registers. In *International Conference on Parallel*

Architectures and Compilation Techniques, pages 257–267, Sept. 2001.

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

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

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

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

[15] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, pages 24–36, March/April 1999.

[16] W.-F. Lin, S. K. Reinhardt, and D. Burger. Reducing dram latencies with an integrated memory hierarchy design. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01)*, pages 301–312, Jan. 2001.

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

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

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

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

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

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

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

[24] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*.

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

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

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

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

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

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

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

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