When Prefetching Works, When It Doesn't, and Why

JAEKYU LEE, HYESOON KIM, and RICHARD VUDUC,
Georgia Institute of Technology

In emerging and future high-end processor systems, tolerating increasing cache miss latency and properly managing memory bandwidth will be critical to achieving high performance. Prefetching, in both hardware and software, is among our most important available techniques for doing so; yet, we claim that prefetching is perhaps also the least well-understood.

Thus, the goal of this study is to develop a novel, foundational understanding of both the benefits and limitations of hardware and software prefetching. Our study includes: source code-level analysis, to help in understanding the practical strengths and weaknesses of compiler- and software-based prefetching; a study of the synergistic and antagonistic effects between software and hardware prefetching; and an evaluation of hardware prefetching training policies in the presence of software prefetching requests. We use both simulation and measurement on real systems. We find, for instance, that although there are many opportunities for compilers to prefetch much more aggressively than they currently do, there is also a tangible risk of interference with training existing hardware prefetching mechanisms. Taken together, our observations suggest new research directions for cooperative hardware/software prefetching.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids—*Simulation*; D.3.4 [**Programming Languages**]: Processors—*Optimization*

General Terms: Experimentation, Performance

Additional Key Words and Phrases: Prefetching, cache

ACM Reference Format:
Lee, J., Kim, H., and Vuduc, R. 2012. When prefetching works, when it doesn't, and why. ACM Trans. Archit. Code Optim. 9, 1, Article 2 (March 2012), 29 pages.
DOI = 10.1145/2133382.2133384 http://doi.acm.org/10.1145/2133382.2133384

1. INTRODUCTION

Although a variety of software and hardware prefetching mechanisms for tolerating cache miss latency exist [Callahan et al. 1991; Perez et al. 2004; Vanderwiel and Lilja 2000], only relatively simple software prefetching algorithms have appeared in state-of-the-art compilers like *icc* [ICC] and *gcc* [GCC-4.0]. Therefore, performance-aware programmers often have to insert prefetching intrinsics manually [Intel 2011]. Unfortunately, this state of affairs is problematic for two reasons. First, there are few rigorous guidelines on how best to insert prefetch intrinsics. Secondly, the

We gratefully acknowledge the support of the National Science Foundation (NSF) CCF-0903447, NSF/SRC task 1981, NSF CAREER award 0953100, NSF CAREER award 1139083; the U.S. Department of Energy grant DESC0004915; Sandia National Laboratories; the Defense Advanced Research Projects Agency; Intel Corporation; Microsoft Research; and NVIDIA. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF, SRC, DOE, DARPA, Microsoft, Intel, or NVIDIA.
Author's address: H. Kim; email: hyesoon@cc.gatech.edu.

DOI 10.1145/2133382.2133384 http://doi.acm.org/10.1145/2133382.2133384

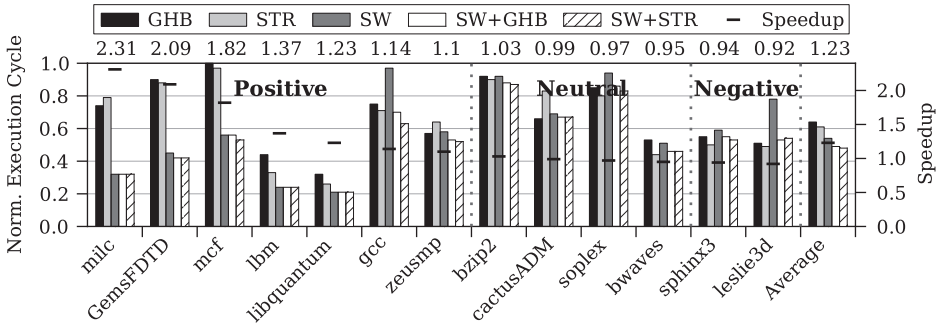


Fig. 1. Summary of the software and hardware prefetching and their interactions. Benchmarks are sorted in decreasing order of the speedup of the best software+hardware prefetching (i.e., the best among SW, SW+GHB, and SW+STR) over the best performing hardware prefetching (i.e., the best among Base, GHB, and STR).

complexity of the interaction between software and hardware prefetching is not well understood. In this study, we try to shed light on these issues. Our goal is to provide guidelines for inserting prefetch intrinsics in the presence of a hardware prefetcher and, more importantly, to identify promising new directions for automated cooperative software/hardware prefetching schemes.

We summarize the complex interactions among software and hardware prefetching schemes in Figure 1.[1] Specifically, we show execution time (left y-axis) of the SPEC CPU 2006 benchmarks (x-axis) under a variety of software and hardware prefetching schemes. For each benchmark, we evaluate three binaries and three different simulated hardware prefetching schemes. All execution times are normalized to the execution time of the corresponding baseline binaries (Base[2]) that have only *compiler-inserted* software prefetchers without any hardware prefetching.[3] The GHB and STR bars (without the SW label) correspond to the baseline binaries running on two hardware configurations: a GHB hardware prefetcher [Nesbit and Smith 2004; Nesbit et al. 2004] and a stream-based prefetcher [Tendler et al. 2002], respectively. Lastly, the "SW+..." binaries include both compiler-inserted prefetches as well as our own manually inserted explicit software prefetch intrinsics. Observe first that baseline binaries do not provide any performance benefits over SW, because the compiler only conservatively inserts software prefetch instructions.

To see the net effects and interactions among the prefetching schemes, we additionally calculate the speedup of the best software+hardware prefetching scheme (i.e., the best among SW, SW+GHB, and SW+STR) over the best performing hardware prefetching (i.e., the best among Base, GHB, and STR). These speedups are shown as labeled horizontal dashes in Figure 1 (scale shown on the right y-axis). Based on these speedups, we group the benchmarks into three categories: positive, neutral, and negative. The milc, GemsFDTD, mcf, lbm, libquantum, and gcc benchmarks show performance improvements from software prefetching in excess of 5% (positive). By contrast, the sphinx3 and leslie3d benchmarks degrade performance more than 5% by using software prefetching (negative). We consider the remaining four benchmarks to exhibit neutral behavior. We can clearly see that the interactions among software and hardware prefetching yield a variety of benchmark-dependent performance behaviors. The goal of our study is to explore this interaction.

[1]Here, we quickly preview these interactions, which Section 4 describes in greater detail.
[2]The performance result of the baseline binaries are presented in Table V.
[3]We use *icc* compilers as default, but we also present the results with *gcc* compilers in Section 5.3.

Although others have studied the benefits and limitations of hardware and software prefetching [Emma et al. 2005; Perez et al. 2004; Vanderwiel and Lilja 2000], we are not aware of any study that explains why some benchmarks show positive/neutral/negative interactions from combined software and hardware prefetching with concrete examples from real benchmarks. We seek to provide such insights in this article by analyzing the source code, data structures, and algorithms used in applications. In particular, we seek answers to the following questions.

(1) What are the limitations and overheads of software prefetching?
(2) What are the limitations and overheads of hardware prefetching?
(3) When is it beneficial to use software and/or hardware prefetching?

Among our numerous findings, we show that when software prefetching targets short array streams, irregular memory address patterns, and L1 cache miss reduction, there is an overall positive impact with code examples. This observation is consistent with previous software prefetching work such as Intel's optimization guidelines [Intel 2011]. However, in this article we demonstrate these observations with actual code examples and empirical data. Furthermore, we also observe that software prefetching can interfere with the training of the hardware prefetcher, resulting in strong negative effects, especially when using software prefetch instructions for a part of streams.

Our study aims to provide a deeper understanding of the interactions between software and hardware prefetching, focusing on schemes currently available in real systems, rather than proposing new schemes or evaluating various hardware/software prefetchers independently. Hence, our hardware prefetching schemes are limited to stream and stride prefetchers, since as far as we know these are the only two implemented commercially today. Similarly, the software prefetching schemes we consider use prefetch intrinsics on top of *gcc* or *icc* compiler-inserted prefetching. Furthermore, we also use both real systems (Intel's Core 2 [Intel 2007] and Nehalem [Intel 2009]) and simulations for our study. Simulations are used for more sophisticated experiments and real systems are used to confirm our findings. Although there are some discrepancies between simulations and real machines, most benchmarks show similar trends, especially for interactions between software and hardware prefetching.

2. BACKGROUND ON SOFTWARE AND HARDWARE PREFETCHING

Table I summarizes common data structures, their corresponding data access patterns, and the hardware/software prefetching mechanisms previously proposed for each. The *Hardware pref.* column suggests possible hardware prefetching mechanisms that are suitable for the respective data structures.[4] The *Software pref.* column includes possible software prefetching mechanisms. In this study, we insert prefetch intrinsics not only for regular structures but also for some irregular data structures, such as Recursive Data Structures (RDS). As Table I indicates, array and some RDS data structure accesses can be easily predicted, and thereby prefetched, by software. However, data structures like hashing are much harder to prefetch effectively. Thus, several hardware and software prefetching mechanisms have been proposed to predict future cache miss addresses of complex data structures. Note that, in this article, we refer to unit-stride cache-line accesses as *streams* and access stride distances greater than two cache lines as *strided*. In our evaluation, we use stream prefetcher, GHB prefetcher, and content-based prefetcher as hardware-based prefetching mechanisms.

[4]Note that there are other hardware prefetchers that aim generic data structures such as dead-block-based prefetching [Lai et al. 2001].

Table I. Data Structures and Corresponding Prefetching

(a is the element to be prefetched, D is prefetch distance)

Structure	Code	Pattern	Size of a	Index	Hardware pref.	Software pref.
Array	a[i]	Stream	< cache block	direct	stream [Jouppi 1990]	a[i+D]
Array	a[i], a[i][const]	Stride	> cache block	direct	stride [Baer and Chen 1991] GHB [Nesbit and Smith 2004]	a[i+D][j]
Array	a[b[i]]	irregular	< cache block	indirect	content-directed (CDP) [Cooksey et al. 2002] markov [Joseph and Grunwald 1997] pointer cache [Collins et al. 2002] jump pointer [Roth and Sohi 1999]	a[b[i+D]]
RDS	a = a→next	stride [Wu 2002] irregular		indirect	CDP, markov pointer cache, jump pointer	a→next→next, Luk-Mowry [Luk and Mowry 1996]
Hash	b[f(i)]→a	irregular		indirect	helper threads, pre-computation [Collins et al. 2001] [Luk 2001; Zilles and Sohi 2001]	Group Prefetching [Chen et al. 2007], SPAID [Lipasti et al. 1995] Push-pull [Yang et al. 2004]
no structure	complex code	regular/irregular			helper threads, pre-computation	

Table II. Different Prefetch Hints in Intrinsic

Hint	Cache Insertion			Remarks
	L1	L2	L3	
_MM_HINT_T0 (T0)	O	O	O	Temporal data with respect to all level caches
_MM_HINT_T1 (T1)	X	O	O	Temporal data with respect to first level cache
_MM_HINT_T2 (T2)	X	X	O	Temporal data with respect to second level cache
_MM_HINT_NTA (NTA)	O	X	X	Non-temporal data with respect to all level caches

Table III. Prefetch Classifications

Classification	Accuracy	Timeliness
Timely	accurate	best prefetching
Late	accurate	demand request arrives before prefetched block is serviced
Early	accurate	demand request arrives after the block is evicted from a cache
Redundant_dc	accurate	the same cache block is in data cache
Redundant_mshr	accurate	the same cache block is in MSHR[5]
Incorrect	inaccurate	block is never used even after the block is evicted from a cache

2.1 Software Prefetch Intrinsics

When manually inserting prefetch requests, we use software prefetch intrinsics that are part of the x86 SSE SIMD extensions [Intel 2008, 2011]. An example of such an instrinsic for x86 systems is *_mm_prefetch(char *ptr, int hint)*, which requires the programmer to specify both a prefetch address as well as a prefetch usage hint. Table II shows the possible hints. One intrinsic is roughly translated to two instructions for direct address prefetches or four instructions for indirect memory addresses. Section 2.4 explains this translation in detail.

2.2 Prefetch Classification

We classify prefetch requests into the six categories listed in Table III. Figure 2 illustrates this classification on a timeline. For example, if a demand access occurs after its corresponding prefetch request but before it is inserted into the cache, we classify the prefetch as "late."

2.3 Software Prefetch Distance

Prefetching is useful only if prefetch requests are sent early enough to fully hide memory latency. The memory address of the cache block to be prefetched in an array data

[5]Miss Status Holding Register [Kroft 1981].

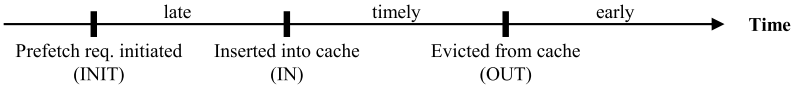


Fig. 2. Prefetch timeliness.

```
LEA &a[i], R0 // calculate addr(a[i])
PREFETCH R0   // prefetch a[i]
```

(a) Direct Indexing Pattern (a[i])

```
LEA &b[i], R0     // calculate addr(b[i])
MOV R1, R0        // LD MEM[R1]: access b[i]
LEA &a[0]+R1, R2 // calculate addr(a[b[i]])
PREFETCH R2       // prefetch a[b[i]]
```

(b) Indirect Indexing Pattern (a[b[i]])

Fig. 3. Assembly code of software prefetch intrinsic.

structure is calculated by adding a constant value D to the array index. This value D is called the *prefetch distance*. We define prefetch distance as the distance ahead of which a prefetch should be requested on a memory address. The prefetch distance D for an array data structure a in Table I can be calculated as follows [Mowry et al. 1992]:

$$D \geq \left\lceil \frac{l}{s} \right\rceil \tag{1}$$

where l is the prefetch latency and s is the length of the shortest path through the loop body. The average memory latency could vary at runtime and the average execution time of one loop iteration can also vary, so the prefetch distance should be chosen in such a way that it is large enough to hide the latency. However, if it is too large, prefetched data could evict useful cache blocks, and the elements in the beginning of the array may not be prefetched, leading to less coverage and more cache misses. Hence the prefetch distance has a significant effect on the overall performance of the prefetcher. We discuss the effect of software prefetch distance in greater detail in Section 5.1.3.

2.4 Direct and Indirect Memory Indexing

There are two types of memory indexing, namely, direct and indirect. Typically, direct memory indexing can be easily prefetched by hardware since the memory addresses show regular stream/stride behavior, but indirect memory indexing requires special hardware prefetching mechanisms. On the contrary, indirect indexing is relatively simpler to compute in software. Thus, we expect software prefetching to be more effective than hardware prefetching for the indirect indexing case.

Even though the address calculation is trivial in software, indirect prefetch requests still have a higher overhead than direct memory access. Figure 3 shows a simplified version of x86 instructions for direct/indirect prefetch requests for a[i]/a[b[i]]. In the case of direct memory indexing, only one prefetch load instruction, *PREFETCH*, is required other than memory index calculation. For indirect memory accesses, however, two loads are required. The first load is usually a regular load because the subsequent instruction is dependent on the first load instruction. Hence, prefetching an indirect memory index usually has a higher overhead than that of direct memory index. Note that the first load instruction can be prefetched. In that case, we need two levels of prefetch streams.

3. POSITIVE AND NEGATIVE IMPACTS OF SOFTWARE PREFETCHING

In this section, we summarize the positive and negative impacts of software prefetching compared to hardware prefetching, as well as both the synergistic and antagonistic effects between the two.

3.1 Benefits of Software Prefetching over Hardware Prefetching

3.1.1 Large Number of Streams (Limited Hardware Resources). The number of streams in the stream prefetcher is limited by hardware resources, such as stream detectors and book-keeping mechanisms. For instance, Intel Pentium-M and Core 2 can issue 8 and 16 stream prefetches, respectively [Intel 2004, 2007]. Unfortunately, in some benchmarks, the number of streams exceeds the hardware's capacity, especially in the case of scientific computing workloads that have multiple streams. For example, `lbm`, a stencil-based computation, a 3D grid data structure and loop references all 27 nearest neighbors for each grid point ($b[i][j][k] = (a[i][j][k] + a[i][j][k+1] + a[i][j][k-1] + a[i][j-1][k] + a[i][j+1][k] + ...)/27$). Software prefetching can insert prefetch requests independently for each stream in `lbm`, whereas it is difficult for hardware prefetchers that are typically easily confused when there are too many streams.

3.1.2 Short Streams. Hardware prefetchers require training time to detect the direction and distance of a stream or stride. Even aggressive hardware prefetchers require at least two cache misses to detect the direction of stream. If the length of stream is extremely short, there will not be enough cache misses to train a hardware prefetcher and load useful cache blocks. For example, `milc` exhibits short stream behavior. In particular, it operates on many 3x3 matrices using 16 bytes per element, so the total size of the matrix is 144 bytes, or just three cache blocks.[6] Therefore, these streams are too short to train the hardware prefetchers. Although there is a new hardware-based prefetching specifically for short streams [Hur and Lin 2006, 2009], no commercial systems presently use it.

3.1.3 Irregular Memory Access. As shown in Table I, we can typically prefetch various kinds of irregular data structures by inserting appropriate prefetch intrinsics, in contrast to hardware prefetchers, which usually require very complex structures. An RDS example from our benchmark suite is the `mcf` benchmark.

3.1.4 Cache Locality Hint. Hardware prefetchers in today's high-performance processors such as Nehalem [Intel 2009] and Barcelona [AMD] place the data in the lower-level cache of the requesting core, whereas most software prefetched data is placed directly into the L1 cache. Lower-level (L2 or L3) prefetching block insertion greatly reduces the higher level (L1) cache pollution, but L2 to L1 latency can degrade performance significantly. In the software prefetching mechanism, there is greater flexibility of placing data in the right cache level. In particular, there is a prefetch hint (Table II) associated with every prefetch request. Current hardware prefetchers either have to apply the same cache insertion policy or they require complex hardware mechanisms to differentiate between the individual prefetches to place data in a smart manner.

3.1.5 Loop Bounds. We can determine loop bounds in array data structures in software with little or no effort. Several methods prevent generating prefetch requests out of array bounds in software such as loop unrolling, software pipelining, and using branch instructions [Chen and Baer 1994; Vanderwiel and Lilja 2000]. The same is not possible in hardware, especially when the hardware prefetcher is aggressive (large prefetch distance and high prefetch degree). However, overly aggressive prefetching results in early or incorrect prefetch requests, in addition to consuming memory bandwidth.

[6]The cache block size is 64B in our all evaluations.

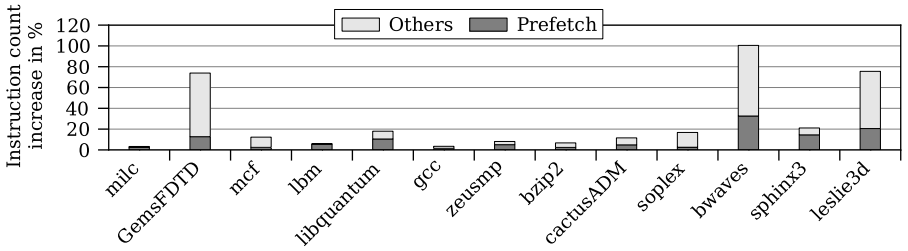


Fig. 4. Instruction count increase.

3.2 Negative Impacts of Software Prefetching

(1) *Increased Instruction Count*. Unlike hardware prefetching, software prefetch instructions consume fetch and execution bandwidth and machine resources, as Figure 4 suggests. For example, the number of instructions in `bwaves` increases by more than 100% due to software prefetches.

(2) *Static Insertion*. The programmer or the compiler determines the data to be prefetched and the corresponding prefetch distance. The decision to prefetch and choice of these parameters are made statically, and therefore cannot adapt to runtime behavioral changes such as varying memory latency, effective cache size, and bandwidth, especially in heterogeneous architectures. Adaptivity is important because it is well-known that the pattern of memory addresses can exhibit phase behavior [Zhang et al. 2006]. In hardware, prefetch distance and prefetching degree are usually fixed as well, although there are some recently proposed feedback-driven adaptive hardware prefetching mechanisms [Ebrahimi et al. 2009; Srinath et al. 2007]. In software, adaptive mechanisms are largely nonexistent.

(3) *Code Structure Change*. When the number of instructions in a loop is extremely small, it can be challenging to insert software prefetching instructions to provide timely prefetching requests because there are not enough computations between prefetching instructions and demand requests. Therefore, code structure changes, such as loop splitting, are required. For some cases, because of excessive increases in the number of instructions, inserting software prefetching becomes difficult.

3.3 Synergistic Effects when Using Software and Hardware Prefetching Together

(1) *Handling Multiple Streams*. Using software and hardware prefetching together can help cover more data streams. For example, a hardware prefetcher might cover regular streams and software prefetching can cover irregular streams.

(2) *Positive Training*. Software prefetch requests can actually help train the hardware prefetcher. If a block prefetched by a software prefetcher is late, then a trained hardware prefetcher can improve prefetch timeliness.

3.4 Antagonistic Effects when Using Software and Hardware Prefetching Together

(1) *Negative Training*. Software prefetch requests can slow down the hardware prefetcher training. If prefetched blocks by software prefetching hide a part of one or more streams, the hardware prefetcher will not be trained properly. Also, software prefetch instructions can trigger overly aggressive hardware prefetches, which results in early requests. Section 5.2.1 shows negative training examples in `milc` and `libquantum`.

(2) *Harmful Software Prefetching*. Hardware prefetchers generally have a lower accuracy than software prefetchers. When software prefetch requests are incorrect or

early, the increased stress on cache and memory bandwidth can further reduce the effectiveness of hardware prefetching.

4. EXPERIMENTAL METHODOLOGY

Given the preceding background on the anticipated effects of and interactions between hardware and software prefetching, the goal of our study is to quantitatively explore these interactions. Our experimental setup is as follows.

4.1 Prefetch Intrinsic Insertion Algorithm

We are interested in the maximum potential of software prefetching. To assess the potential, we insert additional software prefetch intrinsics based on the following methodology.

First, we profile each application to identify prefetch candidates. A prefetch candidate is any load whose L1 misses per thousand instructions (MPKI) exceeds 0.05.[7] Then, for each candidate, we insert a software prefetch. To make our choice of prefetch distance systematic, we compute it as

$$Distance = \frac{K \times L \times IPC_{bench}}{W_{loop}}, \tag{2}$$

where K is a constant factor, L is an average memory latency, IPC_{bench} is the profiled average IPC of each benchmark, and W_{loop} is the average instruction count in one loop iteration. We use $K = 4$ and $L = 300$ for all benchmarks. Importantly, we determine IPC_{bench} and W_{loop} from the profile data, so that the choice of distance is effectively profile-driven. The sensitivity of performance to the prefetch distance is presented in Section 5.1.3. The default prefetch hint is `T0` and the remaining hints are evaluated in Section 5.1.5.

The preceding scheme applies to regular loads. For loads on complex data structures, such as RDS, hash, C++ overloaded operators, and short functions without any loops, we do not use these prefetch intrinsics.

4.2 Simulation Methodology

We use MacSim[8], a trace-driven and cycle-level simulator, to evaluate software and hardware prefetching effects. We faithfully model all components in the memory system. Table IV shows the processor configurations used in our experiments.

We perform experiments on 13 memory-intensive benchmarks from SPEC CPU 2006. We chose benchmarks that achieve at least 100% performance gain with a perfect memory system, that is, one in which all memory accesses hit in the L1 cache. However, we also excluded four additional benchmarks that meet this definition of memory intensive—`omnetpp`, `astar`, `xalancbmk`, and `wrf`—because we were not able to insert any software prefetch intrinsics.

We use Pinpoints [Patil et al. 2004] to select a representative simulation region for each benchmark using the reference input set. Each benchmark is run for roughly 200M x86 instructions.[9] Since different binaries have different numbers of instructions, we ensure that the number of function calls is the same, so that we simulate the same section of the code across different binaries.

[7]We chose 0.05 through experimentation. The sensitivity results are not included due to the space limitations.
[8]http://code.google.com/p/macsim/
[9]Due to extra software prefetching instructions, the SW binaries have more than 200M instructions.

Table IV. Processor Configurations (base:baseline, less: less aggressive, agg: more aggressive)

Execution Core	less	2-wide 10-stage Pipeline, 36-entry ROB, 1GHz in-order
	base	4-wide 15-stage Pipeline, 256-entry ROB, 2GHz out-of-order
	agg	6-wide 18-stage Pipeline, 1024-entry ROB, 4GHz out-of-order
Front-end	less	I-cache: 16KB, 4-way, 2-cycle, 2/2 write/read ports
	base	I-cache: 32KB, 4-way, 3-cycle, 4/4 write/read ports
	agg	I-cache: 64KB, 4-way, 4-cycle, 6/6 write/read ports
On-chip Caches	less	L1 D-cache: 16KB, 4-way, 2-cycle, 2/2 write/read ports L2 Unified-cache: 128KB, 8-way, 6-cycle, 1/1 write/read port, 32 MSHRs
	base	L1 D-cache: 32KB, 4-way, 3-cycle, 4/4 write/read ports L2 Unified-cache: 1MB, 8-way, 10-cycle, 1/1 write/read port, 128 MSHRs
	agg	L1 D-cache: 64KB, 4-way, 4-cycle, 6/6 write/read port L2 Unified-cache: 8MB, 8-way, 14-cycle, 1/1 write/read port, 256 MSHRs
		LRU replacement, 64B line
Buses and Memory	less	DDR2-533, 3-3-3, Bandwidth 2.1GB/s
	base	DDR2 800, 6-6-6, Bandwidth 6.4GB/s
	agg	DDR3-1600, 9-9-9, Bandwidth 25.6GB/s
Stream Prefetcher [Tendler et al. 2002; Srinath et al. 2007] 16 streams, Prefetch Degree 2, Prefetch Dist. 16		
GHB Prefetcher [Nesbit and Smith 2004; Nesbit et al. 2004] 1024 entries, Prefetch Degree 16		

Table V. Characteristics of the Simulated Benchmarks

	BASE						SW		
	Lang/Type	# Inst.	L1 MPKI	L2 MPKI	Base IPC	PerfMem IPC Δ (%)	# Inst.	Static # Pref.	Dyn. # Pref.
bzip2	c/int	203M	13.17	2.02	0.61	110	217M	19	4.91M
gcc	c/int	207M	43.24	15.31	0.26	369	214M	7	3.17M
bwaves	F/fp	202M	28.77	26.5	0.46	243	204M	12	65.91M
mcf	c/int	200M	83.88	29.69	0.12	758	225	6	4.94M
milc	c/fp	210M	35.59	17.40	0.33	482	217M	36	5.2M
zeusmp	F/fp	216M	11.48	6.09	0.51	212	233M	72	10.93M
cactusADM	FPP/fp	201M	9.16	5.97	0.51	204	224M	113	9.78M
leslie3d	F/fp	203M	39.09	23.23	0.27	389	356M	152	41.81M
soplex	c++/fp	257M	14.55	7.84	0.39	223	300M	52	6.61M
GemsFDTD	F90PP/fp	216M	65.10	20.00	0.22	700	376M	97	27.55M
libquantum	c/int	200M	70.10	26.28	0.27	700	236M	4	27.55M
lbm	c/fp	200M	57.89	29.40	0.22	555	212M	19	11.19M
sphinx3	c/fp	201M	68.15	11.63	0.62	292	243M	31	29.16M

PerfMem: the performance of benchmarks with zero-cycle memory latency
Static Pref and Dynamic Pref: the number of prefetch instructions that are inserted by a programmer

Table V shows the characteristics of the evaluated benchmarks (`Base` binaries) on the baseline processor. The remaining SPEC CPU 2006 benchmarks are presented in Table VI. All benchmarks are compiled using *icc* 11.0.081 and *ifort* 11.0.081. Results for the GCC compiler appear in Section 5.3.

5. EVALUATIONS: BASIC OBSERVATIONS ABOUT PREFETCHING

We first make a number of basic observations about software and hardware prefetching and their interactions. Our goal is to evaluate the hypotheses outlined in Section 3. Note that we consider a variety of factors; for a quick summary of the key findings, refer to Section 5.9.

5.1 Overhead and Limitations of Software Prefetching

5.1.1 Instruction Overhead. Instruction overhead is among the major negative impacts of software prefetching (Section 3.2). Figure 4 shows how the instruction count

Table VI. Characteristics of the Remaining SPEC CPU 2006 Benchmarks

Bench	perlbench	gobmk	hmmer	sjeng	h264ref	omnetpp	astar	xalancbmk
Lang/Type	c/int	c/int	c/int	c/int	c/int	c++/int	c++/int	c++/int
Base IPC	1.11	0.79	1.21	1.07	1.53	0.27	0.36	0.76
PerfMem Δ (%)	80	23	64	28	74	592	405	166
Bench	gamess	gromacs	namd	dealII	povray	calculix	tonto	wrf
Lang/Type	FPP/fp	c/fp	c++/fp	c++/fp	c++/fp	F/fp	F90PP/fp	F90PP/fp
Base IPC	1.84	1.89	1.61	1.61	1.16	1.32	2.07	0.41
PerfMem Δ (%)	16	39	29	47	29	71	16	656

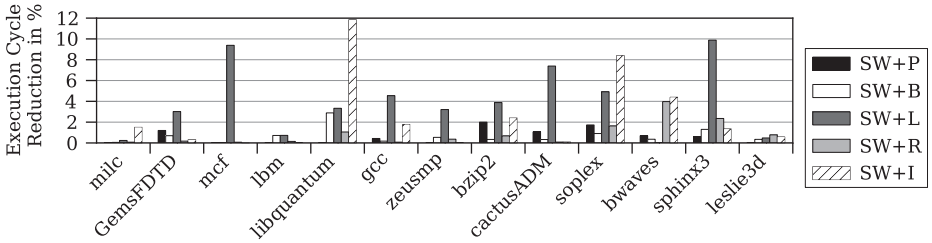


Fig. 5. Ideal experiments (SW+P: no pollution, SW+B: no bandwidth, SW+L: no latency, SW+R: eliminate redundant prefetch overhead, SW+I: eliminate inst. overhead).

increases in SW binaries. For GemsFDTD, bwaves, and leslie3d, the total number of instructions increases by more than 50% (more than 100% for bwaves). These extra instructions come not just from prefetch instructions but also from regular instructions due to handling of indirect memory accesses and index calculations. Surprisingly, although GemsFDTD increases instructions by more than 50%, it is still in the positive (improved performance) group because the benefits outweigh this instruction overhead.

5.1.2 Software Prefetching Overhead. To measure the overhead of software prefetching, we eliminate the overhead of cache pollution (SW+P), bandwidth consumption (SW+B, between core and memory), memory access latency (SW+L), redundant prefetch overhead (SW+R), and instruction overhead (SW+I). Figure 5 summarizes the results.

Cache pollution would be caused by early or incorrect prefetches. However, there are not many of either in our experiments. Thus, the effects of cache pollution are small, as shown by the relatively small values of SW+P in essentially all cases. Similarly, the mostly small values of SW+B show that current machines provide enough bandwidth for single-thread applications. The mostly larger values of SW+L shows that software prefetching is not completely hiding memory latency. The values of SW+R show that, surprisingly, the negative effect of redundant prefetch instructions is generally negligible even though a huge number of redundant prefetches exist. Finally, SW+I experiments show that even though the number of instructions increased significantly for GemsFDTD, bwaves, and leslie3d (see Section 5.1.1), their actual time overhead is not high.

5.1.3 The Effect of Prefetch Distance. Figure 6 shows the sensitivity of performance to prefetch distance. The x-axis show the prefetch distance relative to the base prefetch distance. The base prefetch distance is calculated using Equation (1). The unit of prefetch distance is a cache block.

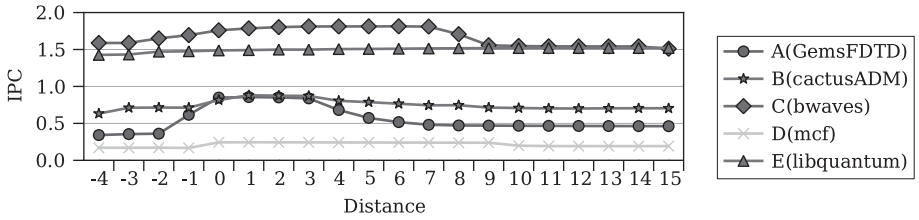


Fig. 6. Performance variations by the prefetch distance.

Table VII. Categorization of Benchmark Groups by the Distance Sensitivity

Group	Optimal Zone	Perf. Delta	Benchmarks
A	narrow	high	GemsFDTD, lbm, milc
B	narrow	low	cactusADM
C	wide	high	bwaves
D	wide	low	mcf
E	insensitive		soplex, sphinx3, libquantum, gcc, bzip2, leslie3d, zeusmp

Based on the shape of the optimal distance zone and the performance variance, we categorize all benchmarks into five groups. Table VII shows all groups and their descriptions. Figure 6 shows the performance variations from each group. Most benchmarks from the neutral and negative groups in Figure 1 are insensitive to the prefetch distance (Group E). The two exceptions are `cactusADM` and `bwaves`. These benchmarks reach the optimal distance from the beginning (-4 in Figure 6) and have wider optimal distance zones.

5.1.4 Static Distance vs. Machine Configuration. One limitation of using static prefetch distances is that the optimal distance might vary by machine. This effect will be exacerbated in future heterogeneous multicore systems, since the programmer is likely not to know the core/system configuration in advance. To understand the sensitivity of optimal prefetch distance to machine configuration, we compare the best performing static distance for three different machine configurations (base, less-aggressive, and aggressive processors in Table IV). The left three figures in Figure 7 show the performance for the best prefetch distance for each machine configuration. The right three figures in Figure 7 show the performance difference between the the best distance in the baseline and the best distance for each machine's configuration. For example, the best distances for `lbm` are 2, 0, and 8 for less-aggressive, base, and aggressive processors, respectively. We measure the performance difference between that of distance 2 and 0 in the less-aggressive processor and the difference between that of distance 8 and 0 in the aggressive processor. Despite the variations in the optimal distance, the performance difference is generally not high except for `lbm`. This observation is not surprising given that most benchmarks are classified as Group E (insensitive) in Table VII. As for `lbm`, we analyze it in more detail in Section 6.1.4. We conclude that the static prefetch distance variance does not impact performance significantly even if the machine configuration is changed at runtime.

5.1.5 Cache-Level Insertion Policy. In general, an out-of-order processor can easily tolerate L1 misses, so hardware prefetchers usually insert prefetched blocks into the last level cache to reduce the chance of polluting the L1 cache. However, when the L1 cache

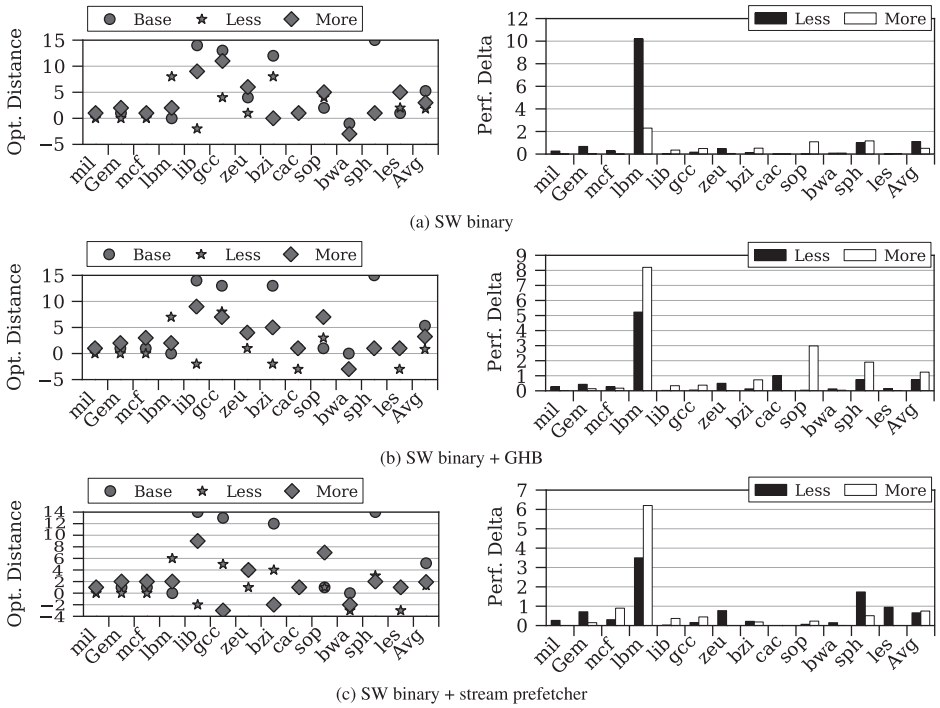


Fig. 7. Best performing distance variances with different core specifications.

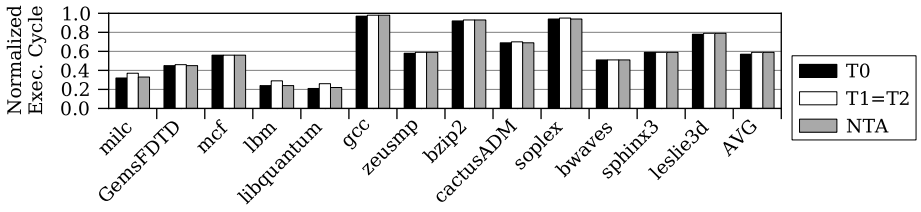


Fig. 8. Different prefetch hint results.

miss rate is higher than what a processor can tolerate, even an L1 cache miss can degrade performance. Since software prefetching is generally accurate, we can safely insert prefetched blocks into the L1 cache directly without the risk of polluting the L1 cache.

In all our experiments, we use the `T0` cache-level placement hint, as shown in Table II. Note that `T1` and `T2` behave the same in our two-level cache hierarchy. Figure 8 shows performance results when varying the hint (level). The benefit of `T0` over `T1` (`T2`) mainly comes from hiding L1 cache misses by inserting prefetched blocks into the L1 cache. In the evaluated benchmarks, `libquantum` represents this behavior. With software prefetching, the L1 cache miss rate is reduced from 10.4% to 0.01% (Figure 26), which results in a significant performance improvement. Where we might expect the `T1` and `T2` hints to work well is in the case of streaming applications that have little data reuse. Since our benchmarks do not have such applications, we find that `T0` always performs better than other hints in all evaluated benchmarks.

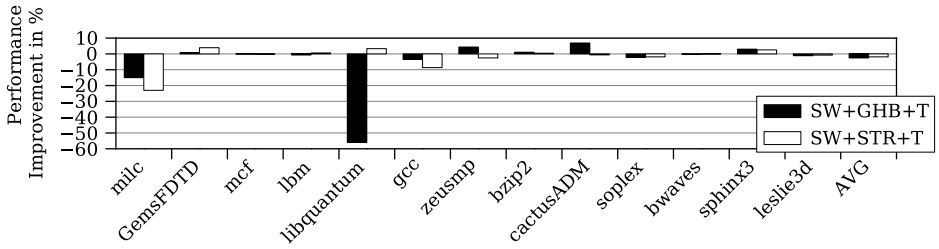


Fig. 9. Hardware prefetcher trained by software prefetch requests.

5.2 Effect of Using Hardware and Software Prefetching Together

5.2.1 Hardware Prefetcher Training Effects. To evaluate how software and hardware prefetching interact with each other, we consider two different hardware prefetcher training policies.

(1) `NT` (`SW+GHB`, `SW+STR`). The hardware prefetcher's training algorithm ignores software prefetch requests. That is, the hardware prefetcher is only trained by demand cache misses.
(2) `Train` (`SW+GHB+T`, `SW+STR+T`). The hardware prefetcher's training algorithm includes software prefetch requests, in addition to the usual demand misses.

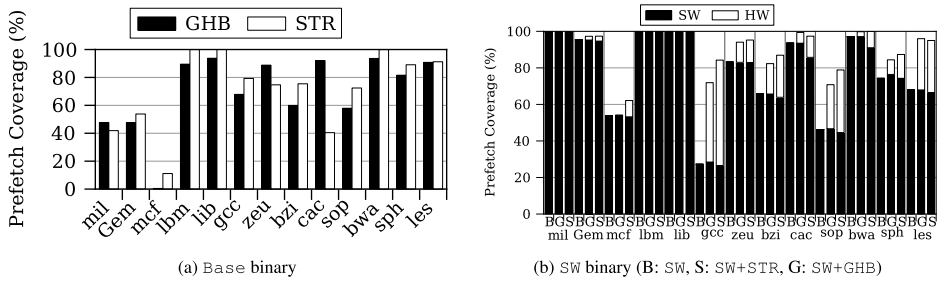
Figure 9 shows the performance improvement of the `Train` policy over `NT` for both GHB and STR. For both GHB and STR, the `sphinx3` benchmark shows a small benefit from training with software prefetches. By contrast, `milc`, `gcc`, and `soplex` exhibit negative effects for both GHB and STR. The `mcf` and `bwaves` benchmarks are largely unaffected by training policy. For the remaining benchmarks, the effect of the training policy is prefetcher-specific, for instance, positive with GHB and negative with STR.

The benefit could be up to 3–5% and these benefits come from training hardware prefetching early, that is, by reducing late prefetching. However, the negative impact can be severe: −55.9% for `libquantum`, and −22.5% for `milc`. The `milc` benchmark has short streams, so software prefetching can trigger unnecessary hardware prefetches. There are two reasons for the performance degradation in `libquantum`. The first reason is because of severe L1 miss penalties. Software prefetches are mainly inserted into the L1 and L2 caches. However, if hardware prefetching is trained by this software prefetching, prefetched blocks are inserted into only L2 rather than L1, thereby reducing the L1 prefetching benefits. The second reason is overly aggressive prefetches. To provide timeliness, software and hardware prefetches have prefetch distances independently. However, when software prefetch instructions train a hardware prefetcher, then hardware prefetch requests can be too aggressive (on top of software prefetch distance, now hardware prefetching adds its own prefetch distance), resulting in many early prefetches.

From these experiments, we can conclude that the negative impact can reduce performance degradation significantly. Thus, although there are some positive benefits, it is generally better not to train hardware prefetching with software prefetching requests.

5.2.2 Prefetch Coverage. We consider the prefetch coverage—or, the ratio of useful prefetches to total L2 misses—in Figure 10(a), for `GHB` and `STR`. Figure 10(b) shows the prefetch coverage of `SW+GHB` and `SW+STR`. Additional useful hardware prefetches are indicated as `HW` and are shown on top of the `SW` bars.

Generally, the coverage of software prefetching is higher than that of hardware prefetchers, but the software coverage of benchmarks in that neutral and negative



(a) `Base` binary (b) `SW` binary (B: `SW`, S: `SW+STR`, G: `SW+GHB`)

Fig. 10. Prefetch coverage.

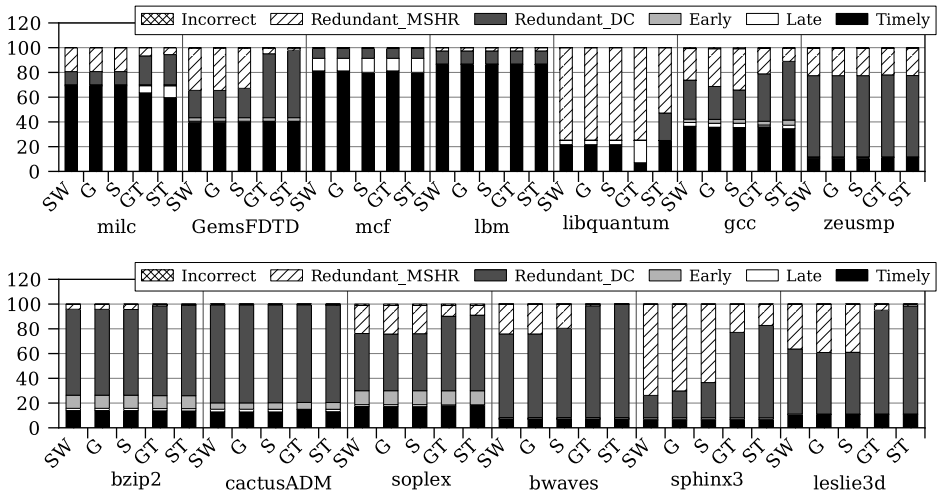


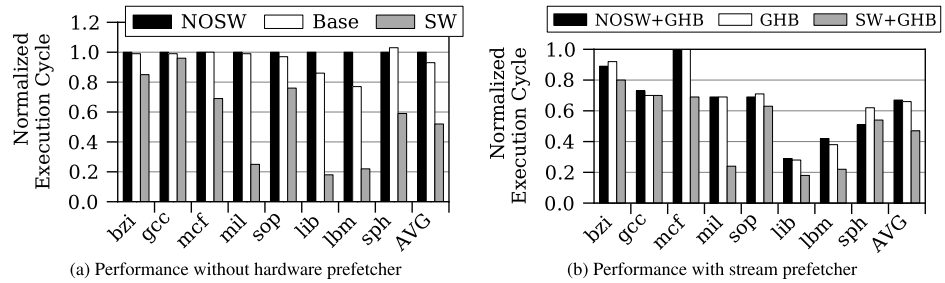
Fig. 11. Software prefetch classifications (G:SW+GHB, S:SW+STR, GT:SW+GHB+T, ST:SW+STR+T).

groups is lower than that of the hardware prefetchers. Thus, we can conclude that less coverage is the main reason for performance loss in the neutral and negative groups.

5.2.3 Prefetching Classification. For each benchmark, we classify the prefetches into the categories of timely, late, early, etc., as summarized in Table III. Figure 11 summarizes the results. Only `bzip2` and `soplex` show more than 10% early prefetches. A prefetch cache could mitigate the cache pollution effect. For `mcf`, 10% of prefetches are late, due to a dependent load instruction that the prefetch intrinsic generates. By eliminating the latency overhead (Figure 5 `SW+L` case), the performance of `mcf` is improved by 9%. Surprisingly, even though a significant number of redundant prefetches exists in many benchmarks, there is little negative effect on the performance (Figure 5 `SW+R` case).

5.3 Using the GCC Compiler

Except in this section, we use the *icc* compiler for our evaluation because *gcc* (more specifically *gfortran*) does not support intrinsics in Fortran. To understand the *gcc* compiler's effect, we insert intrinisics after compiling the code with *gcc* in these experiments. The simulated regions are exactly the same as those of *icc*, but we exclude all Fortran benchmarks since the *gfortran* compiler does not support prefetch intrinsics. We use *gcc 4.4* in our evaluations.



(a) Performance without hardware prefetcher (b) Performance with stream prefetcher

Fig. 12. Baseline experiments with gcc compiler.

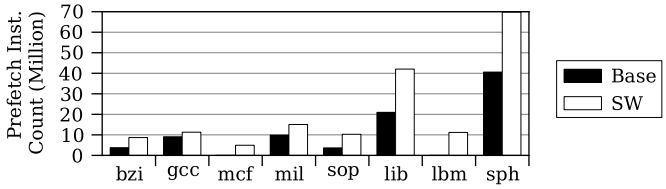


Fig. 13. Instruction counts with gcc compiler.

Table VIII. Specification of Two Intel Processors

Machine	Specification
Nehalem	Dual Intel Xeon E5520 (2.27GHz, 8MB L3 Cache, SMT Disabled), 24GB DDR3 Memory
Core 2	Dual Intel Xeon E5420 (2.50GHz, 12MB L2 Cache), 8GB DDR2 Memory

We summarize the results of the *gcc* compiler experiments in Figure 12. Figure 13 shows the number of prefetch instructions in each binary. The *gcc* compiler inserts more software prefetches than *icc*, except for `mcf` and `lbm`. Recall that *icc* does not insert any prefetch instructions. However, *gcc* still cannot cover all cache misses. Thus, the performance of the `Base` binaries compiled with *gcc* lies between the performance of `NOSW` (prefetch flag is disabled) and `SW`, whereas that of *icc* is the same as `NOSW`.

5.4 Real System Experiments

In addition to the simulation results, we also measured the effect of software prefetching on two recent Intel processors, `Core 2` and `Nehalem`. Table VIII summarizes the specifications of these processors. In our real systems experiments, we run the entire reference input sets, not just the simpointed section.[10] Table IX shows that both simulated regions and the entire execution have a similar portion of prefetch instructions. Except for `milc`, `gcc`, and `zeusmp`, the simulated region of other benchmarks accurately represents the entire code. Note that our simulated system is closer in its configuration to the `Core 2`.

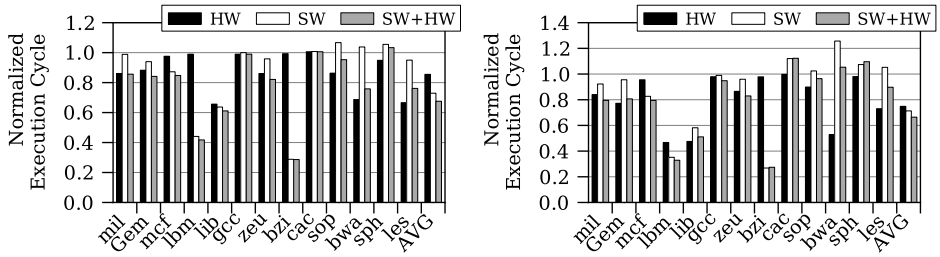
Figure 14 shows results. In real system experiments, we also see that software prefetching reduced the performance of `cactusADM`, `soplex`, `bwaves`, `sphinx3`, and

[10]This is one of the reasons the simulated and measured performance differ.

Table IX.

Comparison between entire and simulated codes. Total ratio is #prefetch inst/#total inst and Sim ratio is the fraction in the simulated regions

Benchmark	Total (Mem) Inst.	Total ratio	Sim ratio	Benchmark	Total (Mem) Inst.	Total ratio	Sim ratio
milc	1.66T (728B)	0.48%	2.40%	GemsFDTD	1.69T (860B)	5.06%	7.32%
mcf	906B (463B)	2.24%	2.19%	lbm	1.45T (609B)	5.1%	5.27%
libquantum	2.41T (799B)	10.7%	8.9%	gcc	157B (81B)	0.02%	1.48%
zeusmp	1.94T (675B)	0.38%	4.68%	bzip2	583B (267B)	1.8%	2.26%
cactusADM	2.78T (1.6T)	4.45%	4.36%	soplex	1.73T (972B)	2.2%	2.2%
bwaves	2.59T (1.8T)	14%	16.26%	sphinx3	4.34T (1.73T)	11.8%	11.97%
leslie3d	2.14T (1.1T)	10.74%	11.71%				



(a) Intel Core 2 (b) Intel Nehalem

Fig. 14. Real system experiment results.

`leslie3d` by a small amount in the real system measurements, even though it always shows benefits in our simulations. However, all benchmarks in the positive group also show benefits from using software prefetches. Compared with the `Core 2`, the hardware prefetchers of `Nehalem` have improved. Also, due to the multiple cache levels of prefetchers in `Nehalem` (L3 to L2 and L2 to L1), the benefits of software prefetching are reduced (In Section 3.1.4, we discussed the software prefetching benefits of inserting prefetched blocks directly into the L1 cache).

5.5 Profile-Guided Optimization (PGO)

We also examine the effect of profile-guided optimization. Note that we perform experiments using both *i*cc and *g*cc, but we do not include the results from *i*cc because it does not insert any software prefetches. We use gcc 4.4 with `fprofile-generate` and `fprofile-use` flags. Figure 15 shows the results of PGO.

Generally, the performance of using PGO lies between that of `Base` and `SW`. However, performance benefits come from other optimizations, not only software prefetching. Figure 15(c) and (d) show the number of total and prefetch instructions. The number of prefetch instructions are rather decreased in the PGO-compiled binaries. Also, the instruction count of the `PGO` binaries is less than that of the `Base` binaries in all cases.

5.6 Hardware Prefetcher for Short Streams

One weakness of hardware prefetching is the difficulty of exploiting short streams. To tackle this problem, some have proposed the so-called ASD hardware prefetcher [Hur and Lin 2006, 2009]. (ASD is currently an idea with no known implementations, as far as we know.) We evaluate ASD along with software and hardware prefetching, with the results summarized in Figure 16.

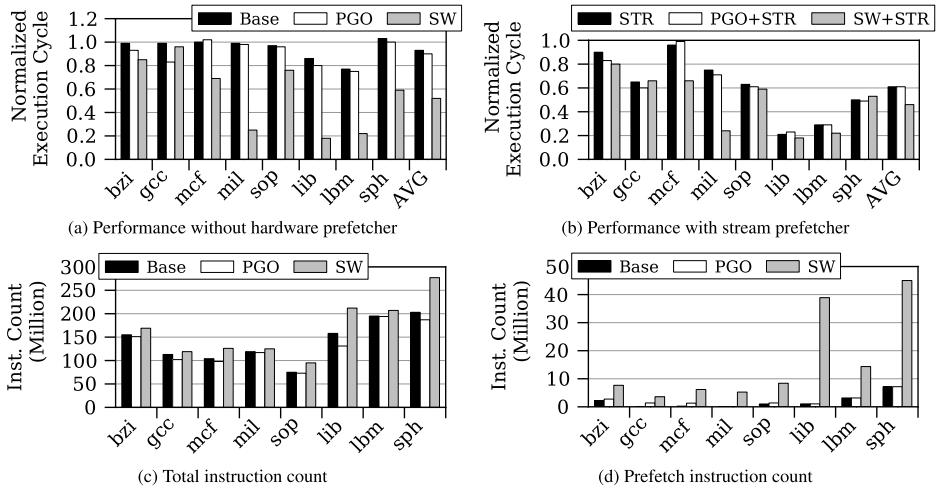


Fig. 15. Profile-guided optimization results.

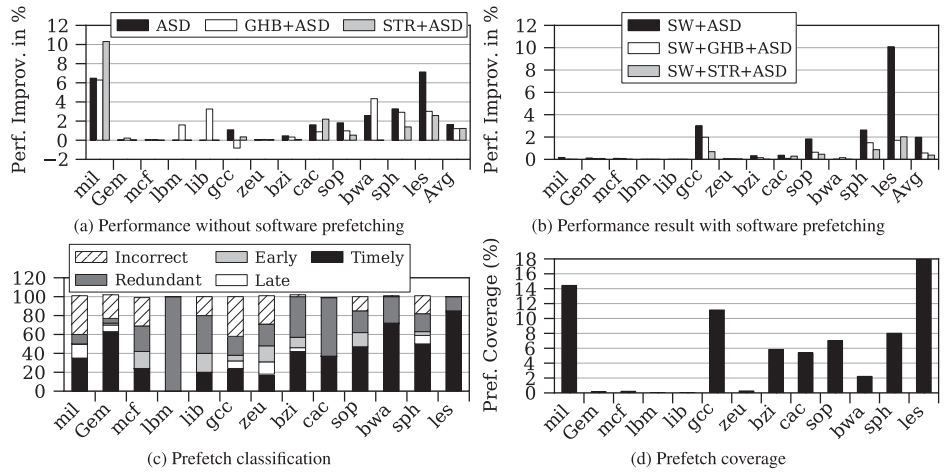


Fig. 16. ASD hardware prefetcher results.

The `milc` benchmark exhibits short stream behavior. On average, ASD provides a 7% performance benefit on `milc` without software prefetching. However, software prefetching itself can still provide more than a 2x speedup. Although ASD can predict short-length streams, it still requires observing the first instances of a memory stream. ASD provides benefits over the other two evaluated hardware prefetchers, but the overall benefit is less than 3%. The `gcc`, `sph`, and `les` benchmarks perform better than the baseline when combining ASD, software prefetching, and GHB/STR hardware prefetching, but only by 2% or less. Thus, we conclude that software prefetching is much more effective for prefetching short streams than ASD.

5.7 Content Directed Prefetching

We also evaluated content directed prefetching (CDP) schemes, which target linked and other irregular data structures [Al-Sukhni et al. 2003; Cooksey et al. 2002; Ebrahimi et al. 2009]. Figure 17 shows results. We focus on `gcc` and `mcf` results since

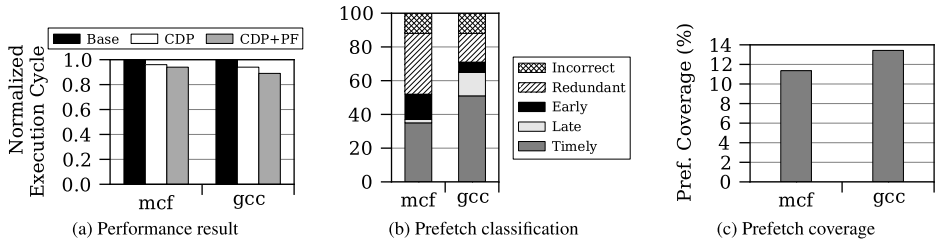


Fig. 17. CDP hardware prefetcher results.

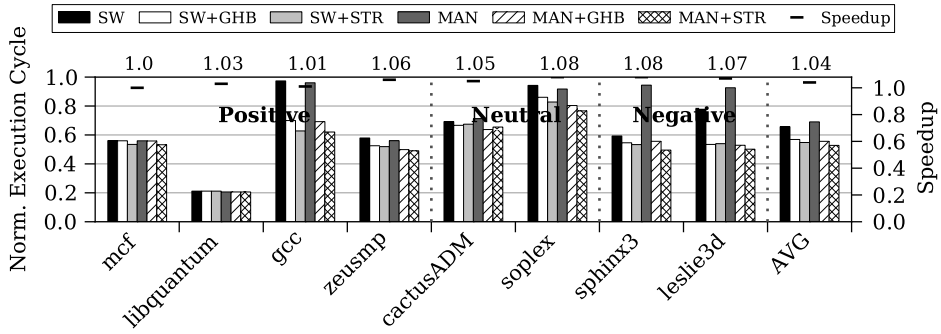


Fig. 18. Manual optimization results.

these are the only two benchmarks affected, positively or negatively, by CDP. CDP provides 3.9% and 6.5% improvement on `mcf` and `gcc`, respectively. We also evaluate CDP with a separate prefetch cache (CDP+PF), which prevents cache pollution, a well-known limitation of CDP. We find additional 2% and 5% benefits. However, when compared with software prefetching's 78% improvement, we conclude that software prefetching is more effective for irregular data structures.

5.8 Manual Tuning

Since we used the prefetch intrinsic insertion algorithm in Section 4.1 for all benchmarks, we cannot be sure that software prefetching is being used to its full potential. Thus, we also manually and exhaustively tune the benchmarks. In particular, we remove useless prefetches, adjust the prefetch distance of each software prefetch, and apply additional overhead-reduction techniques, such as loop unrolling and prologue/epilogue transformations for each benchmark. We exclude `milc`, `GemsFDTD`, `lbm`, `bwaves`, and `bzip2` because they either are already highly optimized or require significant code structure changes.

Figure 18 shows the manual tuning results, where `MAN` indicates our manually tuned binaries. The speedup shown is for `MAN` over `SW`. We vary the distance of all prefetch intrinsics individually and select the distance for each prefetch intrinsic that yields the best performance. The speedup in the neutral and negative groups is much higher than the speedup in the positive group, which implies that it is more effective to remove negative intrinsics manually. However, there are no significant performance changes between manual tuning and the algorithm in Section 4.1, especially in the positive group, which shows that the algorithm is good to use as a guideline for inserting software prefetch intrinisics.

Table X. Prefetching Summary of Benchmark Group

Benchmark		Data structure	GHB+T	STR+T	Why SW is better	Why SW is worse	Best
POS	433.milc	short array	–	–	short array		SW+GHB
	459.GemsFDTD	indirect, stride	=	+	indirect access		SW+STR+T
	429.mcf	RDS, AoP	=	=	irregular (AoP)		SW+STR
	470.lbm	stride	=	=	reduction in L1 misses		SW+STR+T
	462.libquantum	stride	–	=	reduction in L1 misses		SW+STR+T
	403.gcc	RDS, indirect	-	–	indirect access		SW+STR
	434.zeusmp	stride	+	-	reduction in L1 misses		SW+GHB+T
NEU	434.bzip2	indirect, hash	+	+			SW+GHB+T
	436.cactusADM	stride	+	-			SW+GHB+T
	436.soplex	indirect	-	-			STR
	410.bwaves	stream	=	=			STR
NEG	482.sphinx3	stride	+	+		loop is too short	SW+STR
	437.leslie3d	stream	-	-		loop is too short	SW+STR

+: positive, -: negative, Best: best performing case among all evaluated cases

5.9 Summary of New Findings

Though there are numerous results, here we summarize our key findings.

(1) Hardware prefetchers can under-exploit even regular access patterns, such as streams. Short streams are particularly problematic, and software prefetching is frequently more effective in such cases.
(2) The software prefetching distance is relatively insensitive to the hardware configuration. The prefetch distance does need to be set carefully, but as long as the prefetch distance is greater than the minimum distance, most applications will not be sensitive to the prefetch distance.
(3) Although most L1 cache misses can be tolerated through out-of-order execution, when the L1 cache miss rate is much higher than 20%, reducing L1 cache misses by prefetching into the L1 cache can be effective.
(4) The overhead of useless prefetching instructions is not very significant. Most applications that have prefetching instructions are typically limited by memory operations. Thus, having some extra instructions (software prefetching instructions) does not increase execution time by much.
(5) Software prefetching can be used to train a hardware prefetcher and thereby yield some performance improvement. However, it can also degrade performance severely, and therefore must be done judiciously if at all.

6. CASE STUDIES: SOURCE CODE ANALYSIS OF INDIVIDUAL BENCHMARKS

In this section, we explain the reasons for the observed software prefetching behaviors using source code analysis. The key observed behaviors appear in Table X.

6.1 Positive Group

6.1.1 433.milc. Most cache misses in `milc` occur during accesses to small arrays, in such functions as `mult_su3_na`, which is shown in Figure 19. These functions operate on 3x3 matrices. As discussed in Section 3.1.2, these matrix accesses correspond to short streams, which are too short to train the hardware prefetchers. By contrast, software prefetching can effectively prefetch these arrays, provided the prefetch requests are inserted appropriately near the call site. When doing so, we observe a 2.3x speedup compared with the best hardware-only prefetching scheme.

Figure 20 shows the number of L2 cache misses for `Base`, `GHB`, `STR`, and `SW` binaries from the eight source code lines that generate the most cache misses in `Base`. Since

```
// path_product.c (parent function)
49: for (j = 1; j < length; j++) {
50:   ..
54:   FORALLSITES(i,s) {
        PREFETCH(gen_pt[0][i+N]->e[0][0].real);
        PREFETCH(tempmat2t[i+N]->e[0][0].real);
55:     mult_su3_nn((su3_matrix *)(gen_pt[0][i]), &(s->link[dir[j]], &(tempmat2t[i]));
56:   } END_LOOP
57: }

// m_mat_nn.c (target function)
70: void mult_su3_nn(su3_matrix *a, su3_matrix *b, su3_matrix *c) {
71:   int i,j;
72:   register double t,ar,ai,br,bi,cr,ci;
73:   for(i=0;i<3;i++)for(j=0;j<3;j++) {
75:     ar=a->e[i][0].real; ai=a->e[i][0].imag;
76:     br=b->e[0][j].real; bi=b->e[0][j].imag;
        ...
80:     ar=a->e[i][1].real; ai=a->e[i][1].imag;
81:     br=b->e[1][j].real; bi=b->e[1][j].imag;
        ...
90:     c->e[i][j].real=cr;
91:     c->e[i][j].imag=ci;
92:   }
93: }
```

Fig. 19. Code example:433.milc.

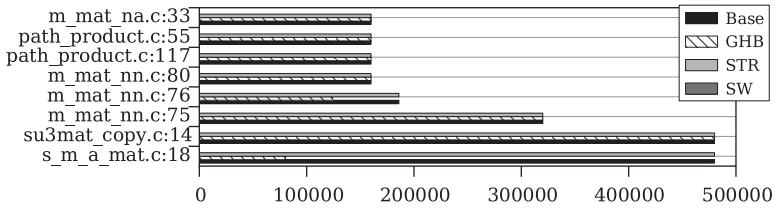


Fig. 20. L2 cache misses in 433.milc.

the Base and SW binaries have different PC addresses, we match each individual PC address with the source line. As expected, hardware prefetchers do not reduce cache misses, whereas software prefetching can eliminate almost all misses. Since software prefetching can cover all misses, training the hardware prefetchers will increase memory traffic with useless prefetches, thereby increasing the execution time by 15% and 23% with GHB and stream prefetcher, respectively.

6.1.2 459.GemsFDTD. Most delinquent loads in GemsFDTD come from complex indirect memory accesses. For instance, consider Figure 21. The variable m is used as an (indirect) index. Since this indirect indexing is very irregular from the hardware perspective, the hardware prefetchers do not prove to be very useful, improving performance by only 12% improvement (stream prefetcher). On the contrary, this complex indirect indexing can be handled very effectively in software prefetching. Doing so results in more than a 50% execution time reduction compared with Base.

6.1.3 429.mcf. The mcf benchmark is well-known as a memory intensive benchmark due to its pointer chasing behavior. It also uses an array of pointers (AoP) data structure. Since AoP exhibits irregular behavior, hardware prefetchers typically do not perform well on such structures. However, the arc structure can be prefetched in software, as shown in Figure 22. A suitable software prefetch can calculate the next address by using the next array index, whereas neither GHB nor stream prefetchers can prefetch them. SW+STR provides a 180% improvement over STR. The stream prefetcher

```
// NFT.F90
1068: do n=1,Nangle
1074:   do j=Y1,Yn
1075:     do k=Z1,Zn-1
1077:       m = nt + nezX1(j-Y1+1,k-Z1+1,n)
            PREFETCH(Fth(n,nt+nexX1(j-Y1+1,k-Z1+8,n)
            PREFETCH(tauezX1(j-Y1+1,k-Z1+8,n)
1078:       ehtmp = Hy(X1-1,j,k)*ezth(n)*DSX
1079:       Fth(n,m) = Fth(n,m) - ehtml
1080:       Fth_tau(n,m) = Fth_tau(n,m)-ehtmp
                         * tauezX1(j-Y1+1,k-Z1+1,n)
            ...
1084:       m = nt + nhyX1(j-Y1+1,k-Z1+1,n)
1085:       ehtmp = Ez(X1,j,k)*hyth(n)*dSX
1086:       Fth(n,m)     = Fth(n,m)     - ehtmp
1087:       Fth_tau(n,m) = Fth_tau(n,m) - ehtmp
                         * tauhyX1(j-Y1+1,k-Z1+1,n)
            ...
1108:     enddo
1109:   enddo
1311: enddo
```

Fig. 21. Code example:459.GemsFDTD.

```
// pbeampp.c
144: for (i=2, next=0; i<=B && i<=basket_size; i++)
145: {
       PREFETCH(perm[i+N]->a->cost);
       PREFETCH(perm[i+N]->a->tail->potential);
       PREFETCH(perm[i+N]->a->head->potential);
146:   arc = perm[i]->a;
147:   red_cost = arc->cost - arc->tail->potential
                + arc->head->potential;
     ...

165: for (; arc<stop_arcs; arc += nr_group)
166: {
       PREFETCH((arc+N*nr_group)->cost);
       PREFETCH((arc+N*nr_group)->tail->potential);
       PREFETCH((arc+N*nr_group)->head->potential);
167:   if (arc->ident > BASIC)
168:   {
170:     red_cost = arc->cost - arc->tail->potential
                  + arc->head->potential;
     ...
```

Fig. 22. Code example:429.mcf.

can prefetch stream accesses exhibited in other data structures and the benefit is still effective with software prefetching.

6.1.4 470.lbm. The `lbm` benchmark is dominated by a 3-D stencil update loop that references all 27 nearest neighbors in two 3-D arrays. Thus, there are many (up to $27 \times 2 = 54$) distinct streams with three distinct stride values. Both GHB and STR can prefetch these streams. As we discussed in Section 5.1.5, because of the high L1 cache hit ratio improvement, software prefetching performs better than hardware prefetching alone even though hardware prefetching also can predict addresses correctly.

Another unusual characteristic of the `lbm` benchmark is that it is very sensitive to prefetch distance variations. Software prefetching shows almost 100% coverage (Figure 10), because almost all the elements of `srcGrid` may be prefetched. Since the working set size is relatively large, very aggressive prefetching can easily generate early evictions. Figure 23 shows the performance and the distribution of prefetch requests as we vary prefetch distances. As these results depict, the performance of

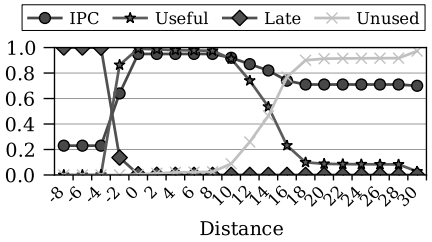


Fig. 23. Prefetch distance variance in 470.lbm.

```
// lbm.c
186: SWEEP_START(0, 0, 0, 0, SIZE_Z)
       // add macro for prefetches
       PREFETCH(DST_N'(dstGrid))
       PREFETCH(DST_S'(dstGrid))
       ...
210:   rho = SRC_C(srcGrid) + SRC_N(srcGrid) + ...
221:   ux  = SRC_E(srcGrid) - SRC_W(srcGrid) + ...
226:   uy  = SRC_N(srcGrid) - SRC_S(srcGrid) + ...
232:   ...
250:   DST_N(dstGrid) = ...*SRC_N(srcGrid) + rho* ...
251:   DST_S(dstGrid) = ...*SRC_S(srcGrid) + rho* ...
253:   ...
260: SWEEP_END
```

Fig. 24. Code example:470.lbm.

```
// gates.c
61: for (i = 0; i < reg->size; i++)
62:
      PREFETCH(reg->node[i+D])
65:   if (reg->node[i].state & ... )
66:     reg->node[i].state ^= ...
    ...
67:
```

Fig. 25. Code example:462.libquantum.

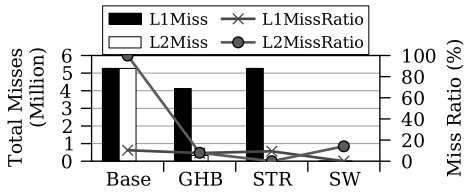


Fig. 26. # Cache misses ratio in 462.libquantum.

software prefetching is sensitive to the prefetch distance. In this example, one prefetch distance unit corresponds to one next loop iteration in Figure 24.

6.1.5 462.libquantum. The dominant memory access pattern in `libquantum` consists solely of fixed-stride streaming over a single array, `reg` which is shown in Figure 25. Hence, both hardware and software prefetching can predict the `reg->node` addresses. However, software prefetching provides 1.23x speedup over the best hardware prefetcher, due to the L1 cache miss penalties. In particular, even though the L1 latency is only three cycles in our baseline configuration, there are too many L1 cache misses due to the high miss ratio (shown in Figure 26). Thus, both execution cycles and on-chip L2 traffics increase. In `GHB` with software training, because it generates useless prefetches, 14% more prefetches become late (4% to 18%), resulting in a 55% performance degradation.

6.1.6 403.gcc. The dominant data structure of `gcc` is RDS. Since the code that accesses this RDS is very irregular, we could do not insert prefetches effectively in RDS. In addition, `gcc` suffers from indirect accesses. By inserting prefetches for such structures, we achieve a 14% improvement. Figure 27 shows a code snippet of the `gcc` benchmark.

6.1.7 434.zeusmp. The `zeusmp` benchmark has a 3-D stencil update memory access pattern, i.e., it iterates over points in a 3-D grid (array) and in each iteration accesses the nearest neighbors in all dimensions. Like `cactusADM`, it has a 3-D array with the iteration over points. However, unlike `cactusADM`, it accesses nearest neighbors in all three dimensions. Figure 28 shows a code snippet with software prefetching. The `v3` structures can be prefetched well by both hardware and software prefetching. Moreover, an extra 10% performance improvement using software prefetching comes from reducing the number of L1 cache misses. Also, a stream prefetcher with training generates an additional 4% improvement by eliminating some of the late prefetches.

```
// dominance.c
548: for (i = 1; i <= di->nodes; i++) {
       PREFETCH(&(di->dfs_to_bb[i+D]->index))
550:   if (i == e_index)
551:     continue;
552:   bb = di->dfs_to_bb[i]->index;
       ...

// BASIC_BLOCK(n) basic_block_info->data.bb[n]
// predict.c
664: for (n = 0; n < n_basic_blocks; n++) {
       PREFETCH(BASIC_BLOCK(n+N));
       PREFETCH(BLOCK_INFO(BASIC_BLOCK(n+N)));
665:   basic_block bb = BASIC_BLOCK(n);
666:   if (BLOCK_INFO(bb)->tovisit)
667:   ...
```

Fig. 27. Code example:403.gcc.

```
// tranx3.f
530 : do j=jbeg-1,jend
703 :   do i=ibeg-1,iend
827 :     do 2040 k=kbeg-1,kend+1
            PREFETCH(v3(i+8,j,k))
828 :       xi = (v3(i,j,k)-vg3(k))*dt
829 :       q1 = sign(haf,xi)
830 :       dtwid(k,1) = (0.5+q1) * (dlo(i,j,k-1)
831 :                  + (dx3a(k-1)-xi) * dd(k-1,1))
            ...
908 :     continue
1057:   continue
1060: continue
```

Fig. 28. Code example:434.zeusmp.

```
// blocksort.c
457: do {
458:  /* 1 */
459:  c1=block[i1]; c2=block[i2];
460:  if (c1!=c2) return (c1>c2);
461:  s1=quadrant[i1]; s2=quadrant[i2];
462:  if (s1!=s2) return (s1>s2);
463:  i1++; i2++;
      ...
501:  /* 8 */
502:  c1=block[i1]; c2=block[i2];
      ...
507:  if (i1>=nblock) i1-=nblock;
508:  if (i2>=nblock) i2-=nblock;
512: }
```

Fig. 29. Code example:401.bzip2.

```
// StaggeredLeapfrog2.fppized.f
317: do j=1,ny
318:   do i=1,nx
         PREFETCH(alp(i+K,j,k+1)
         PREFETCH(ADM_kxx_stag_p(i+8,j,k+1))
319:     lalp(i,j,km)=alp(i,j,1)
320:     fac=-2.0d0*dt*lalp(i,j,1)
321:     ADM_gxx(i,j,1) = ADM_gxx_p(i,j,1)
               + fac*ADM_kxx_stag_p(i,j,1)
322:     lgxx(i,j,km) = ADM_gxx(i,j,1)
         ...
331:   enddo
332: enddo
```

Fig. 30. Code example:436.cactusADM.

6.2 Neutral Group

6.2.1 401.bzip2. The core of the `bzip2` algorithm is the Burrows-Wheeler transform, which is also called block-sorting compression. As shown in Figure 29, since many delinquent loads are dependent on the outcomes of branches, hardware prefetchers are not effective. Software prefetching experiences the same problem. Figure 11 shows that there are many early prefetches in `bzip2`. Nonetheless, software prefetching is still able to improve performance by 8%.

6.2.2 436.cactusADM. The main data structure in `cactusADM` is a 3-D array with iteration over points. Each iteration only accesses nearest neighbors in one dimension. Figure 30 shows how prefetching is inserted for `alp` and `ADM_kxx_stag_p` data structures. Although all data structures in `cactusADM` are fairly easy to predict, the

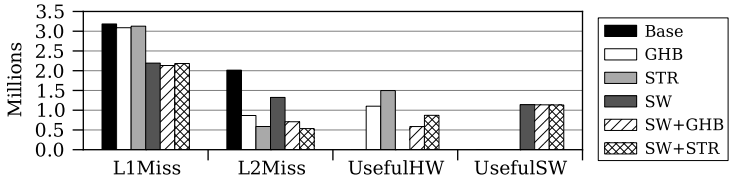


Fig. 31. Number of cache misses in 450.soplex.

```
// block_solver.f
167: do k=1,nz do j=1,ny do i=1,nx
176:   do l=1,nb
178:     do m=1,nb
           PREFETCH(a(l,m+K,i,j,k)
           PREFETCH(axp(l,m+K,i,j,k)
179:       y(l,i,j,k) = y(l.i.j.k)+
180:       a(l,m,i,j,k)*x(m,i,j,k)+
181:       axp(l,m,i,j,k)*x(m,ip1,j,k)+
           ...
187:     enddo
188:   enddo
```

Fig. 32. Code example:410.bwaves.

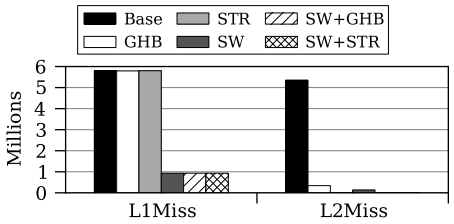


Fig. 33. Number of L1/L2 misses in 410.bwaves.

effectiveness of the stream prefetcher is much less than GHB. This is because there are too many in-flight streams, so the stream prefetcher misses some of them. However, because software prefetching covers many of the streams that the stream prefetcher cannot capture, the gap between GHB and the stream prefetcher is reduced in SW. GHB with training can reduce the number of late prefetches, which results in a 7% improvement.

6.2.3 450.soplex. soplex implements a simplex algorithm. It features primal and dual solving routines for linear programs and is implemented in C++. Many cache misses are occurred from C++ class overloaded operators and member functions that do not contain any loops. Without complete knowledge of its program structure, it is hard to decide where to insert software prefetches. Indeed, these memory accesses have a certain level of regularity, so hardware prefetching turns out to be reasonably effective.

As shown in Figure 31, although software prefetching is effective in terms of cache miss reduction, the reason why soplex is in the neutral group (3% degradation) is due to bandwidth consumption and many early/incorrect software prefetches. In Figure 5, soplex shows benefits in the SW+B and SW+P cases.

6.2.4 410.bwaves. The dominant computational loop in bwaves is a block-structured matrix-vector multiply, whose memory access pattern consists of 7 distinct unit-stride streams with no temporal reuse coupled with 7 additional semi-regular accesses (strided "bursts" of unit-stride access) on data with a high degree of temporal reuse. The cache misses are primarily due to the non-temporal unit-stride streams, which we would expect to be easily prefetched by conventional hardware mechanisms. Figure 32 shows the main loop of bwaves and how we insert software prefetches. Figure 33 shows the number of L1 and L2 cache misses. Compared with STR, SW+STR shows similar number of L2 cache misses, but it reduces L1 cache misses significantly. However, the performance of SW+STR is actually 5% worse than STR, due to instruction overhead. Specifically, the instruction count increases by 100% (from 200M to 400M), but the number of prefetch instructions is only 65M. Moreover, among 65M prefetches, more than 90% are redundant prefetches.

```
// tml.f
798: DO L = 1, 5
799:   DO I = I1+1, I2
         PREFETCH(DU(I+DIST,J,K,L)
800:     DU(I,J,K,L) = -DTVOL * (FSI(I,L) - FSI(I-1,L)
801:   ENDDO
802: ENDDO
```

Fig. 34. Code example:437.leslie3d.

6.3 Negative Group

6.3.1 482.sphinx3. The sphinx3 benchmark, a speech recognition application, contains a variety of access styles such as RDS, array of pointers, and regular strides, but is dominated primarily by unit-stride streams on short vectors. Hence, hardware prefetchers perform well for this benchmark. Because of timeliness issues and the short length of arrays, not all software prefetch requests are inserted for all the data structures, which results in lower coverage than hardware prefetchers (shown in Figure 10).

6.3.2 437.leslie3d. leslie3d (Figure 34) has very regular stride behavior. However, because the work between loop iterations is very limited, we inevitably use large prefetch distances. As a result, the coverage of SW is only 68% (HW: $>$ 90%). Even though there is a superposition effect with hardware prefetchers (now, more than 95%), the instruction overhead offsets the benefits.

7. RELATED WORK

There is considerable literature on both software and hardware prefetching. For different prefetch mechanisms, please refer to Vanderwiel and Lilja [2000] and the paper on MicroLib [Perez et al. 2004]. In this section, we only discuss the most relevant work that focuses on the limitations of software prefetching and the interaction of hardware and software prefetching.

Reducing software prefetching overhead. How to reduce the overhead of software prefetching has been discussed even from the first software prefetching proposal by Callahan et al. [1991]. They analyzed the overhead of software prefetching and suggested saving address recalculations by storing the calculated addresses into registers. Subsequently, many studies have discussed how to request software prefetch requests in a timely manner. Several algorithms were proposed to find the optimal prefetch distance in software prefetching [Badawy et al. 2004; Chen and Baer 1994; Mowry et al. 1992; Pai and Adve 2001; Vanderwiel and Lilja 2000]. Chen and Baer [1994] proposed using prologue and epilogue to insert software prefetches. Luk and Mowry [1996] proposed software prefetching mechanisms for RDS and also suggested several methods to reduce useless prefetches. Zhang et al. [2006] discussed the overhead of software prefetching and suggested using dynamic compilation to reduce the overhead of software prefetching, especially for reducing redundant prefetch requests.

Jerger et al. [2006] analyzed prefetch requests in multiprocessors and discussed harmful prefetches due to shared data. Their work focused on the negative effects of hardware prefetching on multiprocessors. Recently, the limitation of existing compiler algorithms for software prefetching in emerging chip multiprocessors is discussed [Son et al. 2009]. In their work, helper thread-based software prefetching is used rather than simple and low overhead software prefetching schemes. Our work is different from the previous work on reducing the overhead of software prefetching. Although some work provided the analysis of software prefetch requests, none of the work analyzed the behavior of software prefetching from the application's behavior, code

analysis, data structures, and algorithms. Furthermore, our work emphasizes the interactions between software and hardware prefetching. We also increase the understandability of software prefetching limitations and hardware prefetching limitations when both prefetching schemes are used together.

Hardware and software cooperative prefetching. Several mechanisms were proposed to take advantage of hardware and software prefetching together. Chen and Baer [1994] proposed a software prefetching mechanism that inserts data objects into a secondary cache and the a hardware supporting unit brings them to the primary cache. Wang et al. [2003] proposed guided region prefetching. In their work, the compiler inserts load instructions with hints indicating memory access patterns, and the information is consumed by hardware prefetchers. Saavedra and Park [1996] proposed a runtime-adaptive software prefetching mechanism to adjust prefetch distances and degree. Luk and Mowry [1998] proposed a hardware-based filtering mechanism for harmful software prefetch requests. In their work, they only focused on instruction prefetching, which is not a critical performance bottleneck in today's processors. All these mechanisms were built on the assumption that software and hardware prefetching provide superposition effects. In contrast to the above work, our work provides detailed analysis of when software and hardware prefetching can generate synergistic/antagonistic effects. Our work is focused on improving the understanding of interactions rather than proposing a new hardware or software prefetching mechanism.

8. CONCLUSION AND FUTURE WORK

Our detailed empirical and qualitative study of the benefits and limitations of software prefetching provides concrete examples showing the complexity of current schemes and most interestingly, on the synergistic and antagonistic *interaction effects* between software and hardware prefetching.

Our analysis of the SPEC CPU 2006 suite, whose benchmarks we classified into positive, neutral, and negative groups based on observed software/hardware prefetch interactions, leads us to confirm, quantitatively the following conclusions.

(1) Having to select a static prefetch distance is generally considered one of the limitations of more aggressive software prefetching. Indeed, in benchmarks like `lbm` where aggressive prefetching generates many early prefetch requests, the optimal prefetch distance becomes sensitive to the machine configuration. However, we observe that for most of the benchmarks in our suite, performance is not very sensitive to distance, and so not too difficult to tune. Moreover, we observe that when software prefetch has a high coverage of delinquent loads, we can overcome the overhead of extra instructions due to prefetching. Thus, provided we can identify the right context for prefetching, these two observations imply that we can prefetch more aggressively than previously believed.
(2) Prefetching for short streams shows the most positive effects, as seen in the `milc` benchmark. Hardware prefetchers cannot prefetch short streams, irregular addresses, or a large number of concurrent streams. Thus, it is in these contexts that we should expect to use software-based prefetching techniques more.
(3) Coverage should be the main metric to decide which prefetching scheme to use. It is via this metric that we see that short stream and indirect references are good candidates for using software prefetching; and that, by contrast, if a data structure shows strong stream behavior or regular stride patterns, using software prefetching does not provide a significant performance improvement and can even degrade the performance of hardware prefetching.

Future work will focus on developing software prefetch algorithms to decide how to train hardware prefethers and reducing the negative interactions between software and hardware prefetching schemes so that compilers can insert prefetch instructions more aggressively.

REFERENCES

AL-SUKHNI, H., BRATT, I., AND CONNORS, D. A. 2003. Compiler-directed content-aware prefetching for dynamic data structures. In *Proceedings of the 12th International Conference on Parallel Architecture and Compilation Technology*. IEEE, Los Alamitos, CA, 91–100.

AMD. AMD Phenom II Processors. http://www.amd.com/us/products/desktop/processors/phenom-ii/Pages/phenom-ii.aspx.

BADAWY, A.-H. A., AGGARWAL, A., YEUNG, D., AND TSENG, C.-W. 2004. The efficacy of software prefetching and locality optimizations on future memory systems. *J. Instruct.-Level Parallelism 6*.

BAER, J. AND CHEN, T. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. ACM, New York, NY, 176–186.

CALLAHAN, D., KENNEDY, K., AND PORTERFIELD, A. 1991. Software prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, 40–52.

CHEN, S., AILAMAKI, A., GIBBONS, P. B., AND MOWRY, T. C. 2007. Improving hash join performance through prefetching. *ACM Trans. Datab. Syst. 32*, 3, 17.

CHEN, T.-F. AND BAER, J.-L. 1994. A performance study of software and hardware data prefetching schemes. In *Proceedings of the 16th International Symposium on Computer Architecture*. 223–232.

COLLINS, J. D., TULLSEN, D. M., WANG, H., AND SHEN, J. P. 2001. Dynamic speculative precomputation. In *Proceedings of the 34th International Symposium on Microarchitecture*. IEEE Computer Society, Los Alamitos, CA, 306–317.

COLLINS, J. D., SAIR, S., CALDER, B., AND TULLSEN, D. M. 2002. Pointer cache assisted prefetching. In *Proceedings of the 35th International Symposium on Microarchitecture*. IEEE Computer Society Press, Los Alamitos, CA, 62–73.

COOKSEY, R., JOURDAN, S., AND GRUNWALD, D. 2002. A stateless, content-directed data prefetching mechanism. In *Proceedings of the 10th International Conference on Architectural Support for Prog. Languages and Operating Systems*. ACM, New York, NY, 279–290.

EBRAHIMI, E., MUTLU, O., AND PATT, Y. N. 2009. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, Los Alamitos, CA, 7–17.

EMMA, P. G., HARTSTEIN, A., PUZAK, T. R., AND SRINIVASAN, V. 2005. Exploring the limits of prefetching. *IBM J. Resear. Devel. 49*, 127–144.

GCC-4.0. GNU compiler collection. http://gcc.gnu.org/.

HUR, I. AND LIN, C. 2006. Memory prefetching using adaptive stream detection. In *Proceedings of the 39th International Symposium on Microarchitecture*. IEEE Computer Society, Los Alamitos, CA, 397–408.

HUR, I. AND LIN, C. 2009. Feedback mechanisms for improving probabilistic memory prefetching. In *Proceedings of the 15th International Symposium on High Perf Compo Architecture*. IEEE Computer Society, Los Alamitos, CA, 443–454.

ICC. Intel C++ compiler. http://www.intel.comlcd/software/products/asmo-na/eng/compilers/clin/277618.htm.

INTEL. 2004. Intel Pentium M Processor. http://www.intel.com/design/intarch/pentiumm/pentiumm.htm.

INTEL. 2007. Intel core microarchitecture. http://www.intel.com/technology/45nm/index.htm?iid=tech_micro+45nm.

INTEL. 2008. Intel AVX. http://software.intel.com/en-us/avx.

INTEL. 2009. Intel Nehalem microarchitecture. http://www.intel.com/technology/architecture-silicon/next-gen/index.htm?iid=tech_micro+nehalem.

INTEL. 2011. *Intel 64 and IA-32 Architectures Software Developer's Manual*. http://www3.intel.com/Assets/PDF/manual/253667.pdf.

JERGER, N., HILL, E., AND LIPASTI, M. 2006. Friendly fire: Understanding the effects of multiprocessor prefetches. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, Los Alamitos, CA, 177–188.

JOSEPH, D. AND GRUNWALD, D. 1997. Prefetching using Markov predictors. In *Proceedings of the 19th International Symposium on Computer Architecture*. ACM, New York, NY, 252–263.

JOUPPI, N. P. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 12th International Symposium on Computer Architecture*. ACM, New York, NY, 388–397.

KROFT, D. 1981. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 3rd International Symposium on Computer Architecture*. IEEE Computer Society Press, Los Alamitos, CA, 81–87.

LAI, A.-C., FIDE, C., AND FALSAFI, B. 2001. Dead-block prediction and dead-block correlating prefetchers. In *Proceedings of the 23rd International Symposium on Computer Architecture*. ACM, New York, NY, 144–154.

LIPASTI, M. H., SCHMIDT, W. J., KUNKEL, S. R., AND ROEDIGER, R. R. 1995. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th International Symposium on Microarchitecture*. IEEE Computer Society Press, Los Alamitos, CA, 232–236.

LUK, C.-K. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 23rd International Symposium on Computer Architecture*. ACM, New York, NY, 40–51.

LUK, C.-K. AND MOWRY, T. C. 1996. Compiler-based prefetching for recursive data structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*. 222–233.

LUK, C.-K. AND MOWRY, T. C. 1998. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings of the 31st International Symposium on Microarchitecture*. IEEE Computer Society Press, Los Alamitos, CA, 182–194.

MOWRY, T. C., LAM, M. S., AND GUPTA, A. 1992. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, 62–73.

NESBIT, K. J., DHODAPKAR, A. S., AND SMITH, J. E. 2004. AC/DC: An adaptive data cache prefetcher. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Technology*. IEEE Computer Society, Los Alamitos, CA, 135–145.

NESBIT, K. J. AND SMITH, J. E. 2004. Data cache prefetching using a global history buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, Los Alamitos, CA, 96–105.

PAI, V. S. AND ADVE, S. V. 2001. Comparing and combining read miss clustering and software prefetching. In *Proceedings of the 10th International Conference on Parallel Architecture and Compilation Technology*. IEEE Computer Society, Los Alamitos, CA, 292–303.

PATIL, H., COHN, R., CHARNEY, M., KAPOOR, R., SUN, A., AND KARUNANIDHI, A. 2004. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *Proceedings of the 37th International Symposium on Microarchitecture*. IEEE Computer Society, Los Alamitos, CA, 81–92.

PEREZ, D. G., MOUCHARD, G., AND TEMAM, O. 2004. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *Proceedings of the 37th International Symposium on Microarchitecture*. IEEE Computer Society, Los Alamitos, CA, 43–54.

PIN. A binary instrumentation tool. http://www.pintool.org.

ROTH, A. AND SOHI, G. S. 1999. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 21st International Symposium on Computer Architecture*. IEEE Computer Society, Los Alamitos, CA, 111–121.

SAAVEDRA, R. H. AND PARK, D. 1996. Improving the effectiveness of software prefetching with adaptive execution. In *Proceedings of the 5th International Conference on Parallel Architecture and Compilation Technology*. IEEE Computer Society, Los Alamitos, CA, 68–78.

SON, S. W., KANDEMIR, M., KARAKOY, M., AND CHAKRABARTI, D. 2009. A compiler-directed data prefetching scheme for chip multiprocessors. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY, 209–218.

SRINATH, S., MUTLU, O., KIM, H., AND PATT, Y. N. 2007. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, Los Alamitos, CA, 63–74.

TENDLER, J., DODSON, S., FIELDS, S., LE, H., AND SINHAROY, B. 2002. POWER4 system microarchitecture. *IBM J. Resear. Devel. 46*, 1, 5–25.

VANDERWIEL, S. P. AND LILJA, D. J. 2000. Data prefetch mechanisms. *ACM Comput. Surv. 32*, 2, 174–199.

WANG, Z., BURGER, D., MCKINLEY, K. S., REINHARDT, S. K., AND WEEMS, C. C. 2003. Guided region pre fetching: A cooperative hardware/software approach. In *Proceedings of the 25th International Symposium on Computer Architecture*. ACM, New York, NY, 388–398.

WU, Y. 2002. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, 210–221.

YANG, C.-L., LEBECK, A. R., TSENG, H.-W., AND LEE, C.-H. 2004. Tolerating memory latency through push prefetching for pointer-intensive applications. *ACM Trans. Architect. Code Optim. 1*, 4, 445–475.

ZHANG, W., CALDER, B., AND TULLSEN, D. M. 2006. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *Proceedings of the 4th International Symposium on Code Generation and Optimization*. IEEE Computer Society, Los Alamitos, CA, 50–64.

ZILLES, C. AND SOHI, G. 2001. Execution-based prediction using speculative slices. In *Proceedings of the 23rd International Symposium on Computer Architecture*. ACM, New York, NY, 2–13.

Received November 2010; revised July 2011; accepted August 2011