Journal of Circuits, Systems, and Computers

The interaction and relative effectiveness of hardware and software data prefetch

SANTHOSH VERMA
Department of Electrical And Computer Engineering,
Louisiana State University, Baton Rouge, Louisiana 70803, United States
†*sverma3@lsu.edu*

DAVID M. KOPPELMAN
Department of Electrical And Computer Engineering,
Louisiana State University, Baton Rouge, Louisiana 70803, United States
§*koppel@ece.lsu.edu*

Received (08th August 2010)
Revised (12th December 2010)
Accepted (Day Month Year)

A major performance limiter in modern processors is the long latencies caused by data cache misses. Both compiler and hardware based prefetching schemes help hide these latencies and so improve performance. Compiler techniques infer memory access patterns through code analysis, and insert appropriate prefetch instructions. Hardware prefetching techniques work independently from the compiler by monitoring an access stream, detecting patterns in this stream and issuing prefetches based on these patterns. This paper looks at the interplay between compiler and hardware architecture based prefetching techniques. Does either techniques make the other one unnecessary? First, compilers' ability to achieve good results without extreme expertise is evaluated by preparing binaries with no prefetch, one-flag prefetch (no tuning), and expertly tuned prefetch. From runs of SPECcpu2006 binaries, we find that expertise avoids minor slowdown in a few benchmarks and provides substantial speedup in others.

We compare software schemes to hardware prefetching schemes and our simulations show software alone substantially outperforms hardware alone on about half of a selection of benchmarks. While hardware matches or exceeds software in a few cases, software is better on average. Analysis reveals that in many cases hardware is not prefetching access patterns that it is capable of recognizing, due to some irregularities in the observed miss sequence. Hardware outperforms software on address sequences that the compiler would not guess. In general, while software is better at prefetching individual loads, hardware partly compensates for this by identifying more loads to prefetch. Using the two schemes together provides further benefits, but less than the sum of the contributions of each alone.

1. Introduction

The performance of modern processors is significantly limited by long access latencies to the secondary caches and especially to memory. A cache miss to the main memory results in a latency of hundreds of cycles, and most of these cycles are often

spent by the processor stalling. For example, the SPECcpu2000 benchmarks spend more than half their time stalling for load instructions that access main memory [1]. Prefetching is a technique whereby data needed by a program is requested earlier than it otherwise would, so that it is either already in the cache or on its way by the time the instruction needing the data is executed. This makes it possible to reduce or completely hide the cache miss latency.

Software prefetching is a compiler based technique, where special prefetch instructions are inserted into the program ahead of the original instruction requesting the data. If the instruction is placed sufficiently ahead and is accurate, most or all of the miss latency will be covered by the time the instruction in the original program executes. While software prefetching has the potential to be very highly targeted and very accurate, there are some drawbacks and limitations. Firstly, processor resources have to be used to fetch, decode and execute these instructions. Hence, these instructions not only increase the code size but also use up resources that could otherwise be used by regular program instructions. For the benchmarks we simulated, software prefetch instructions were 5.4% of all committed instructions. This use of resources does not guarantee that the prefetch is useful because of the possibility of the prefetched data already being present in the cache. In fact, the prefetch can be damaging if it evicts an existing cache line that would otherwise have been used in a different part of the program. Secondly, it is not always easy for the compiler to place software prefetch instructions significantly ahead of the instruction making the demand request. This reduces the extent to which software prefetch instructions can hide miss latencies, particularly misses to main memory.

Hardware based prefetching techniques are independent of the compiler, and work by monitoring the stream of requests within the memory hierarchy and detecting patterns in this stream. Prefetches are issued directly by the hardware based on these patterns. The main advantage of hardware prefetching is that it works without increasing the code size or interfering with the underlying program being executed. Hardware prefetchers are more flexible because they will automatically stop issuing prefetches if data is consistently found in the caches. Since they monitor patterns globally, hardware prefetchers can also detect patterns which are not easily detectable by the compiler. The drawbacks of hardware prefetchers are the increase in complexity of the processor and the strain they put on memory bandwidth by creating additional memory traffic.

Since both techniques have their advantages and disadvantages, we focus on the interplay between the compiler based software technique and hardware techniques. We evaluate one simple hardware prefetcher and one advanced hardware prefetcher implemented in real-world state of the art processors as well as another advanced prefetcher proposed by computer architects in literature. We examine the software vs. hardware prefetch question starting from commercial hardware and compilers, and using code tuned for prefetch by motivated experts. Specifically, we look at the SPECcpu2006 benchmarks, which were chosen to exercise the CPU and memory system, and whose rules allow testers to prepare both baseline (or base) and peak

(sometimes called result) tuning of the benchmarks. Base tuning is intended to reflect typical effort in setting compiler and other options, whereas peak tuning reflects thorough tuning of each program in the suite. If a tester is testing his company's processor, as is often the case, we can be sure the tester has all the expertise and resources necessary to get the highest peak score on that particular processor. Starting with these "config" files produced by motivated experts we prepared binaries without software prefetch, having baseline software prefetch, and the original peak software prefetch, with prefetch settings being the only difference between the builds. Data was collected from native runs, in which the hardware prefetch scheme is fixed, and also simulation runs, in which hardware prefetch mechanisms of varying complexity were used.

We show that while software based prefetching schemes have the potential to be very effective, their effectiveness is often limited in practical settings by the need for significant human effort and expertise during compilation. Under conditions of real world constraints, software developers may not have the time to tune their programs for software prefetching or the availability of experts who have a detailed understanding of the architecture and compilers used for the target system. Nevertheless, software prefetch schemes often outperform hardware prefetch schemes. The reasons for these performance differences, whether they favor hardware or software, are investigated for the three hardware prefetch schemes, looking at the natural strengths of each. Programs that are well tuned for software prefetching benefit further from hardware prefetching. Hence, the best prefetching solution is one that involves both compiler and hardware based techniques. We look into the reasons for the observed performance by analyzing prefetcher coverage and accuracy data, and observe that both correlate well with performance. By analyzing average coverage data and comparing it to coverage for only the most commonly prefetched loads, we conclude that while software is more effective at prefetching important individual loads, hardware partly compensates for this by identifying more loads to prefetch.

2. Background

In this section, we look at related work done on hardware prefetching schemes, compiler based software prefetchers and work related to combined hardware and software prefetching.

The simplest hardware prefetching schemes are sequential schemes which prefetch the next cache lines [2] [3] if a sequential access pattern is detected. Stride prefetching schemes [4] [5] issue prefetches if a constant stride in the memory access is detected. A table stores the most recent stride as well as the most recent memory address that was issued for a load instruction. A stride is computed by calculating the difference between the memory address value of the current access and the previous address value that is stored in the table entry. If the stride is the same between consecutive access', a two-bit status field is incremented. The limitations of basic Stride prefetchers are their use of program counters which may not be easily

accessible at the L2 cache level and their reliance on detecting per instruction patterns instead of global memory patterns. Stride Stream Buffer Czone prefetching [6] overcomes these issues by dividing the memory space into fixed size partitions (Czones). The accesses in each Czone are monitored to determine stride patterns, avoiding the need for a program counter.

More complex address patterns are considered in correlation prefetching methods such as Markov predictors [7] which predict repeated and irregular address sequences. Instead of directly accessing a table, Global History Buffer based schemes [8] [9] utilize entries which store an address and a link to a previous entry. The GHB is a FIFO table of the most recent addresses and can be used to create a time ordered linked list of addresses. In C/DC prefetching [8], an index table holds an initial pointer to an entry in the GHB. The index table is accessed using the tag of the Czone memory region and delta correlation address patterns are detected.

Nesbit et al. [8] also propose an adaptive component in which the CZone size and prefetch degree of the C/DC predictor is varied dynamically based on the detection of a change in program phase. Adaptive hardware prefetch schemes adjust the aggressiveness of prefetching based on some type of data. Dahlgreen et al. [10] propose an adaptive prefetching scheme for sequential access' by monitoring the percentage of useful prefetches and adapting aggressiveness if this percentage is above a threshold. Hur and Lin [11] propose Adaptive Stream Detection which adjusts the aggressiveness of a Stream prefetcher based on stream length histograms. Ramos et al. [12] propose low cost solutions for adjusting sequential prefetcher aggressiveness and Srinath et al. [13], adjust aggressiveness based on certain feedback metrics collected at intervals during program execution.

Hardware Prefetchers are used in real systems. The Sun Ultrasparc IV system and the IBM Power 4 system use stream based sequential prefetchers. The Intel Core Microarchitecture used in multi-core systems has stride based prefetchers at both the L1 and L2 level.

The idea of software prefetching was originally proposed by Callahan et al. [14] who discuss a simple compiler based strategy to select prefetches and some basic optimizations to eliminate unnecessary prefetches. Ghosh et. al [15] explain that accurately determining which loads to prefetch requires precise representation of cache misses in the program, and propose a mathematical framework to provide this representaiton. Mowry et. al [16] use locality analysis, loop splitting and software pipelining to efficiently insert prefetches in scientific code dealing with dense matrices. Their algorithm factors in parameters such as cache line size, cache size and the size of loop bounds in its decision making. Klaiber et al. [17] estimate cache miss latencies and instruction execution rates in their compiler based software controlled prefetching scheme. Ranganathan et. al [18] study the effectiveness of software prefetching in shared-memory multiprocessors built from ILP processors. They observe that while software prefetching provides significant benefits in terms of execution time reduction, the benefits are lower than those achieved for multiprocessors built from previous generation, non-ILP processors. Vanderweil and Lilja [19] survey

several approaches, including combined hardware/software schemes, and discuss the design tradeoffs involved in implementing a strategy.

Wang et. al [20] propose a combined Hardware and Software prefetching approach in Guided Region Prefetching, in which hints placed in load instructions by the compiler indicate presence of pointer structures, spatial locality indirect array accesses, indirect pointer and size hints. While the hardware does the prefetching, it is kept simple. For example, instead of using complex hardware to recognize pointer traversal patterns or store pointer correlations, this prefetching scheme greedily generates a prefetch for any fetched value that falls within the ranges of legitimate heap memory addresses. Gornish and Veidenbaum [21] propose an integrated scheme where software is used to decide the number of contiguous blocks that the hardware should prefetch upon a miss. Baer and Chen [4] have the compiler determine address and stride information and provide it to a hardware prediction table. Karlsson et. al [22] use compiler generated prefetch arrays to assist the hardware in doing pointer prefetching in linked data structures. Lu et al. [23] implement runtime prefetching by using a dynamic trace based optimization system which selects and profiles traces and uses the profiled information to assist prefetching. Son et. al [24] observe that the benefits of data prefetching are significantly reduced as the number of cores is increased, largely due to harmful prefetches. They propose a compiler directed scheme for on-chip cache based CMP's. The scheme reduces the number of prefetches issued by having the compiler identify program phases, followed by dividing the threads into phase based groups and assigning a customized prefetcher thread for each group. Badawy et al. [25] propose a locality optimization based software prefetching scheme and evaluate the impact of combining hardware prefetching with their scheme. They observe that locality optimizations enable stride based hardware prefetching for benchmarks that normally do not exhibit striding.

Some real-world systems also use some type of combined software/hardware prefetching. The IBM Power 4 uses compiler hints to enable its Stream prefetcher to prefetch more aggressively. In most cases, the hardware waits for several cache misses before initiating a prefetch. However, the software can tell the hardware to initiate a prefetch early in some cases without waiting for the higher number of cache misses. The Ultrasparc IV+ has special prefetch instructions called strong prefetches which will not be dropped even in the case of a TLB miss or if the prefetch queue is full.

This paper focuses on the interaction and relative performance differences between hardware prefetching and the software prefetching schemes implemented by a state of the art compiler. In prior work in the area of hardware prefetching, the affect of software prefetching on the implemented hardware prefetchers is not discussed. Infact, it is rarely mention whether software prefetch instructions are built into and/or honored by the simulators.

3. Software Prefetching

Software prefetch is the use of prefetch instructions provided in most modern instruction sets. A typical prefetch instruction has an address argument and a hint argument. The hint indicates something about the expected future access to the address such as whether it will be read but not written, or that it will be accessed just once. In response hardware can move the addressed data towards the CPU, perhaps placing it in the level-2 cache. Hardware can also ignore prefetch instructions, perhaps in response to congestion, or always for a low-cost design.

When coding in a high-level language, prefetch instructions can explicitly be placed by the programmer using some language extension (such as a library of macros expanding to inline assembler statements), or they can be placed by the compiler. Programmer placement of prefetch instructions is tedious and can slow down execution if not done well. Compilers too can slow down execution of code by ill-chosen prefetch instructions, and so many compilers do not emit prefetch instructions unless a prefetch compile option is used. If a prefetch instruction is not needed (because the data is already cached) it can slow down execution by delaying the fetch of useful instructions. Worse, it can bring in data which is never accessed but evicts other data therefore inducing a cache miss rather than avoiding one.

Compilers typically emit prefetch instructions for array accesses inside of loops, when a future array index can be determined, as it easily can in many cases, such as when the array index is the loop index. An estimate of the ratio of memory latency to loop body time provides the prefetch distance. More sophisticated prefetch schemes exist and are being developed. Often a compiler can make better prefetch decisions with knowledge of the number of iterations in each loop, and other information about the shape and access frequency of the data. It analyzes data from a profile run and inserts prefetch instructions with the goal of getting them started early enough so that most or all of the miss latency of a future instruction is hidden.

An application developer compiling a program has to select from a wide variety of prefetch options which determine, amongst other things, the aggressiveness and type of software prefetching that will be compiled into the program. The selection of these parameters is not trivial because one needs to know not only which parts of the code need prefetching, but how to set the flags to achieve prefetching in those parts but not others. A set of poorly selected options (such as making prefetching too aggressive) will not provide the optimal performance benefit and can easily hurt performance. The following are the set of prefetch options from the Sun Studio 12 compiler that we used in our simulations.

xprefetch - If set to "yes" or "auto, explicit", allow generation of prefetch instructions.
xprefetch_level - Determines the aggressiveness, i.e, how much effort the compiler puts in to finding prefetch opportunities.
xprefetch_auto_type - Determines whether or not the compiler generates indirect

prefetches for data arrays accessed indirectly.
xprefetch=latx:n - Adjust the compilers assumptions about prefetch latency by a factor of n.

4. Importance and Practicality of Software Prefetcher Tuning

To measure the impact of expert prefetch tuning separate SPECcpu2006 peak-tuning binaries were generated for a Fujitsu M5000 [26]. This is a dynamically scheduled system that implements the SPARC V9 instruction set, including prefetch instructions. It includes a large L1 data cache, 128 KB, and a large L2 cache, 6 MB, and uses a simple stream hardware prefetcher. The SPECcpu2006 benchmarks were built using configuration files based on one prepared by Fujitsu and disclosed to SPEC. Three sets of binaries were generated as described below.

1. Prefetching is explicitly turned off, but expert tuning is available for other components of the benchmark code.
2. Expert tuning is not available for prefetching, but available for other components of the benchmark code (referred to as base prefetch tuning).
3. The original configuration file (with minor changes that do not affect code generation), where expert tuning is available for prefetching and for other components of the benchmark code (referred to as peak prefetch tuning).

Of the 17 benchmarks presented here, 7 had non-default prefetch flags. Figure 1 demonstrates the impact that expert software prefetcher tuning has on performance by comparing the three configurations above. These results are obtained from runs performed on a Fujitsu M5000 configured with eight 2-core SPARC64 VI processors. The performance ratio is the improvement ratio over a reference system [27] that SPEC uses to normalize performance metrics. The reference system is a Sun machine with a 296 MHz UltraSPARC II processor.

Expert tuning for prefetching had a major impact on three of the seventeen benchmarks, yielding speedups of 20%, 6%, and 6% respectively. The results demonstrate the importance of expert (peak) for software prefetching. Peak prefetch tuning outperforms a system with base prefetch tuning by an average of 6% and a system with no software prefetching by an average of 23%. These speedups are significant given that they are achieved without any increase in hardware costs. For benchmarks sjeng, omnetpp and astar, base prefetching tuning has a negative impact on performance but the decline for sjeng and astar is minimal. A bigger potential issue for benchmarks that do not benefit from software prefetching is that any software prefetch instructions in these benchmarks take up code space and may waste energy when they are fetched, decoded and executed. This waste of energy is an important consideration in current generation processors, where conserving power has become an important a design consideration as maximizing performance. For the benchmarks that we simulated in the next section, software prefetch instruc-

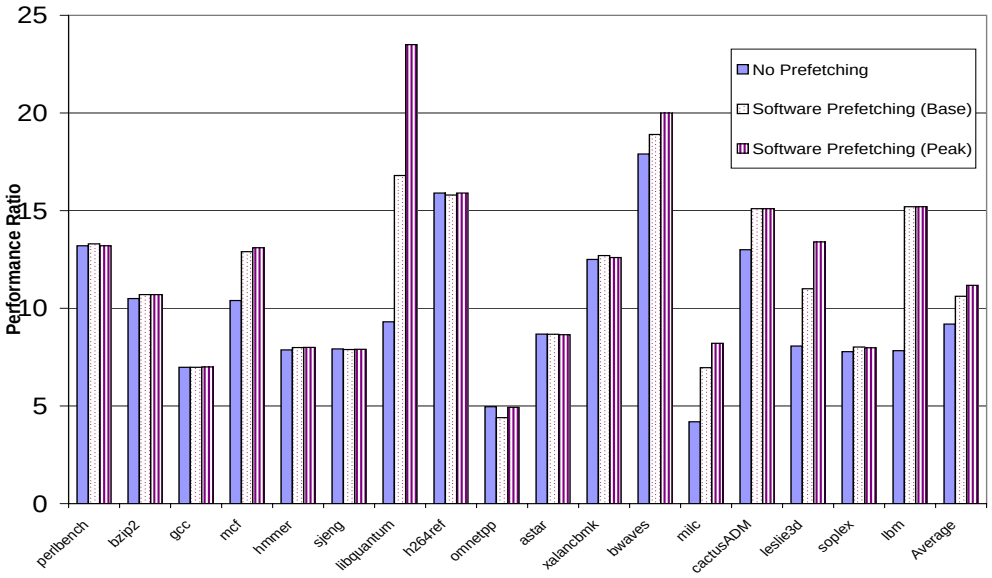

Fig. 1. Impact of Expert Tuning on Performance

tions represented 5.4% of all dynamic instructions under base tuning. There are a total of nine benchmarks (including the three mentioned in this paragraph) in the above chart where expert tuning simply consists of turning software prefetching off. Instead of choosing a generic prefetch setting for all benchmarks, turning prefetching off on a selective basis can save code space and energy and can even prevent a performance drop.

5. Hardware Prefetchers

We test three prefetch mechanisms of varying complexity: sequential, stride, and C/DC. The sequential prefetcher is a simple but effective design that easily exploits spatial locality and sequential access. Most general-purpose processors have some variation of sequential prefetch, including the SPARC64VI. The stride prefetcher exploits accesses at some regular stride, the design tested here is more capable than stride prefetchers present in real systems [28]. The difference correlated predictor (C/DC) [8] [9] is the most complex and powerful, it can prefetch repeated address sequences as well as repeated patterns. It has not been incorporated in real designs.

All simulated hardware prefetch schemes monitor the stream of addresses that miss the L1 and L2 caches. Streams that hit the L2 cache will generate L2 to L1 prefetches, and those that hit memory will generate memory to L2 prefetches. Hardware prefetches are rejected or later dropped if cache/memory request resources are fully occupied, to avoid interference with demand requests.

5.1. *Sequential Prefetch*

The simplest evaluated scheme is an aggressive sequential prefetcher. Sequential prefetch mechanisms predict the sequence $a, +a+L, a+2L, \ldots$, where a is a memory address and L is the line size. Prefetch is initiated on a cache miss to either the L1 or L2 cache and prefetch requests are generated for up to the next d lines, where d is the prefetch degree. Arriving lines are tagged as having been prefetched and access to such a line at address a triggers the prefetch of a line at address $a + d \times L$. That is, a miss initiating prefetch brings d lines, while hits bring in one each. Sequential prefetch requires only a tiny amount of storage, one bit per line, but some increase in cache controller complexity, including logic to drop prefetches when the system is busy.

5.2. *Stride Prefetch*

Stride prefetchers predict sequences of the form $a, a+s, a+2s, \ldots$, where s is called the stride. A challenge in the design of stride prefetchers is to determine just what the strides are. The implemented scheme discovers strides by separately tracking misses in each region of memory. Two tables are used, a direct-mapped correlation table for finding address strides, and a set-associative stream table for generating prefetch requests based on discovered strides.

The correlation table is accessed on a cache miss, but only if the address also misses the stream table. The correlation table is indexed using higher-order address bits (thus correlating strides by memory region). An entry holds the address of the last miss and a *delta* (the difference between the previous two miss addresses). (Some designs include a count of how many consecutive times the delta appeared). A new delta is computed and if it matches the prior one an entry is created in the stream table using the delta. Otherwise, to discover and filter out sequential access patterns, a stream table entry is created using a delta of one line. The correlation table is updated with the new address and delta.

The stream table is accessed on each cache lookup and is indexed by address. An entry contains a tag, delta, distance, and state information. Using this entry the next prefetch address is $a + c \times \Delta$, where a is the miss address, c is a distance, and Δ is the delta. The system will generate several prefetches, $a + x \times \Delta$, $x = c, c+1, c+2, \ldots$, until a maximum degree is met (the default is 4, shared by all other prefetch schemes) or until a prefetch is rejected. If p prefetches were generated the entry is written back with c changed to $c - 1 + p$. A stream table entry is initialized with a distance of one. Note that the distance field provides flexibility in how far ahead prefetch reaches. During busy times c will be low as prefetch requests are rejected. During quiet times it can build a reserve.

5.3. *C/DC Prefetch*

The most elaborate prefetch mechanism used is one based on difference history, essentially the C/DC prefetcher presented by Nesbit et al. [8]. The differ-

ence history of address stream $a_0, a_1, a_2, \ldots$ is the sequence $\Delta_1, \Delta_2, \Delta_3, \ldots$, where $\Delta_i = a_i - a_{i-1}$. Given some difference history $\ldots, \Delta_{t-1}, \Delta_t$ the prefetcher will predict $\Delta_{t+1}, \Delta_{t+2}, \ldots$ and use these to generate prefetch addresses, $a_t + \Delta_{t+1}, a_t + \Delta_{t+1} + \Delta_{t+2}, \ldots$. To predict $\Delta_{t+1}, \ldots$ the CD predictor scans the difference for a prior occurrence of the two most recent deltas, that is, it finds the largest $x < t$ such that $\Delta_{x-1} = \Delta_{t-1}$ and $\Delta_x = \Delta_t$. Difference Δ_{x+1} is thus the predicted Δ_{t+1}, etc.

C/DC [8] is implemented using two tables, a small correlation table and a larger global history buffer (GHB). The correlation table is indexed by region and provides a GHB index as well as the previous address in the region. On a miss or prefetch hit a correlation table entry is retrieved, it is used to initialize a new GHB entry (these are allocated sequentially), using the delta in this and the older entry, the GHB is scanned for prior consecutive occurrence of these deltas, if a match is found the following deltas are used to predict prefetch addresses. Normally, C/DC is used for L2 prefetch so that there is time for the sequential scan of the GHB.

In the implementation simulated here, the scan is instantaneous, though limited to 100 entries. Also, if $\Delta t - 1 = \Delta_t$ a stride sequence is assumed, and so the GHB is not scanned. GHB entries are also used to infer a sequential direction, that is used if the deltas are not found.

6. Evaluation of Hardware Based Prefetchers

6.1. *Experimental Framework*

We use an extensively modified version of the RSIM simulator [29]. The simulator does detailed simulation of a dynamically scheduled superscalar processor and memory system and implements a subset of the SPARC V9 ISA [30]. The parameters of the simulator are adjusted so that it matches the Fujitsu M5000 system used to evaluate the software prefetching schemes described previously as closely as possible. It fetches a maximum of four instructions and one branch instruction per cycle, and has an L1 cache of size 128 KB and an 4 MB L2 cache. Since these benchmark inputs are very large, we use periodic sampling to select a set of samples for simulation. Specifically, 100 samples are selected for data collection at regular intervals throughout the program, up to a maximum of the first 700 billion instructions in the program. Each sample has a warmup period of 500K instructions (no statistics are collected during warmup), followed by a segment of 3.5 million instructions over which statistics are collected. We simulate a prefetcher friendly subset of 10 benchmarks from SPECcpu2006.

6.2. *Performance comparison of Hardware Only with Software Only Prefetching*

In the charts presented in this sub-section and the next sub-section, speedup is measured against a system which has no prefetching and is compiled using base settings.

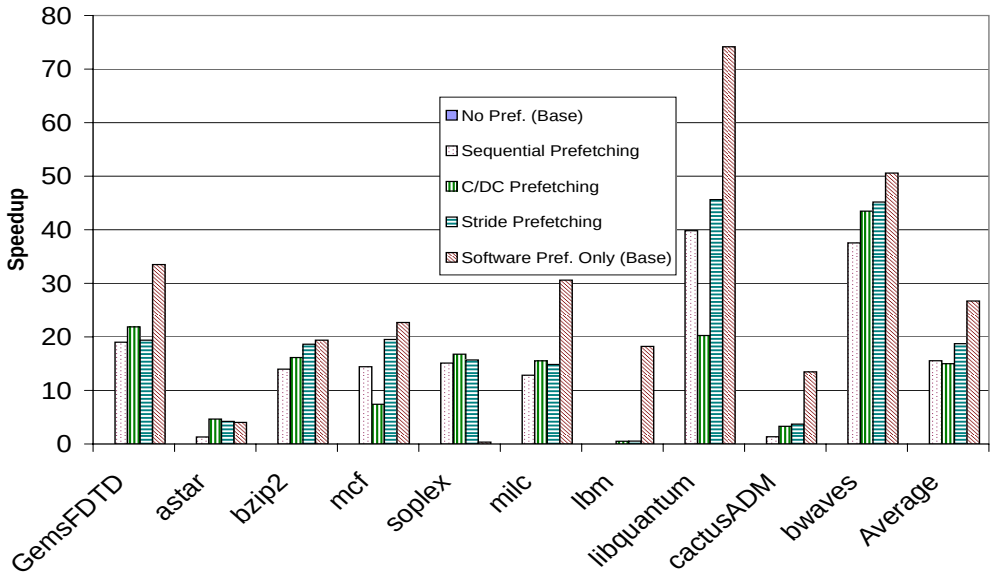

Fig. 2. Performance comparision of hardware only with software only prefetching (Base Tuning)

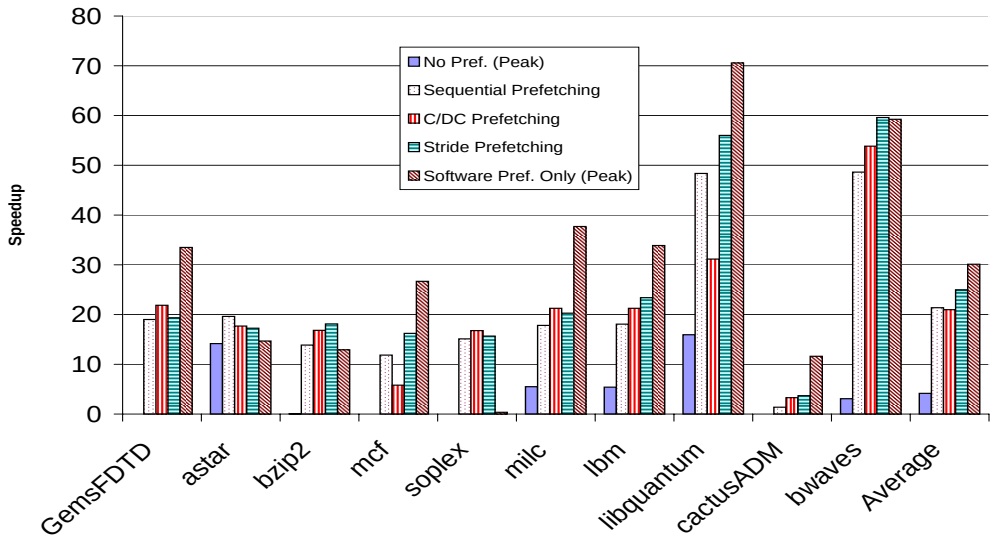

Fig. 3. Performance comparision of hardware only with software only prefetching (Peak Tuning)

Figures 2 and 3 compares the speedup obtained with hardware only prefetching schemes for base and peak tuning respectively with the speedup obtained from the respective software only prefetching schemes. As one might expect, hardware prefetching schemes provide a clear performance benefit over a system with all

prefetching turned off. However, it is surprising to observe that the software only prefetching scheme outperforms all of the hardware based prefetchers for the majority of benchmarks in both charts. We see this phenomenon in the benchmarks GemsFDTD, mcf, milc, lbm and cactusADM in the Figure 3 (peak tuning). The difference is more pronounced in Figure 2 (base tuning) where software only prefetching outperforms the hardware prefetchers in almost all cases. The reader may be wondering if and why these results are different from the ones presented in Figure 1, where base prefetch tuning hurt performance in some benchmarks. The differences are because of variations between the real system we used in the previous section and the simulated one in this section and also the subset of benchmarks evaluated. The benchmarks selected in this section are a prefetcher friendly subset. Our goal in this section is to compare the prefetch ability of the different hardware schemes and software prefetching while the goal in the previous section was to discuss the impact of expert tuning on all benchmarks.

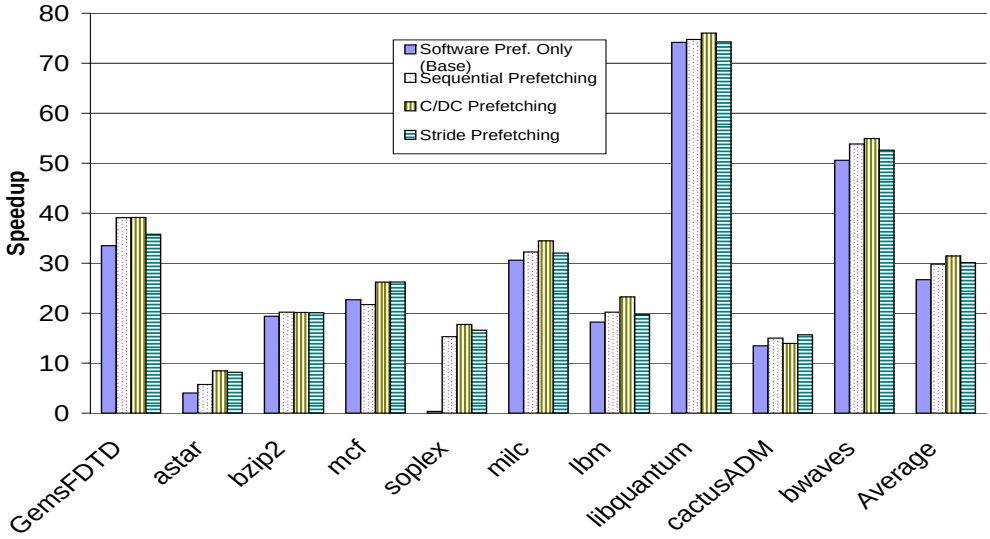

Fig. 4. Performance of Hardware Prefetchers with Software Prefetching (Base Tuning)

6.3. *Performance of Hardware Prefetching with Software Prefetching Support*

Figures 4 and 5 show the speedup of the hardware prefetchers with software prefetching turned on for base and peak settings respectively. In the base setting, the benchmarks are compiled with the generic options for both the software prefetching and non-prefetching components of the programs (i.e, no expert tuning is available). For the benchmarks in peak setting, all aspects of the program, including the soft-

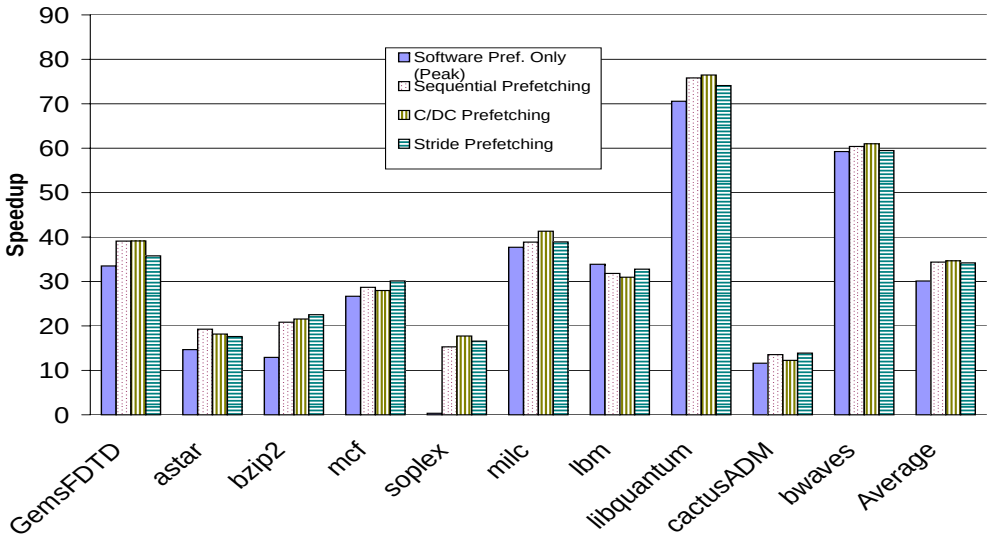

Fig. 5. Performance of Hardware Prefetchers with Software Prefetching (Peak Tuning)

ware prefetching component are compiled with expert tuning. Unlike the results in the previous section, we see that all of the hardware prefetchers now outperform the software only prefetching schemes for almost all benchmarks (the only exception is the Sequential prefetcher for mcf). The C/DC prefetcher is the best prefetcher on average, and it outperforms the software only prefetching scheme by approximately 5% in both figures, a relatively modest improvement. The results here and in the previous subsection demonstrate that consistent and effective benefits from hardware prefetching are only possible with software prefetching support, although the improvement over software only prefetching is relatively modest for many benchmarks and on average.

Figure 6 compares the best pure software solution (expertly tuned software prefetching) with combined hardware and software prefetching compiled under different settings. For each hardware prefetching bar, we select the best individual hardware/software prefetcher for the given compilation setting. The combined prefetcher under base settings represents a situation where no effort is made to tune the compiler settings, either for the prefetcher or for the other components of the program. When we compare it to software only prefetching on an individual benchmark basis, this configuration significantly outperforms software only in some cases (for example, soplex) while it significantly underperforms in other cases (for example, lbm). On average, it outperforms software only even though the software prefetcher has both the advantage of expert tuning in its prefetching component and the additional advantage of expert tuning in its non-prefetching component. In other words, the combined prefetcher under base settings "jumps two hurdles" to catch up with and outperform software only prefetching on average. The combined

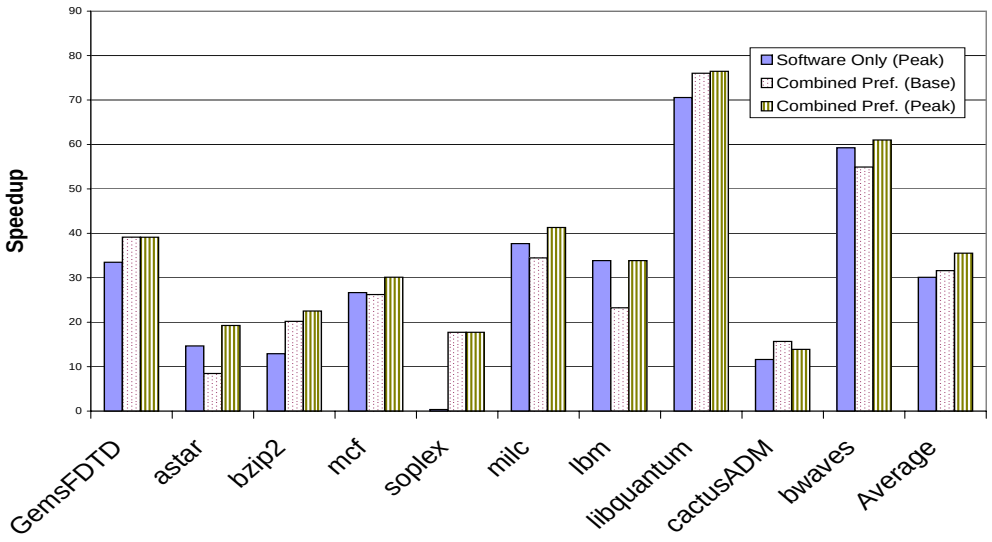

Fig. 6. Comparison of peak software prefetching with combined schemes

prefetcher under peak settings outperforms, or atleast matches the best software solution for all benchmarks. This demonstrates that the best overall performance is obtained by combining expert software tuning with a good hardware prefetcher and that hardware and software prefetching are complimentary approaches.

7. Analyzing the Performance Results

An important question is whether the observed differences in prefetcher performance are due to inherent strengths and weaknesses or merely to factors that can be improved with further tuning and development. In particular, does software prefetch benefit from the ability of the compiler to perform extensive analysis, and does hardware prefetch benefit from adaptation to run-time conditions?

First, comparing hardware-only to software-only prefetch, simulations of the base and peak-tuned benchmarks and the three hardware prefetchers show that hardware prefetch is superior for two benchmarks, bzip2 (peak tuning) and soplex, is about as good for three more, astar, bzip (base tuning), and mcf (base tuning), but hardware prefetch is much worse than software for two benchmarks, cactus and lbm, and for the remaining benchmarks hardware has about half the speedup of software. These results can be seen as disappointing for hardware prefetch advocates, the analyses below show that the hardware schemes could be improved to match software, ignoring cost and other tradeoffs.

Measurements of prefetch coverage and accuracy give a sense of the mechanisms' abilities. As plotted here, prefetch coverage is the number of used prefetches (on-time or late) divided by used prefetches plus L2 cache misses. Accuracy is the number of used prefetches divided by the total number of surviving (not dropped)

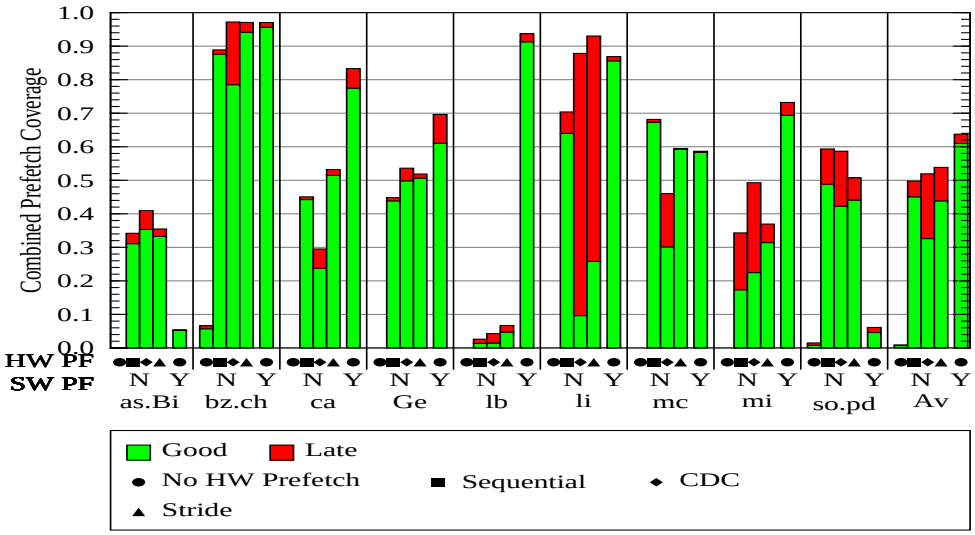

Fig. 7. Prefetch Coverage

prefetch requests. Figure 8 shows the coverage for selected benchmarks on the different systems. (Note that there is still some software prefetch in the no-software-prefetch binaries due to prefetches in library code.) The first four bars per benchmark are for configurations with no software prefetching (indicated by SW PF - N) and the last bar is a configuration with software prefetching (indicated by SW PF - Y). The lower segment of each bar counts on-time prefetches, the upper segment counts late prefetches.

If overheads, such as congestion and cache pollution are low, coverage should match performance. We can get performance data by looking at figure 3 and figure 5. For the tested systems, coverage matches performance for all but three benchmarks, the exceptions are astar, cactusADM, and mcf. Note that for libquantum coverage is high but much of that is late coverage and so performance suffers. This can be seen by comparing C/DC to Stride. The C/DC prefetcher has less than half the number of timely prefetches, and as such its speedup is relatively lower.

Prefetch accuracy is plotted in Figure 9. Notice that hardware prefetch accuracy exceeds software prefetch accuracy for every benchmark, except for the aggressive sequential prefetcher. This is what one would expect, since there was no mechanism to throttle ineffective software prefetch, whereas hardware prefetch requires misses or prefetched data hits to continue prefetching.

As mentioned, coverage is a key performance factor, and so the reasons for the differences in coverage will be analyzed for selected benchmarks. The analysis is performed by finding frequently missing (or prefetched) loads and determining why they weren't benefiting, or benefiting more, from prefetch. The simulator provides a table of the loads that most often hit prefetched data, and simulator output can

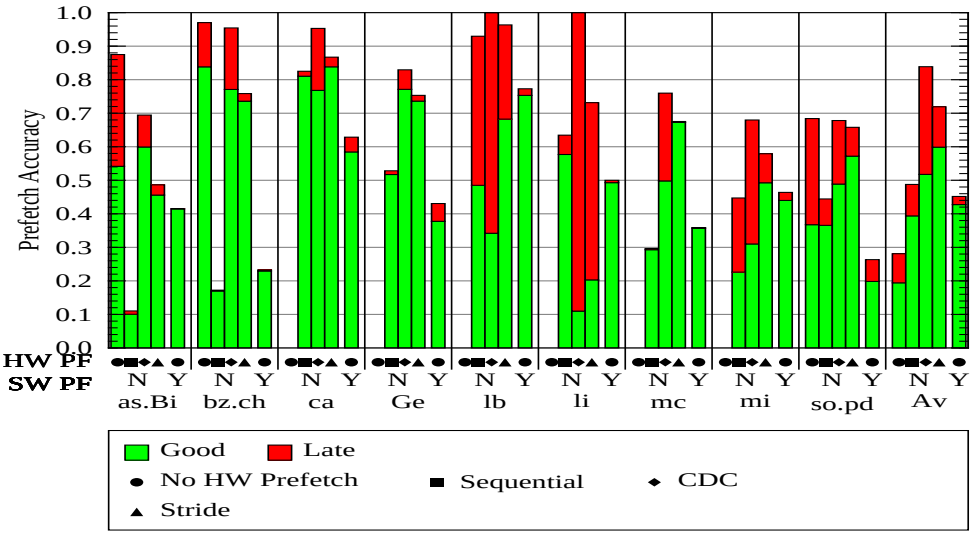

Fig. 8. Prefetch Accuracy

also be visualized showing cycle-by-cycle instruction events. Many of the tested benchmarks have small kernels, and so such hand analysis can explain a substantial amount of their observed performance.

Hardware prefetch is able to squeeze a respectable 20% speedup out of soplex, while software prefetch achieves almost none, despite use of the prefetch flag. This was true also in the peak runs, which had the benefit of profiling. Soplex is a C++ benchmark that includes a mixture of direct strided and indirect (through an index array) array access. The hardware prefetch schemes were able to prefetch strided accesses that the compiler was too shy to prefetch for, such as a negative loop index increment in SPxSteepPR::selectLeaveX. Here hardware prefetch has a natural advantage, since it can exploit patterns not easily discoverable from the code.

At the other end of the spectrum hardware prefetch does much worse on cactus and lbm. In both of these programs the loads that miss the cache access data using a relatively stable stride, something both the stride predictor and CDC should have captured. For lbm the stride was a constant 160 (in performStreamCollide), but because loads accessed a base address at varying offsets, the stride predictor was confused. A PC-indexed stride predictor might have done better. Here, hardware prefetch could be improved by better filtering the access stream. Such improvements could approach software on these codes, but at the cost of complexity that software prefetch does not share.

Five benchmarks had hardware prefetch speedup about half that of software prefetch. Of these, mcf peak, milc, and libquantum were undermined by prefetch lateness. This was especially large in libquantum. The base hardware prefetchers

had a modest degree of four, and would not prefetch further. Limiting the distance reduces damaging effects but results in lateness when the targeted load is executed soon. More elaborate adaptive hardware prefetch would close the gap for these benchmarks by adjusting degree based on, say, the number of damaging prefetches.

Benchmarks Gems, lbm (peak), and to a smaller extent mcf (peak) have lower coverage in hardware than in software prefetch mechanisms, even counting late prefetches. Gems has a substantial amount of stride and sequential access patterns, however over a short interval of time loads accessing different parts of the address space can miss, making it more difficult for stride and CDC to train, especially because loads don't miss consistently. As in the other cases, hardware could potentially close the gap with more elaborate hardware to filter the access stream.

Many of the benchmarks with lower coverage also seem to suffer from miss irregularity. That is, hardware prefetch of an access pattern can be stopped if one item is cached. This too is potentially fixable, by having a stream design, like the stride predictor used here, monitor all L1 accesses.

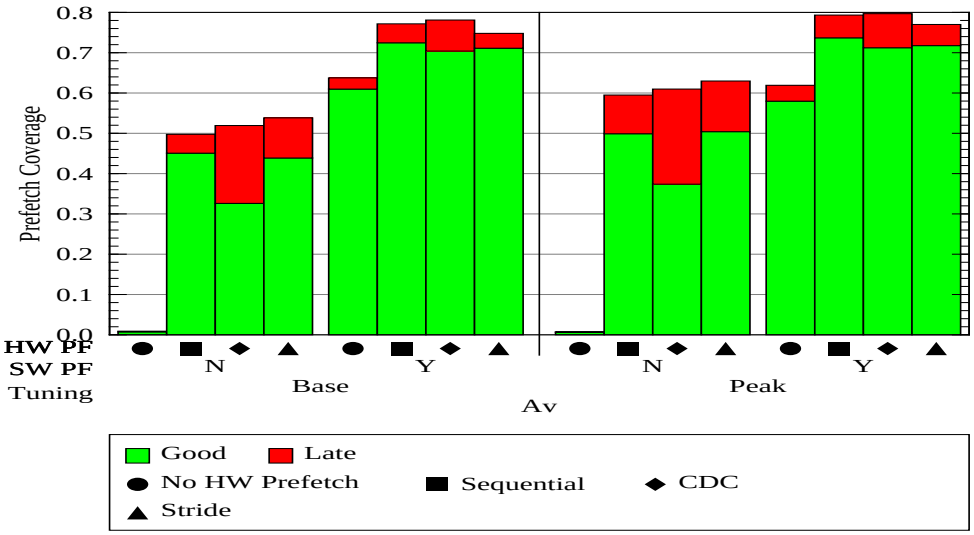

Fig. 9. Average coverage for all tuning combinations

7.1. *Summary*

Based on the software vs. hardware prefetch comparisons, it appears that when the compiler is able to prefetch, it can prefetch well. In only one case was hardware much better, though the selection of benchmarks might be biased towards compiler analyzability. Hardware could conceivably catch up, if there were a reason to do so. Turning on prefetch in the peak-with-base prefetch experiments only slows down a

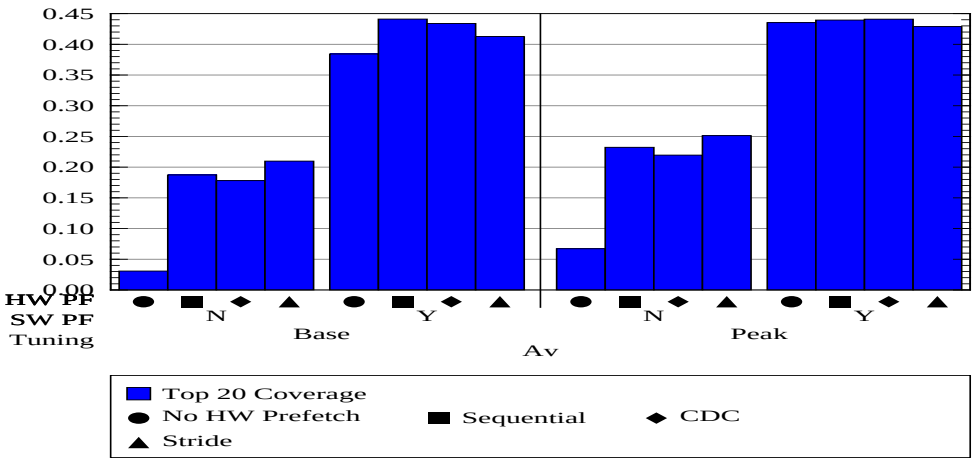

Fig. 10. Average coverage for 20 most benefited loads

few benchmarks. However as mentioned earlier, even if software prefetches do not hurt performance, they still result in wasted energy.

When software and hardware prefetch are combined there are further small performance gains (average improvement of 5.5%). Figure 10 shows average coverage data for all combinations of base and peak tuning, and software and hardware prefetch. We observe that software only prefetching (the fifth bar per tuning) gets much better coverage than any of the hardware only schemes (the first four bars) for base tuning and roughly matches coverage for the peak tuning. For either tuning, adding hardware to software prefetch increases coverage, as one might expect.

Figure 11 shows the coverage of only the 20 most benefited loads. Combined prefetching achieves much higher coverage than hardware only prefetching. On the other hand, when comparing software only prefetching to the combined schemes, there is almost no increase in top-20 coverage for the combined schemes, suggesting that for peak builds (which benefit from profiling) software prefetch is perfectly prefetching some of the most important loads. This figure shows each configuration at its best, meaning coverage of the 20 loads with the most prefetch hits. Since software gets significantly better coverage than hardware prefetching, this data indicates that software prefetching helps an individual load much more than hardware prefetch. However if we go back to figure 10, we notice that the gap in coverage is lower than that for the top 20 most benefited loads. Hence we can conclude that while software prefetching is more beneficial on an individual load basis, hardware atleast partly compensates for this by identifying more loads for prefetch.

8. Conclusions

This paper considers the interaction between state of the software prefetching techniques used in a modern compiler with hardware based prefetching. We evaluate

the ability of compilers to achieve good results without expertise by using binaries with no prefetch, generic one-flag prefetch and expert tuning of the prefetcher. Expert prefetching avoids minor slowdown in a few benchmarks and also provides substantial speedup in two or three benchmarks. By omitting unneeded prefetch instructions, it also avoids the waste in energy that the processing of such instructions would require. We evaluate several hardware only schemes and show that these schemes underperform an expertly tuned software prefetching scheme. Combined hardware/software prefetching is a significant improvement over hardware only prefetching, but only modestly outperforms software only prefetching. The best results are obtained when peak software tuning is combined with a good hardware prefetcher, indicating that hardware and software prefetching compliment each other. We look into the reasons for the obtained performance by analyzing accuracy and coverage data and observe that they correlate well with performance. While results for hardware prefetching seem disappointing, our analysis reveals that the basic hardware prefetchers could be improved, but at increased cost and other tradeoffs. By comparing average coverage data for all prefetched loads with data for just the top 20 loads, we conclude that software prefetching is more beneficial on an individual load basis, but hardware prefetching is able to identify more loads for prefetch.

References

1. W.F. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. *In Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 301-312, Jan 2001.
2. N.P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990.
3. A. Smith. Sequential program prefetching in memory hierarchies. *IEEE Transactions on Computers*, December 1978.
4. J-L. Baer and T.-F Chen. Effective hardware based data prefetching for high performance processors. *IEEE Transactions on Computers*, 1995.
5. J.W.C. Fu and J.H. Patel. Stride directed prefetching in scalar processors. *25th Annual Symposium on Microarchitecture*, 1992.
6. S. Palacharla and R. Kessler. Evaluating stream buffers as a secondary cache replacement. *International Symposium on Microarchitecture*, 1994.
7. D. Joseph and D. Grunwald. Prefetching using markov predictors. *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.
8. K. Nesbit, A. Dhodapkar, and J. Smith. AC/DC: An adaptive data cache prefetcher. In *PACT*, 2004.
9. K. Nesbit and J. Smith. Prefetching with a global history buffer. *HPCA*, 2004.
10. F. Dahlgreen, M. Dubois and P Stenstrom. Sequential Hardware prefetching in shared memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*,1995
11. Ibrahim Hur and Calvin Lin. Memory Prefetching Using Adaptive Stream Detection. *International Symposium on Microarchitecture*, 2006.
12. Luis M. Ramos, José Luis Briz1, Pablo E. Ibáñez and Víctor Viñals. Low-Cost Adaptive Data Prefetching. *Lecture Notes in Computer Science. Springer Berlin / Heidel-*

berg, 2008.
13. S. Srinath, O. Multu, H. Kim, Y. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. *HPCA*, 2007.
14. D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. *In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40-52, Apr. 1991.
15. S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. *In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228- 239, Oct. 1998.
16. T. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. *In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62-73, Oct. 1992.
17. A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. *In Proceedings of the 18th International Symposium on Computer Architecture*, pages 43-53, May 1991.
18. P. Ranganathan, V.S. Pai, H. Abdel-Shafi and S.V. Adve. The interaction of software prefetching with ILP Processors in Shared Memory Systems. *ISCA*, 1997.
19. S.P. Vanderweil and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 2000.
20. 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*, Jun., 2003.
21. E. H. Gornish and A. V. Veidenbaum. An integrated hardware/ software scheme for shared-memory multiprocessors. *In Proceedings of the 1994 International Conference on Parallel Processing*, pages 281-284, 1994.
22. M. Karlsson, F. Dahlgren, and P. Sternstrom. A prefetching technique for irregular accesses to linked data structures. *In Sixth International Symposium on High Performance Computer Architecture*, page 206, Jan. 2000.
23. J. Lu , H. Chen , R. Fu , W. Hsu , B. Othmer , P. Yew , D. Chen. The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System. *In Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, p.180, 2003.
24. S.W. Son, M. Kandemir, M. Karakoy and D. Chakrabartil. A compiler directed data prefetching scheme for chip multiprocessors. *ACM PPoPP*, 2009.
25. A. Badawy, A. Aggarwal, D. Yeung, and C. Tseng. The Efficacy of Software Prefetching and Locality Optimizations on Future Memory Systems. *Journal of Instruction-Level Parallelism*, Vol. 6, July 2004.
26. SPARC Enterprise Architecture, M5000 Servers. *Fujitsu White Papers*, 2009.
27. SPEC CPU2006: Read Me First, Version 1.1. *Standard Performance Evaluation Corporation (SPEC)*, June 17, 2008. http://www.spec.org/cpu2006/Docs/readme1st.html
28. T. Horel and G. Lauterbach. UltraSPARC-III: Designing Third- Generation 64-Bit Performance. *IEEE Micro*, vol. 19, no. 3, pp. 73- 85, May-June 1999.
29. Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM Reference Manual Version 1.0. *Rice University Dept. of Electrical and Computer Engineering*, Technical Report 9705, August 1997.
30. D.L. Weaver and T. Germond. The SPARC Architecture Manual, Version 9. *Sparc International and PTR Prentice Hall, Englewood Cliffs, NJ*, 1994.