Journal of Circuits, Systems, and Computers
Vol. 21, No. 2 (2012) 1240006 (23 pages)

DOI: 10.1142/S0218126612400063

World Scientific
www.worldscientific.com

EFFECTIVENESS OF COMPILER-DIRECTED PREFETCHING ON DATA MINING BENCHMARKS*

RAGAVENDRA NATARAJAN[†,§],
VINEETH MEKKAT[†,¶], WEI-CHUNG HSU[‡,‖]
and ANTONIA ZHAI[†,**]

[†]*Computer Science and Engineering,*
University of Minnesota,
Minneapolis, Minnesota 55455, USA

[‡]*Computer Science, National Chiao*
Tung University,
Hsinchu, Taiwan
[§]*natar@cs.umn.edu*
[¶]*mekkat@cs.umn.edu*
[‖]*hsu@cs.nctu.edu.tw*
[**]*zhai@cs.umn.edu*

Received 25 July 2010
Accepted 23 January 2011

For today's increasingly power-constrained multicore systems, integrating simpler and more energy-efficient in-order cores becomes attractive. However, since in-order processors lack complex hardware support for tolerating long-latency memory accesses, developing compiler technologies to hide such latencies becomes critical. Compiler-directed prefetching has been demonstrated effective on some applications. On the application side, a large class of data centric applications has emerged to explore the underlying properties of the explosively growing data. These applications, in contrast to traditional benchmarks, are characterized by substantial thread-level parallelism, complex and unpredictable control flow, as well as intensive and irregular memory access patterns. These applications are expected to be the dominating workloads on future microprocessors. Thus, in this paper, we investigated the effectiveness of compiler-directed prefetching on data mining applications in in-order multicore systems. Our study reveals that although properly inserted prefetch instructions can often effectively reduce memory access latencies for data mining applications, the compiler is not always able to exploit this potential. Compiler-directed prefetching can become inefficient in the presence of complex control flow and memory access patterns; and architecture dependent behaviors. The integration of multithreaded execution onto a single die makes it even more difficult for the compiler to insert prefetch instructions, since optimizations that are effective for single-threaded execution may or may not be effective in multithreaded execution. Thus, compiler-directed prefetching must be judiciously deployed to avoid creating performance bottlenecks that

*This paper was recommended by Regional Editor Gayatri Mehta.
¶Corresponding author.

otherwise do not exist. Our experiences suggest that dynamic performance tuning techniques that adjust to the behaviors of a program can potentially facilitate the deployment of aggressive optimizations in data mining applications.

Keywords: Multicore; data mining; prefetching; compilers; optimization.

1. Introduction

At the turn of the century, traditional out-of-order uniprocessor designers were facing a unique set of challenges that are often identified as the ILP wall, the memory wall, and the power wall. These barriers, in particular the power wall, eventually brought the exponential increase in clock speed to a halt. After the stabilization in the operating frequency, replicating cores for creating chip-multiprocessors became the *de-facto* method for achieving performance improvement. However, as the number of cores integrated on to a single die increases, it can potentially become impossible to power up the entire die.[1,2] Furthermore, increasing popularity of mobile computing devices places an even more strigent budget on power consumption. Thus, the integration of simple in-order cores becomes attractive for multicore architectures.[3]

As the performance gap between memory and processor widens, the memory wall has become one of the key hindrances in achieving high performance in modern microprocessors. The emergence of multicore and multithreaded processors further complicate the situation: the underlying memory hierarchy must not only serve memory requests in a timely manner, but also satisfy the increasing bandwidth requirements and the effects of interaction between threads for shared data. Diverse optimization techniques, both hardware- and software-based, have been proposed to overcome this gap. Prefetching has been demonstrated effective for bringing data from memory into the cache before they are used, and thus is able to shorten the effective memory access latency. Both hardware[4–7] and compiler[8–11] optimizations have been investigated and implemented for modern microprocessors. While out-of-order processors can tolerate some memory access latencies by issuing instructions out-of-order, prefetching assumes even greater significance for in-order processors. Compiler-based techniques can perform aggressive optimizations across large segments of codes, even when they are spread across multiple procedures or files. It is also possible for compilers, taking advantage of available profile information, to break ambiguous control/data dependencies and schedule instructions even more aggressively. As a result, compiler-directed prefetching can potentially tolerate long memory access latencies and complex data structures by scheduling prefetch instructions appropriately. However, compiler optimizations must rely on static estimates of runtime behaviors, and it is difficult to verify the effectiveness of compiler optimizations across all applications on all inputs. It is possible for compilers to make poor optimization decisions, and degrade performance. This possible

performance degradation often hinders the deployment of aggressive optimizations. The increasing popularity of multithreaded execution further exacerbates the situation as optimizations that are effective for single-threaded execution may or may not be effective for multithread execution.

Explosive growth in the availability of various kinds of data in both commercial and scientific domains have resulted in an unprecedented need for developing novel data-driven, knowledge discovery techniques. Data mining[12,13] is one such application. Researchers from both academia[14] and industry[15] have recognized that the challenges of data mining applications will shape the future of multicore processor and parallelizing compiler designs. There have been numerous studies[16–19] on the performance characteristics of data mining applications. These works have pointed out that memory hierarchy performance, especially that of the last level cache, plays an important role in the performance characteristics of such applications. Hence, compiler optimizations that aim to improve memory hierarchy performance can potentially benefit data mining applications. However, many data mining applications contain irregular data structures, such as *hash-tree* and *hash-table*, and complex control flow, which can potentially make it difficult for compilers to deploy aggressive optimizations. Thus, this paper investigates the effectiveness of compiler optimizations on these emerging workloads, especially prefetching techniques that target memory hierarchy performance.

In our study, we quantify the impact of memory hierarchy performance on data mining applications; identify memory intensive operations algorithmically and determine whether compiler-directed prefetching is effective on these data structures both in single-threaded and multithreaded executions. We have found that:

- the performance impact of compiler-directed prefetching on data mining applications is unpredictable. Furthermore, the effectiveness of static prefetching in the same code segments can change over time.
- prefetching at the proper granularity is key. Due to complex control flow and irregular access patterns, prefetching at the inner-most loop level is often inadequate. Thus, new algorithms must be developed to identify the proper granularity for prefetching.
- on multicore systems, resource contention and thread-interaction can influence the effectiveness of compiler optimization. Hence, dynamically tuning compiler optimization deployment can potentially out perform static only approaches.

The rest of the paper is organized as follows: Sec. 2 details our experimental infrastructure and briefly describes the various data mining algorithms; Sec. 3 investigates the major characteristics of data mining applications and compares them with SPEC Integer applications; Sec. 4 analyzes cache performance of

NU-MineBench; Secs. 5 and 6 evaluate the effectiveness of compiler-directed prefetching on data mining applications in both single-threaded and multithreaded execution modes on multicore architectures, respectively; Sec. 7 presents the effect of compiler-directed prefetching on the scalability of data mining applications; Sec. 8 discusses related work. Finally, we present our conclusions in Sec. 9.

2. Evaluation Infrastructure

We study the performance characteristics of data mining applications using the NU-MineBench[20] benchmark suite. We evaluate 13 out of 17 benchmarks in this suite, omitting Bayesian, Birch, SNP, and GeneNet due to difficulties encountered in compilation. All benchmarks are written in `C/C++`, and all, with the exception of AFI, GETI, and ECLAT, are parallelized with OpenMP directives. The benchmarks are compiled with `-O3` using the Intel `C/C++` compiler version 11.0.

We choose Intel® Itanium® [21] as the in-order architecture processor for our study for two main reasons. First, Itanium, arguably, relies on sophisticated compiler optimizations to achieve its performance, thus making it the proper platform for evaluating the effectiveness of compiler-directed prefetching. Second, Itanium contains a rich set of hardware performance counters that enable us to study performance characteristics without cumbersome simulations or intrusive instrumentations. These applications are evaluated on an eight-core Itanium-based CMP machine with dual-core Intel® Itanium® 2 processors running at 1.6 GHz. Each of the eight cores has a 16 KB L1I, a 16 KB L1D cache, a 1 MB L2I, a 256 KB L2D cache, and a 9 MB unified private last-level cache. Although our analysis is derived from a system utilizing private last-level cache, we include discussions on the potential impact of shared last-level cache systems in appropriate sections.

The PerfMon[22] library manages the hardware performance counters. We use Intel VTune Performance Analyzer tool to identify *hot-spots* in the programs and to drill-down to assembly level. Stalls in the Itanium pipeline back-end are divided into five mutually exclusive categories. Most of the stalls caused by execution units, waiting for operands, are due to cache misses. We refer to them as `Cache` stalls in our figures. Stalls due to recirculation of data access requests from L1D due to either TLB or L2D OzQ overflow are referred to here as `Recirculation` stalls. `Flush` corresponds to stalls caused by pipeline flushes. In our case, almost all such stalls are caused by branch mispredictions. `RSE` corresponds to stalls caused by the Register Stack Engine, which is negligible in all applications. `FE` corresponds to stalls caused by the pipeline front-end. `Pin`,[23] a binary instrumentation tool, is used to collect dynamic instruction profile data.

In this work, we analyze data mining applications from six different categories which are listed in Table 1. A detailed description of these benchmarks is provided by Narayanan *et al.*[20]

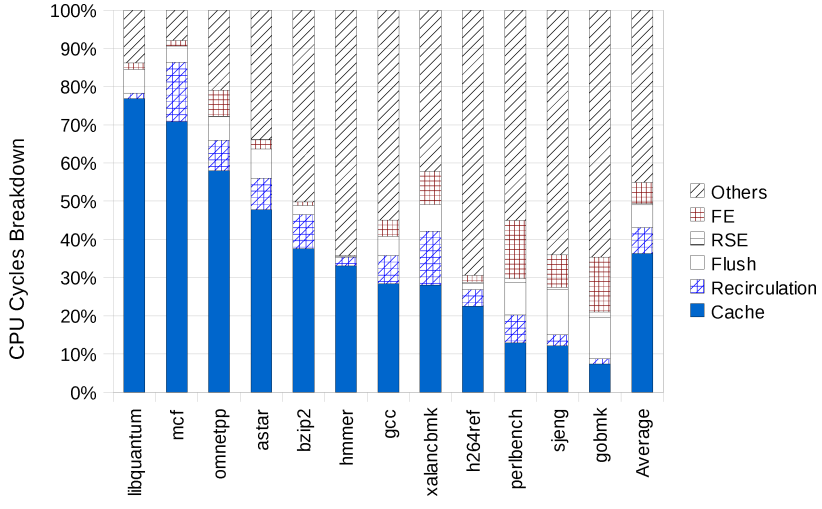
Table 1. NU-MineBench categories and applications studied.

Category	Applications
Association Rule Mining	Apriori, Utility Mine and ECLAT
Error Tolerant Itemset Mining	AFI and GETI
Classification	ScalParC, Rsearch and SVM-RFE
Clustering	K-means, Fuzzy K-means and HOP
Structure Learning	SEMPHY
Optimization	PLSA

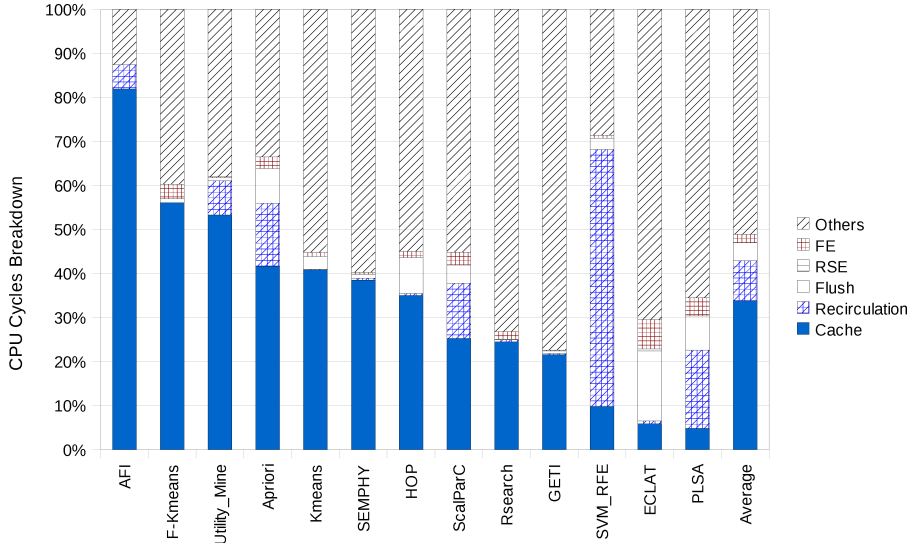
3. Characteristics of NU-MineBench

In our prior work,[19] we examined the instruction distribution, memory hierarchy usage and stall distribution in order to study the basic characteristics of data mining applications. We used SPEC CPU[24] for comparison as SPEC has been widely accepted by computer architects as the chosen benchmark suite for measuring CPU performance. We observed that, in terms of dynamic instruction mix, the two benchmark suites show similar trends, but with the following distinctions: SPEC Integer applications have 54% more store instructions than NU-MineBench programs, but have 16% fewer load instructions. Data mining applications typically use the input data to build temporary data structures. These data structures are then traversed many times with a small number of updates. This access pattern leads to more loads but fewer stores. SPEC programs also have 45% more branch instructions as compared to NU-MineBench applications. Having fewer store and branch instructions potentially gives the compiler more freedom to schedule instructions, since branches and stores are often the sources of ambiguous control and data dependencies. In terms of cache miss rate, We observed that L2 cache misses per 1000 instructions is 34% higher for SPEC Integer; and L3 cache miss per 1000 instructions is the same for both benchmark suites.

Figure 1 shows the stall cycle distribution for the SPEC Integer and the NU-MineBench benchmark suites are similar despite the differences in instruction mix. Eliminating stalls due to cache misses is key to better performance as nine out of 13 data mining benchmarks spend more than 40% time stalling on cache misses. Many data mining applications process large input sets, accesses to which typically exhibit good locality, and thus does not cause significant stalls. However, at runtime, data mining benchmarks build large, complex auxiliary data structures, such as hash trees, to keep track of intermediate states. These structures are often irregular, and accesses to these cause a large number of cache misses. Hence, for data mining applications, compiler-directed prefetching is an important optimization. In this paper, we investigate the effectiveness of compiler-directed prefetching for data mining workloads on in-order processors. In Sec. 4, we revisit the key observations on memory characteristics of data mining benchmarks as discussed in Mekkat *et al.*[19] and in Secs. 5 and 6, we study the effect of compiler-directed prefetching on data mining benchmarks running in single-thread and multithread modes, respectively.



(a)



(b)

Fig. 1. CPU cycles breakdown comparison for (a) SPEC CPU2006 Integer Benchmarks and (b) NU-MineBench Programs.

4. Memory Hierarchy Performance

Improving cache performance is key for many data mining applications, since nine out of the 13 benchmarks we examined spent over 40% of total execution cycles stalling as a result of cache misses. Our analysis reveals that, although many data

mining applications process large input sets, accesses to the input data usually exhibit spatial locality and do not suffer from high cache miss rate. However, these applications often construct large and irregular auxiliary data structures, such as hash trees.[13] Accesses to these data structures exhibit poor locality due to indirect accesses and pointers.

Although ScalParC and SVM-RFE are both classification benchmarks, they show significantly different behaviors. ScalParC stalls on 57% of CPU cycles due to cache misses. The program generates a large, tree-based model built on the training (input) data and at each node, a hash table is used. The hash table contains millions of entries, and does not fit in the cache. By examining the addresses accessed by the program it is clear that accesses to the hash table are irregular and this leads to frequent cache misses. The compiler does not insert prefetch instructions for this program as the elements of the hash table accessed cannot be predicted. SVM-RFE, another classification benchmark, shows one of the highest stall percentages. It stalls for 69% of total CPU Cycles, which is almost entirely due to recirculation stalls. The program kernel makes use of Math Kernel Library, to compute vector dot product, where it stalls most of the time. Since the stalls occur in the library and not in the user code, we did not investigate this further.

Apriori and Utility Mine, the two association rule mining applications in NU-MineBench, show similar execution cycle breakdowns. Apriori stalls for 66% of total cycles and Utility Mine stalls for 62% of total cycles, almost all of which are caused due to cache misses. Both these association rule mining programs use hash trees in their algorithm, which tends to be too large to fit into the cache and exhibit poor locality. Hence, repeated accesses to these hash tree data structures generate a large number of cache misses. AFI, an error tolerant itemset mining algorithm, is very similar to Apriori and finds frequent itemsets in noisy data. Here, compiler is not able to identify the right granularity to insert prefetch instructions. Moving the prefetch instruction manually to the outer loop improves the performance of this application by more than 30%. It is difficult for the compiler to identify this optimization opportunity since the outer loop contains conditional branches.

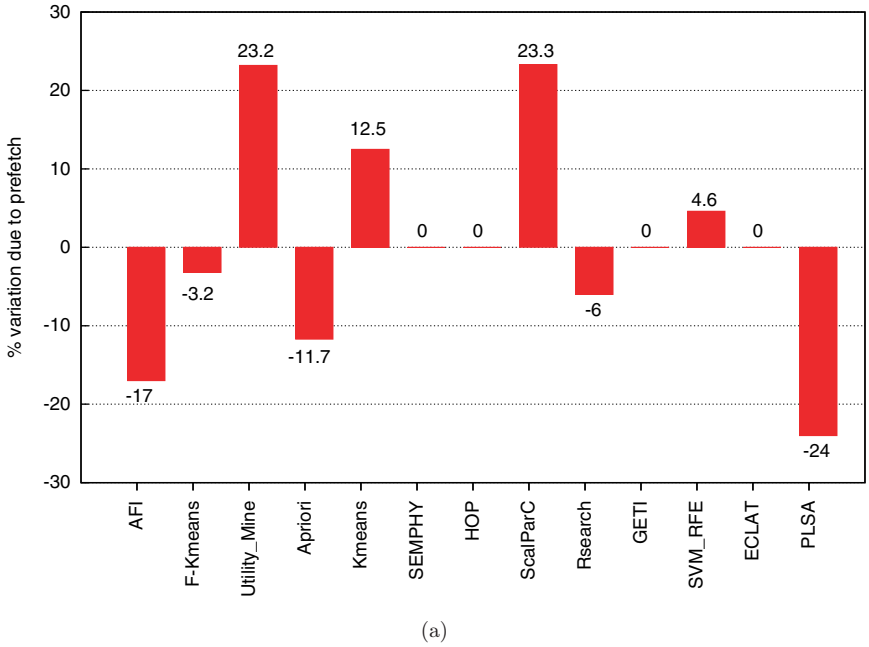
HOP, K-means, and Fuzzy K-means are the three clustering applications in the NU-MineBench benchmark suite. HOP stalls for 45% of total CPU cycles, of which cache misses account for 35% of the CPU cycles. In this application, the instructions were scheduled poorly which could not effectively hide the latency of load instructions. The compiler failed to unroll certain loops which were inside control statements. Since the compiler was not sure if the loop would be executed at runtime, it was conservative and did not unroll these loops. By manually unrolling them, the performance of the program improved by 14%.

5. Prefetching in Single-Threaded Execution

From the previous section, it is clear that the performance of data mining applications, similar to SPEC Integer applications, highly depends on the performance of the

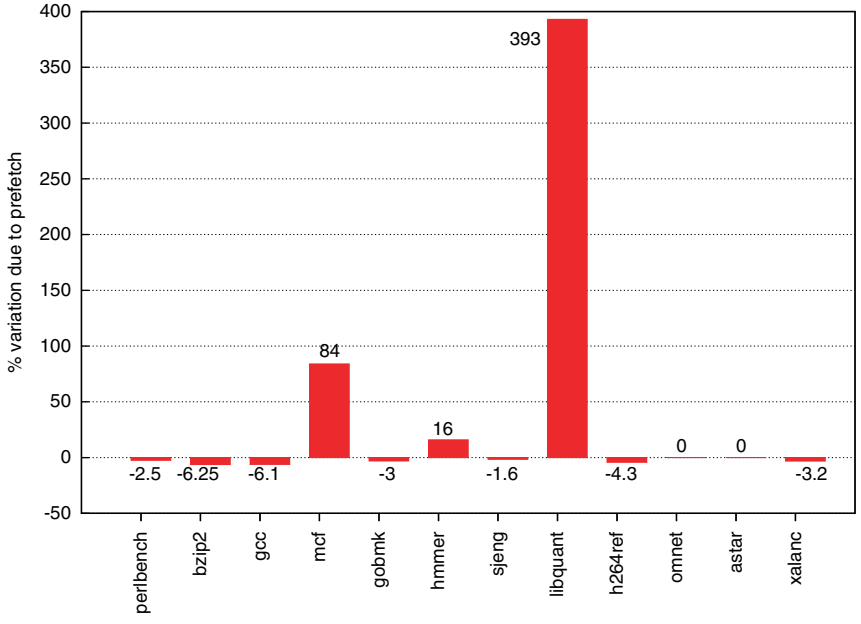
memory hierarchy. A large number of compiler optimizations have been developed for improving memory hierarchy performance. While these optimizations have been demonstrated effective on SPEC benchmarks, it is unclear whether these optimizations can be effective on data mining applications given their complex memory access patterns. One key compiler optimization for improving memory hierarchy performance is prefetching. As mentioned earlier, the impact of compiler-directed prefetches is more significant for in-order processors as they lack the complex hardware mechanism to hide memory access latency and depend on compiler optimizations to achieve performance. In this section, we provide an in-depth analysis of the impact of compiler-directed prefetching on data mining applications executing in single-threaded mode.

Figure 2(a) shows the speedup achieved by compiler-directed prefetching on NU-MineBench benchmarks in percentage. These applications are compiled with `-O3 -opt-prefetch`, and the comparison baseline has prefetching disabled (`-O3 -no-opt-prefetch`). Bars above zero indicate that compiler-directed prefetching is able to improve performance, bars below zero indicate otherwise. We also include the speedup for SPEC benchmarks in Fig. 2(b) for comparison. For SPEC, while a few benchmarks, MCF, LIBQUANTUM, and HMMER, significantly benefit from compiler-directed prefetching, most benchmarks are not affected. It is worth pointing out that none of the SPEC benchmarks suffer significant performance degradation. The



(a)

Fig. 2. Impact of compiler inserted prefetch instructions on (a) NU-MineBench Programs and (b) SPEC CPU2006 Integer Benchmarks.



(b)

Fig. 2. (*Continued*)

responses of NU-MineBench benchmarks, however, are dramatic: while the performance of some benchmarks improves, equal number of benchmarks show significant performance degradation. Hence, compiler-directed prefetching, although effective for SPEC benchmarks, cannot be blindly applied to data mining applications.

Prefetch instructions can potentially reduce stalls due to increased load latency by bringing data into the cache before they are used. However, if not properly deployed, they can also cause significant performance degradation. First, compiler-directed prefetch instructions and instructions used to calculate the prefetching addresses take up issue slots, and thus increase both static and dynamic instruction counts. Second, improperly inserted prefetch instructions can bring useless data into the cache, and even pollute the cache by replacing useful data. This effect can increase cache misses in the application. Finally, prefetching can increase accesses to shared resources, such as the off-chip communication channel, and thus create a bottleneck that would not have existed otherwise. This effect is more evident when the programs are executed in multithreaded modes, and will be discussed in Sec. 7.

While instruction overhead, cache pollution and resource contention are the most common causes for performance degradation due to prefetching, some impacts are less obvious and architecture dependent. We discuss two such cases, where prefetch

instructions themselves introduce stalls in the pipeline. Although not common, prefetch instructions can cause exceptions, such as page faults. In Itanium, the compiler can append prefetch instructions with completers that make the processor either ignore faults made by prefetch instructions, or service them. For a useful prefetch instruction, allowing it to raise a fault can bring data into the cache early. However, a useless prefetch instruction, if allowed to raise a fault, can significantly degrade performance. Hence, aggressive optimizations that can lead to exceptions should be performed only when prefetching is accurate. Prefetch instructions can also stall the pipeline by competing for resources in the processor. Most modern processor have a load buffer where requests to the L2 cache are buffered and issued out-of-order. Prefetch instructions have been known to stall the processor pipeline when the L2 out-of-order load buffer is full in certain architectures.[25] Aggressive compiler-directed prefetching can lead to many prefetch instructions being issued. This can quickly saturate the out-of-order load buffer, thereby stalling the pipeline since new load or prefetch instructions cannot be issued.

Inaccurate prefetch instructions cannot only cause detrimental side effects such as instruction overhead, cache pollution, and resource contention, but also cause pipeline stalls. Thus, the compiler needs to make intelligent choices while inserting them. In the rest of this section, we analyze the reasons for performance degradation of data mining programs due to compiler-directed prefetching.

Figure 2 shows that AFI slows down by 17%, APRIORI by approximately 12% and PLSA by 24% due to compiler directed prefetching. To examine why compiler prefetching degraded the performance in these applications, three different versions of the binaries are compared. The first binary (*binary*1) is produced using the compiler with prefetching enabled (`-O3 -opt-prefetch`), the second binary (*binary*2) is produced without prefetching (`-O3 -opt-no-prefetch`), and the last binary (*binary*3) is produced manually by replacing key prefetch instructions in *binary*1 with `nop` instructions. Hence *binary*3 has the same instruction count as *binary*1, while *binary*2 has considerably fewer instructions due to the absence of prefetch instructions and the corresponding address calculation instructions.

In Table 2, for each benchmark, the first row shows the speedup of *binary*2 and *binary*3 with respect to *binary*1, which is the baseline. The second and third rows show the L2D and L3 misses per 1000 instructions, respectively. In all three programs, *binary*2 always has the best performance, and *binary*3 is a close second. Both *binary*2 and *binary*3 perform considerably better than *binary*1. Although they show the same speedup, *binary*3 has fewer L2D and L3 misses per 1000 instructions than *binary*2 for all the programs. This is because *binary*3 has the same number of L2D and L3 misses, but a higher instruction count that lowers the cache misses per 1000 instructions measurement.

*binary*2 and *binary*3 show similar speedup despite the fact that *binary*3 has more instructions for prefetch address calculation. These numbers reveal that the overhead of prefetch address calculation is negligible for all three programs. Thus, it is the

Table 2. Comparison of the different binaries for AFI, Apriori, and PLSA.

Attribute	*binary1*	*binary2*	*binary3*
(a) AFI			
Speedup	1.00	1.51	1.49
L2D Misses/K inst	27.1	21.23	17.78
L3 Misses/K inst	19.08	9.85	8.15
(b) Apriori			
Speedup	1.00	1.16	1.10
L2D Misses/K inst	18.58	16.01	14.52
L3 Misses/K inst	10.94	7.22	6.82
(c) PLSA			
Speedup	1.00	1.38	1.21
L2D Misses/K inst	0.59	0.73	0.58
L3 Misses/K inst	0.07	0.08	0.07

prefetching effect that causes performance degradation, while instruction overhead is only responsible for a small segment of performance degradation.

Prefetch instructions also account for a large number of L2D and L3 misses in case of AFI and apriori, but do not affect the miss numbers in PLSA. Analyzing the performance monitoring counters in Itanium reveals the cause of the poor performance of each program. In case of AFI and Apriori, the stalls are caused by execution units waiting for operand data to be loaded. Prefetch instructions inserted by the compiler bring in useless data into the cache, and replace useful data resulting in increased L2D and L3 misses.

In case of PLSA, prefetch instructions inserted by the compiler are redundant, and prefetch the same data multiple times. The prefetch instructions do not increase the number of L2D and L3 misses since once the data has been prefetched, subsequent requests to the same location hit in the cache. But the pipeline stalls are caused due to prefetch instructions being recirculated. Aggressive prefetching by the compiler saturates the L2 load buffer and the pipeline stalls since new load and prefetch instructions cannot be issued until there are vacant entries in the load buffer.

6. Prefetching in Multithreaded Execution

On multicore architectures, compiler optimization developers must address the following concerns: are optimizations effective for single-thread execution, effective on multithreaded execution; and are optimized codes equally scalable, as unoptimized codes, when the number of threads increase. Furthermore, it is also unclear how compiler optimizations can affect threads that are explicitly synchronized through barriers and locks. Thus, in this section, we examine the effectiveness of compiler-directed prefetching on multithreaded data mining applications.

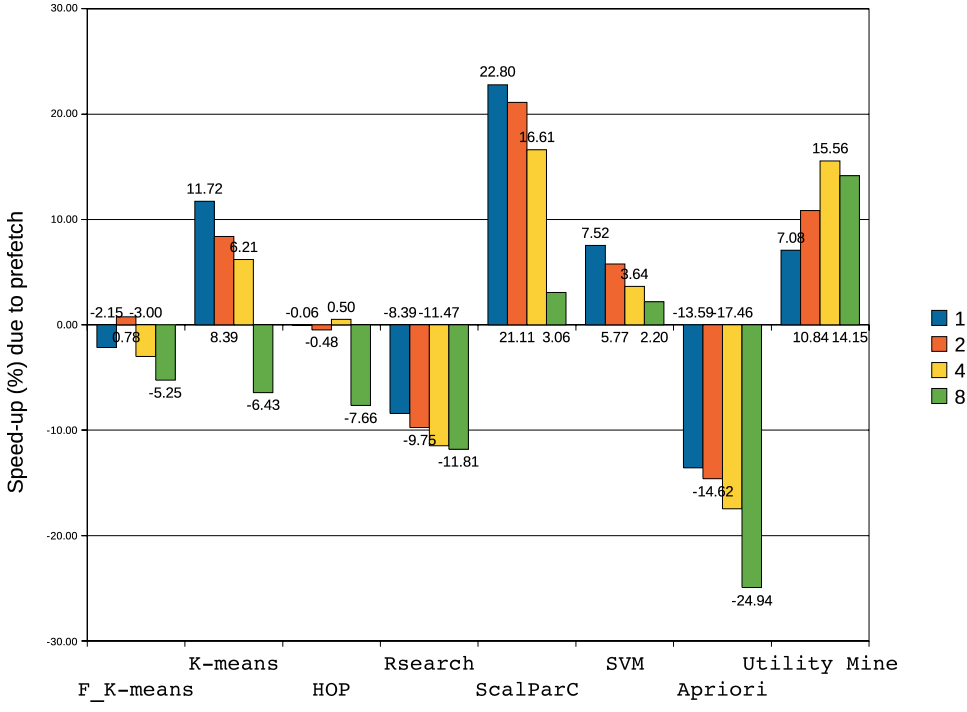


Fig. 3. Effect of compiler-directed prefetching in multithreaded execution.

Figure 3 shows the speedup achieved by compiler-directed prefetching on multithreaded NU-MineBench benchmarks in percentage. Similar to Fig. 2, these applications are compiled at the optimization level -O3 with prefetching enabled (`-opt-prefetch`), and the comparison baseline has -O3 with prefetching disabled (`-no-opt-prefetch`). Each benchmark has four bars, that correspond to the speedup with one, two, four, and eight threads. A positive value on the graph indicates that compiler-directed prefetching is effective, where as, negative value indicates that prefetching is detrimental to performance.

For all benchmarks, with the exception of Utility Mine, compiler-directed prefetching becomes progressively detrimental with increasing number of threads. This phenomenon can be mainly attributed to the competition for shared resources among threads. However, in one isolated case, compiler-directed prefetching becomes progressively more beneficial as the number of threads increase. In this case, prefetching interfered with the explicit synchronization in the application. In this section, we investigate the effectiveness of compiler-directed prefetching in multithreaded execution of data mining benchmarks. Here, we omit the discussion of: SVM-RFE due to poor parallel implementation of the algorithm; Semphy since there are no active compiler-directed prefetching in its critical sections; and also Fuzzy K-means as its behavior is very similar to K-means.

6.1. *Competition for shared resources*

Compiler-directed prefetching is known to increase the utilization of resources such as memory bus bandwidth. As shown in Fig. 3, the increase in resource utilization aggravates with multithreaded execution. We examine two cases, where resource-sharing becomes the bottleneck and compiler-directed prefetching becomes progressively less effective, as the number of threads increase. Figure 4 shows the bus utilization of each benchmark, with and without compiler-directed prefetching, running with one, two, four, and eight threads.

Apriori, as discussed in Sec. 4, suffers performance degradation when compiler-directed prefetching is deployed in sequential execution. The effect of aggressive prefetching becomes more significant as the number of threads increase, as the bus utilization is near saturation when prefetching is enabled and thread count is high.

ScalParC benefits from compiler-directed prefetching at single-thread. However, in multithread mode, its benefit from prefetching drops from 23% for single-threaded execution to 3% for eight threads of execution. As the number of threads increase, the fraction of execution time spent in its most time-consuming function increases. In the code with prefetching, bus utilization is 21%, 37%, 50%, and 55% at this function for thread numbers one, two, four and eight, where as, code without prefetching has 13%, 24%, 39%, and 53%, respectively. At lower number of threads, code with prefetching has a much higher bus utilization than the code without in this function. This indicates that prefetching is effective and making good use of the bus bandwidth at lower number of threads. However, with increasing number of threads, the difference becomes smaller and at eight threads, the bus utilization is very similar for the two codes. This means that at higher number of threads, with the increased portion of execution time spent in this function, both executables tend to saturate the bus and hence there is no additional benefit from having the prefetching.

Although, HOP and Rsearch appear to have dramatic increase in bus utilization in Fig. 4, their impact on performance is minimal since overall bus utilization is low. Hence, we do not discuss them in this section.

6.2. *Cache utilization*

For some benchmarks, compiler is able to implement appropriate prefetching and improve the performance. However, in multithreaded mode, these static optimizations are not able to adapt to the changing runtime conditions, rendering them ineffective. K-means, a clustering algorithm that aims at discovering the underlying data distribution in a collection of objects, is one such application. It shows receptivity to compiler-directed prefetching in single-thread mode, running 12% faster than the code without prefetching. This receptivity changes with increasing number of threads and at eight threads, code with prefetching become 7% slower than the one without.

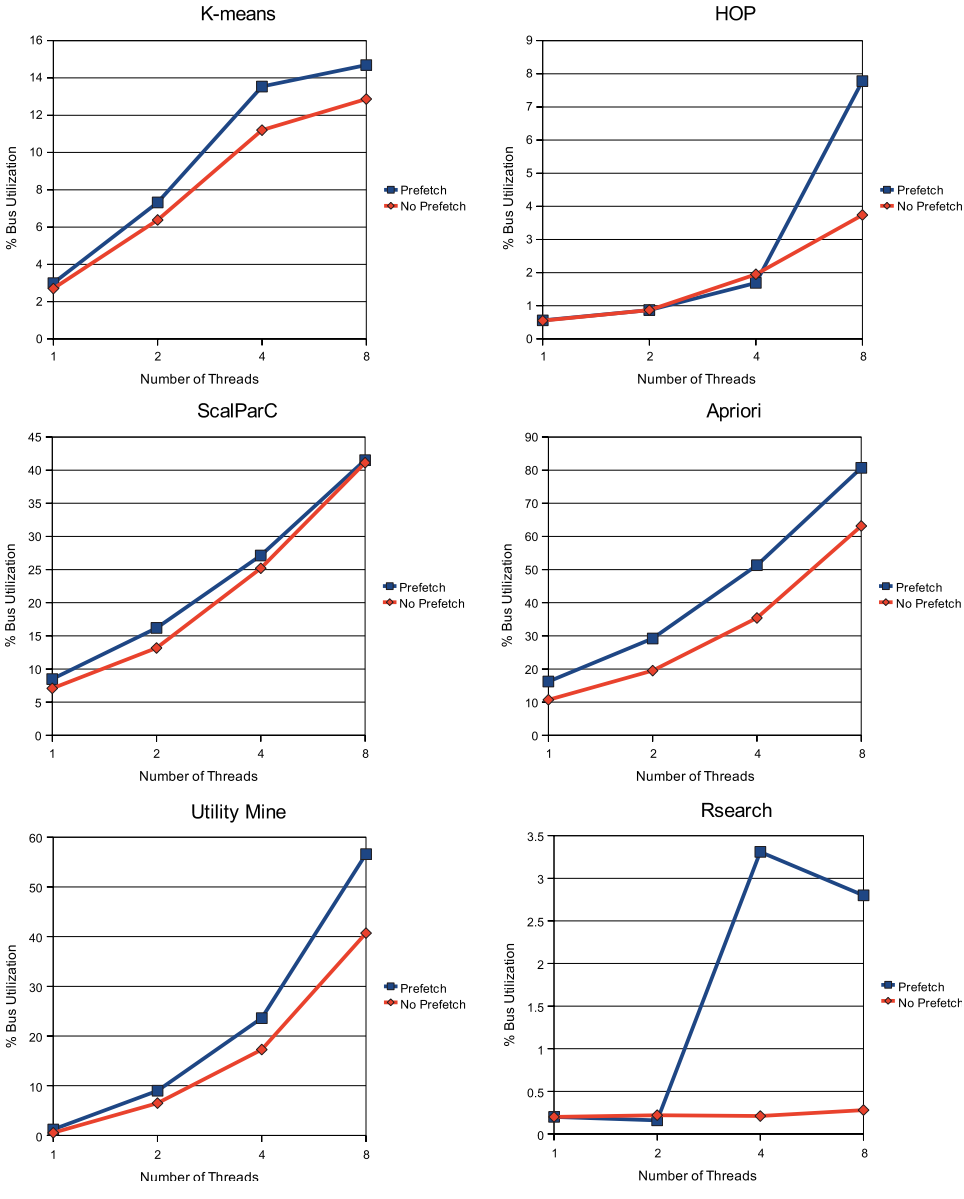


Fig. 4. Effect of prefetching on bandwidth utilization for NU-MineBench applications.

In its critical section, data access is *strided* for K-MEANS and compiler-directed prefetch instructions are useful in single-threaded mode. In multithreaded mode, the entire data is divided among different threads and with increasing number of threads, the data that each thread handles become small enough to fit inside the cache, thus making the prefetching redundant. The additional memory bus utilization that these

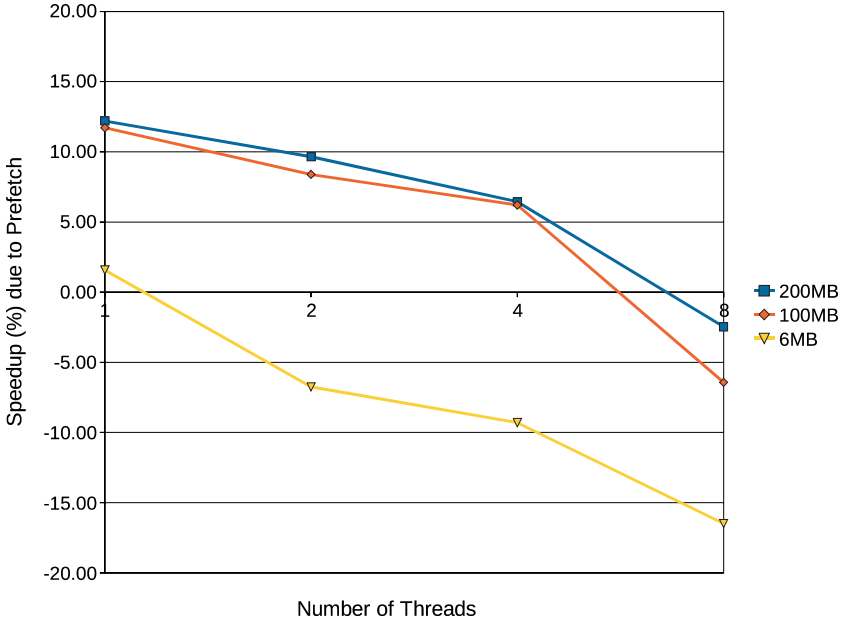


Fig. 5. *K*-means speedup (%) for different datasets.

instructions create, along with CPU-CYCLES required for address calculation, make the code with prefetching less efficient with increasing number of threads.

To study the effect of dataset sizes on the effectiveness of compiler-directed prefetching, we used three datasets of different sizes to run K-MEANS and their performance is as shown in Fig. 5. The figure shows the speedup achieved by compiler-directed prefetching in percentage for *K*-MEANS, for different dataset sizes, for thread numbers one, two, four, and eight. The smaller dataset of size 6 MB was chosen so as to fit into the last-level cache at relatively lower number of threads whereas, the datasets of size 100 MB and 200 MB were chosen so as not to fit into the last-level cache at lower number of threads. As seen in the figure, the smaller dataset fits into the cache at two threads and renders the compiler-directed prefetching ineffective. The 100 MB and 200 MB datasets seem to fit into the last-level cache at eight threads only. At eight threads, the prefetch effectiveness is still much better for 200 MB than 100 MB which indicates that the dataset is not yet completely fitting into the last-level cache as compared to the 100 MB dataset. As we increase the number of threads even further, the largest dataset will also fit into the last-level cache.

6.3. *Effects of locking and serialization*

The only exceptional behavior we see in Fig. 3 is UTILITY-MINE. For this application, compiler-directed prefetching has a positive effect and this improvement increases

with number of threads. Utility-Mine, an association rule mining algorithm similar to Apriori, features strided data access at its major *hot-spot*. The critical function uses a lock that serializes the execution. Compiler-directed prefetching is inserted in this critical section and thus helps in speeding up the application. The effectiveness improves with increasing number of threads as the prefetching reduces the execution time of the critical section and increases parallel overlap. This leads to a performance improvement of 1%, 22%, 34%, and 30%, for thread numbers one, two, four, and eight, respectively, in the critical section. This improvement, however, seems to halt at eight threads as another section becomes the most time-consuming and the advantages of compiler-directed prefetching becomes less dominating.

In this section, we examined several scenarios where static compiler optimizations are not able to provide optimal solutions: (i) in Apriori and ScalParC where additional pressure on shared resources is introduced with increasing number of threads; (ii) in K-means where runtime characteristics like dataset size changes with number of threads. In these scenarios, static compiler optimizations are not able to adapt to the runtime conditions and these cases warrant dynamic optimization techniques[26–29] for achieving optimal performance. Previous works[17–19,30,31] have studied the sensitivity of data mining benchmarks to last-level cache architecture. In multithreaded execution, the effects of data sharing and resource utilization could be sensitive to last-level cache architecture. Our study is limited to private last-level cache, although, we believe these observations hold true for shared last-level cache architecture.

7. Effect of Prefetching on Scalability

Many data mining applications exhibit thread-level parallelism, and previous works have demonstrated that these applications can scale linearly on parallel machines.[32] However, we have demonstrated that only a few benchmarks scale linearly when multiple threads of execution shares the same cores.[19] Our work shows the impact of data sharing and organization of last-level cache architecture can affect scalability. Previous work[16] has also pointed out the importance of communication overheads and resource utilization in the workload scalability.

Aggressive optimizations can change the memory access patterns of parallel threads, and affect scalability. In this section, we examine the effect of compiler-directed prefetching on the scalability of data mining benchmarks on multicore architecture. Figure 6 shows the relative speedup of data mining applications for thread numbers one, two, four, and eight, for codes generated with and without compiler-directed prefetching.

For most applications, such as HOP, Rsearch, ScalParC, and Apriori, we observe that code without prefetch scales better than with prefetching. At higher thread count, these applications suffer from increased pressure on the memory hierarchy due to prefetch instructions which are not always beneficial as discussed in Sec. 6.

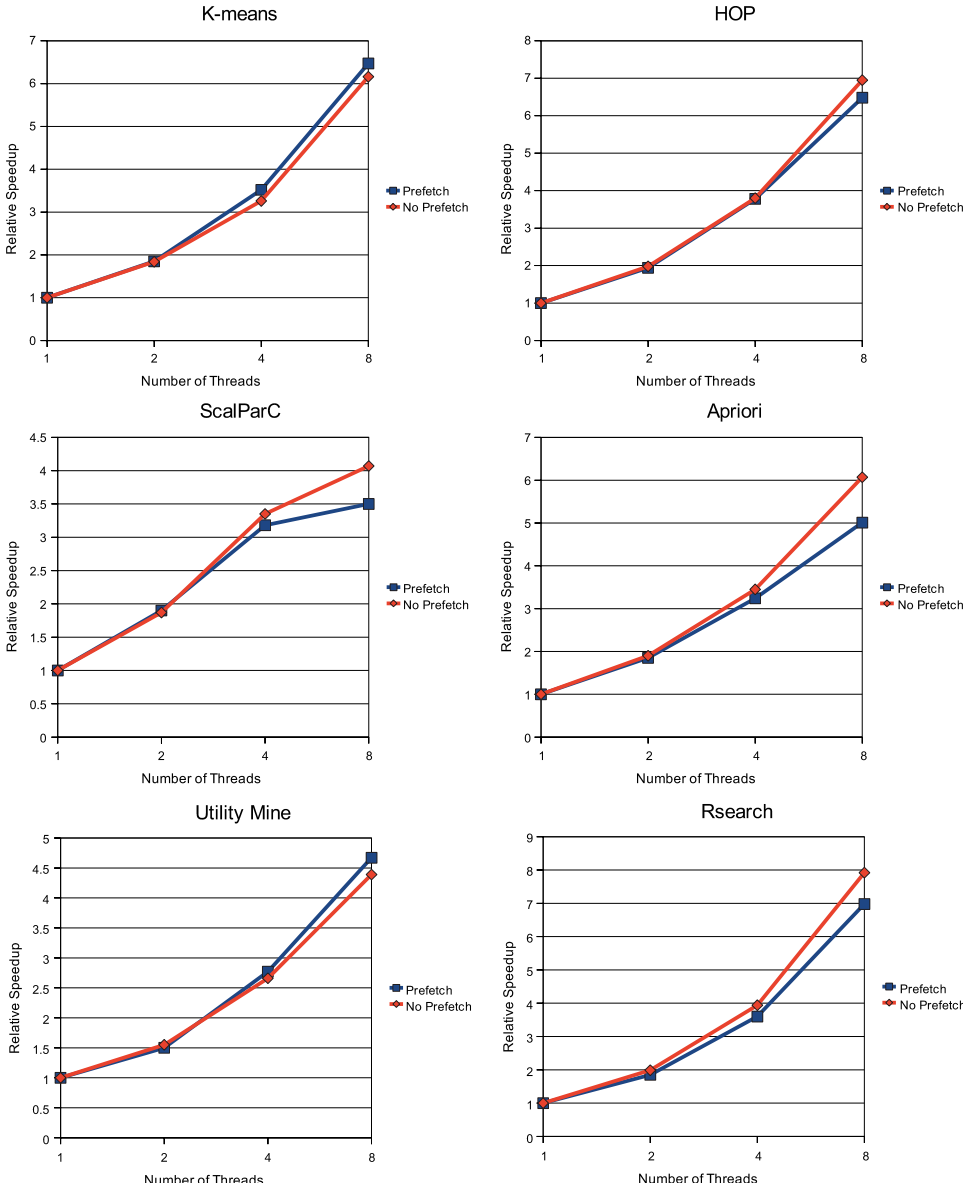


Fig. 6. Effect of prefetch on scalability.

We observe exceptions to this general behavior in Utility Mine and K-means. As we discussed in Sec. 6, Utility Mine benefits due to prefetch instructions, with increasing thread count, due to an inherent serialization in its *hot-spot*. This translates to better scalability for prefetch-enabled code. In the case of K-means,

the dataset fits into last-level cache with increasing thread count and prefetch instructions are beneficial, although their benefit decreases with increasing thread count. Our infrastructure consisted of microprocessor with private last-level cache. The behavior observed could become even more complex if we use microprocessor with shared last-level cache, due to increased interaction between threads on common data in the cache.

To summarize, on multicore processors, aggressive compiler optimizations can affect the scalability in either ways: when it exacerbates resource contention, it is less scalable. This was demonstrated by increased bus bandwidth utilization in APRIORI and SCALPARC; when it mitigates resource contention, it is more scalable. This was observed in the cases of K-MEANS and UTILITY MINE where prefetch instructions aid access to the memory hierarchy.

These behaviors are hard to determine statically for the compiler and can change with number of threads used in the multithreaded execution. Recent works[28,33,34] have explored hardware- and software-based optimization techniques for managing threads in multithreaded execution on out-of-order multicore architectures. Similar optimization techniques might be even more effective, on in-order multicore processors, for emerging workloads like data mining applications.

8. Related Work

There have been a number of studies in characterizing data mining applications.[16–18,35,36] and various works have analyzed the performance of specific categories of data mining workloads.[37–39] Most of the previous studies have been on processors with out-of-order issue logic. Considering the growing importance of in-order issue processors in multicore architectures, our study is based on an advanced in-order processor. Mekkat *et al.*[19] provided a comprehensive study of data mining benchmarks from five different categories. In addition to memory hierarchy and scalability characteristics, they also discuss the instruction-level parallelism and dynamic (runtime) behaviors of these applications. In this paper, we extend this study to present a detailed analysis of effectiveness of compiler optimization techniques on data mining applications in the context of in-order processor architectures. These compiler optimizations gain importance as they play a significant role in extracting performance from the relatively simpler hardware of in-order processors. In particular, we look at the effectiveness of compiler-directed prefetching on serial and parallel executions of data mining benchmarks.

There have been numerous efforts on adapting data mining algorithms to parallel platforms. These efforts have included parallelized algorithms for clustering, classification, and association rule mining; mostly for shared and distributed memory architectures. Examples include: Refs. 32, 40–44. In this paper, we use the OpenMP parallelized versions of data mining applications provided by NU-MineBench to study the effectiveness of compiler-directed prefetching on data mining applications

in shared memory multicore processor systems, which is increasingly becoming the *de-facto* standard for modern multiprocessor systems.

Data prefetching is an extensively investigated topic, and Vanderwiel and Lilja[45] survey existing work in this area. Data prefetching can be implemented in both hardware and software. Previous works have shown that hardware prefetching is effective for strided memory accesses.[4−7] Hardware prefetching techniques for more complex patterns are proposed by Refs. 46−49. Previous works on compiler-directed prefetching for regular memory accesses have been done by Mowry *et al.*[9,10] Luk and Mowry[8] apply compiler-directed prefetching to linked data structures.

The unpredictable responses of data mining applications to compiler-directed prefetching shows that the compiler cannot make the best decisions statically. Software-based dynamic optimization techniques have been proposed by previous works[26−29] to supplement the effectiveness of compiler-based static optimization techniques. Data mining applications can potentially benefit from adaptive techniques that improve last-level cache performance. This is an important problem and has been studied extensively and previous works[33,34,50] discuss dynamic hardware solutions to improve the performance of last-level cache on multicore systems.

9. Conclusion

In this paper, we evaluate the effectiveness of compiler-directed prefetching, in the context of in-order multicore architectures, on reducing memory access latencies for several classes of data mining applications. Our study reveals that although properly inserted prefetching instructions can often effectively reduce memory access latencies for these applications, compilers are not always able to exploit this potential. In fact, compiler-directed prefetching is effective on some applications, but can degrade performance dramatically for others. Thus, existing compiler technologies for inserting prefetch instructions cannot be directly deployed on data mining applications.

Our investigation on single-threaded data mining applications shows that the causes for ineffective prefetching is multi-facet: while cache pollution and resource contention are the most common causes, some impacts are less obvious and architecture dependent. For example, prefetching instructions can cause pipeline stalls by causing exceptions, such as page faults, and saturating the L2 cache load buffer. For multithreaded execution on a single chip, the impact of resource contention becomes more prominent. In almost all applications, prefetching becomes progressively detrimental as the number of threads increase. As a result, applications are more scalable without compiler-directed prefetching. However, we also observe an exceptional case where compiler-directed prefetching is able improve scalability by effectively optimizing codes inside a critical section.

In the context of data mining applications, existing compiler-directed prefetching can become ineffective if it is unable to accurately estimate the runtime

behaviors in the presence of (i) complex control flow and memory access patterns; (ii) architectural dependent behaviors; and (iii) bottlenecks created by resource contentions. Thus, dynamic optimization techniques that can monitor the runtime behaviors of these application and tune prefetching accordingly can potentially exploit the full power of compiler-directed prefetching.

Acknowledgments

This work is supported in part by grants from National Science Foundation under CNS-0834599, CSR-0834599, and CPS-0931931, a contract from Semiconductor Research Corporation under SRC-2008-TJ-1819, and gift grants from HP, IBM and Intel.

References

1. I. Kadayif, M. Kandemir and U. Sezer, An integer linear programming based approach for parallelizing applications in on-chip multiprocessors, *Proc. 39th Annual Design Automation Conf., DAC '02*, ACM, New York, NY, USA (2002), pp. 703−706.
2. J. Li and J. F. Martinez, Dynamic power-performance adaptation of parallel computation on chip multiprocessors, *High-Performance Computer Architecture, 2006. The Twelfth Int. Symp.* (2006), pp. 77−87.
3. L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan and P. Hanrahan, Larrabee: A many-core x86 architecture for visual computing, *ACM SIGGRAPH 2008 papers, SIGGRAPH '08*, ACM, New York, NY, USA (2008), pp. 18:1−18:15.
4. J.-L. Baer and T.-F. Chen, An effective on-chip preloading scheme to reduce data access penalty, *Supercomputing '91: Proc. 1991 ACM/IEEE Conf. Supercomputing*, ACM, New York, NY, USA (1991), pp. 176−186.
5. F. Dahlgren and P. Stenstrom, Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors, *HPCA '95: Proc. 1st IEEE Symp. High-Performance Computer Architecture*, IEEE Computer Society, Washington, DC, USA (1995), p. 68.
6. N. P. Jouppi, Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, *SIGARCH Comput. Architecture News* **18** (1990) 364−373.
7. I. Sklenář, Prefetch unit for vector operations on scalar computers, *SIGARCH Comput. Architecture News* **20** (1992) 31−37.
8. C.-K. Luk and T. C. Mowry, Compiler-based prefetching for recursive data structures, *Proc. Seventh Int. Conf. Architectural Support for Programming Languages and Operating Systems* (1996), pp. 222−233.
9. T. C. Mowry, M. S. Lam and A. Gupta, Design and evaluation of a compiler algorithm for prefetching, *SIGPLAN Not.* **27** (1992) 62−73.
10. T. C. Mowry and A. Gupta, Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *J. Parallel Distr. Comput.* **12** (1991) 87−106.
11. J.-F. Collard and D. Lavery, Optimizations to prevent cache penalties for the Intel® Itanium® 2 processor, *Code Generation and Optimization, 2003. CGO 2003. Int. Symp.* (2003), pp. 105−114.

12. J. Han, R. B. Altman, V. Kumar, H. Mannila and D. Pregibon, Emerging scientific applications in data mining, *Commun. ACM* **45** (2002) 54–58.
13. P. Tan, M. Steinbach and V. Kumar, *Introduction to Data Mining* (Addison-Wesley, 2005).
14. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams and K. A. Yelick, The landscape of parallel computing research: A view from berkeley, University of California (2006).
15. P. Dubey, A platform 2015 workload model: Recognition, mining and synthesis moves computers to the era of tera, *Intel Technol. J.* (2005).
16. B. Ozisikyilmaz, R. Narayanan, J. Zambreno, G. Memik and A. Choudhary, An architectural characterization study of data mining and bioinformatics workloads, *The IEEE Int. Symp. Workload Characterization* (*IISWC*) (2006).
17. K. Shaw, Understanding the working sets of data mining applications, *The Eleventh Workshop on Computer Architecture Evaluation using Commercial Workloads* (*CAECW-11*) (2008).
18. I. Jibaja and K. Shaw, Understanding the applicability of CMP performance optimizations on data mining applications, *The IEEE Int. Symp. Workload Characterization* (*IISWC 2009*) (2009).
19. V. Mekkat, R. Natarajan, W.-C. Hsu and A. Zhai, Performance characterization of data mining benchmarks, *INTERACT-14: Proc. 2010 Workshop on Interaction Between Compilers and Computer Architecture*, ACM, New York, NY, USA (2010), pp. 1–8.
20. R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik and A. Choudhary, Minebench: A benchmark suite for data mining workloads, *the IEEE Int. Symp. Workload Characterization* (*IISWC*) (2006).
21. Intel Corporation Intel Itanium-2 processor, http://www.intel.com/products/processor/itanium/index.htm.
22. S. Eranian, Perfmon: Linux performance monitoring for IA64, http://perfmon2.sourceforge.net/.
23. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood, Pin: Building customized program analysis tools with dynamic instrumentation, *Proc. Programming Language Design and Implementation* (*PLDI*) (2005).
24. SPEC.org. The SPEC CPU 2006 Benchmark Suite, http://www.specbench.org.
25. R. Fu, A. Zhai, P.-C. Yew, W.-C. Hsu and J. Lu, Reducing queuing stalls caused by data prefetching, *INTERACT-11: Proc. 2007 Workshop on Interaction Between Compilers and Computer Architecture* (2007).
26. J. Lu, H. Chen, P. C. Yew and W. C. Hsu, Design and implementation of a lightweight dynamic optimization system, *J. Instruction-Level Parallelism* **6** (2004).
27. J. Lu, A. Das, W. Hsu, K. Nyugen and S. Abraham, Dynamic helper threaded prefetching on the Sun UltraSPARC® CMP processor, *Proc. 38th IEEE/ACM Intl. Symp. Microarchitecture* (*Micro*) (2005).
28. Y. Luo, V. Packirisamy, W.-C. Hsu, A. Zhai, N. Mungre and A. Tarkas, Dynamic performance tuning for speculative threads, *Proc. 36th Intl. Symp. Computer Architecture* (*ISCA*) (2009).
29. Y. Luo, V. Packirisamy, W.-C. Hsu and A. Zhai, Energy-efficient speculative threads: Dynamic thread allocation in same-is a heterogeneous multicore system, *Proc. 2010 Int. Conf. Parallel Architectures and Compilation Techniques* (*PACT*) (2010).

30. A. Jaleel, M. Mattina and B. Jacob, Last level cache (LLC) performance of data mining workloads on a CMP — A case study of parallel bioinformatics workloads, *The Twelfth Int. Symp. High-Performance Computer Architecture* (2006), pp. 88−98.
31. W. Li, E. Li, A. Jaleel, J. Shan, Y. Chen, Q. Wang, R. Iyer, R. Illikkal, Y. Zhang, D. Liu, M. Liao, W. Wei and J. Du, Understanding the memory performance of data-mining workloads on small, medium, and large-scale cmps using hardware-software co-simulation, *ISPASS 2007*: *IEEE Int. Symp. Performance Analysis of Systems & Software* (2007), pp. 35−43.
32. M. Joshi, G. Karypis and V. Kumar, Scalparc: A new scalable and efficient parallel classification algorithm for mining large datasets, *Int. Parallel Processing Symp.* (1998).
33. M. A. Suleman, M. K. Qureshi and Y. N. Patt, Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs, *SIGPLAN Not.* **43** (2008) 277−286.
34. E. Ebrahimi, O. Mutlu, C. J. Lee and Y. N. Patt, Coordinated control of multiple prefetchers in multi-core systems, *MICRO 42*: *Proc. 42nd Annual IEEE/ACM Int. Symp. Microarchitecture*, ACM, New York, NY, USA (2009), pp. 316−326.
35. Y. Liu, J. Pisharath, W. Liao, G. Memik, A. Choudhary and P. Dubey, Performance evaluation and characterization of scalable data mining algorithms, *Proc. IASTED* (2004).
36. W. Li, E. Li, A. Jaleel, J. Shan, Y. Chen, Q. Wang, R. Iyer, R. Illikkal, Y. Zhang, D. Liu, M. Liao, W. Wei and J. Du, Understanding the memory performance of data-mining workloads on small, medium, and large-scale cmps using hardware-software co-simulation, *the IEEE Int. Symp. Performance Analysis of Systems and Software* (*ISPASS*) (2007).
37. J. P. Bradford and J. Fortes, Performance and memory-access characterization of data mining applications, *Annual IEEE Int. Workshop on Workload Characterization* (1998).
38. A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen and P. Dubey, A characterization of data mining algorithms on a modern processor, *DaMoN '05: Proc. 1st Int. Workshop on Data Management on New Hardware*, ACM, New York, NY, USA (2005).
39. Y. Chen, Q. Diao, C. Dulong, W. Hu, C. Lai, E. Li, W. Li, T. Wang and Y. Zhang, Performance scalability of data-mining workloads in bioinformatics, Technical Report, Intel Technology Journal (2005).
40. M. J. Zaki, C.-T. Ho and R. Agrawal, Parallel classification for data mining on shared-memory multiprocessors, *Proc. 15th Int. Conf. Data Engineering* (1999), pp. 198−205.
41. S. Parthasarathy, M. J. Zaki, M. Ogihara and W. Li, Parallel data mining for association rules on shared-memory systems, *Knowl. Inf. Syst.* **3** (2001) 1−29.
42. E.-H. Han, G. Karypis and V. Kumar, Scalable parallel data mining for association rules, *SIGMOD Rec.* **26** (1997) 277−288.
43. D. Foti, D. Lipari, C. Pizzuti and D. Talia, Scalable parallel clustering for data mining on multicomputers, *Int. Parallel and Distributed Processing Symp. 2000* (*IPDPS'00*) (2000), pp. 390−398.
44. K. Stoffel and A. Belkoniene, Parallel k/h -means clustering for large data sets, *Euro-Par'99*: *Proc. 5th Int. Euro-Par Conf. Parallel Processing* (1999), pp. 1451−1454.
45. S. P. Vanderwiel and D. J. Lilja, Data prefetch mechanisms, *ACM Comput. Surv.* **32** (2000) 174−199.
46. T.-F. Chen and J.-L. Baer, Effective hardware-based data prefetching for high-performance processors, *IEEE Trans. Comput.* **44** (1995) 609−623.

47. A. Roth, A. Moshovos and G. S. Sohi, Dependence based prefetching for linked data structures, *SIGOPS Oper. Syst. Rev.* **32** (1998) 115−126.
48. T. Alexander and G. Kedem, Distributed prefetch-buffer/cache design for high performance memory systems, *Int. Symp. High-Performance Computer Architecture* (1996), pp. 254−263.
49. D. Joseph and D. Grunwald, Prefetching using markov predictors, *ISCA '97: Proc. 24th Annual Int. Symp. Computer Architecture*, ACM, New York, NY, USA (1997), pp. 252−263.
50. M. A. Suleman, O. Mutlu, M. K. Qureshi and Y. N. Patt, Accelerating critical section execution with asymmetric multi-core architectures, *SIGPLAN Not.* **44** (2009) 253−264.