Understanding the Working Sets of Data Mining Applications

Kelly A. Shaw
University of Richmond
Richmond, Virginia
kshaw@richmond.edu

Abstract

Data mining applications discover useful information or patterns in large sets of data. Because they can be highly parallelizable and computationally intensive, data mining applications have the potential to take advantage of the large numbers of processors predicted for future multi-core systems. However, the potential performance of these applications on this emerging platform is likely to be impeded by their intensive memory usage. In addition to accessing memory frequently, some of these applications exhibit exceedingly large working set sizes. Storing these large working sets on chip in their entirety may be prohibitively expensive or infeasible as these working set sizes continue to grow with problem size. Greater insight into the characteristics of these working sets is needed in order to determine alternative approaches to storing the entire working set on-chip.

In this paper, we examine the memory system characteristics of a set of applications from the MineBench data mining suite. We analyze these applications in an architecture independent manner in order to gain greater understanding into the composition of the data working set; in particular, we document the duration and frequency of active and idle periods for working set data. We find that working set data may be reused repeatedly throughout a program's execution, but each use is for a short period of time. The resulting long idle periods may enable alternate techniques to be used instead of caching in order to obtain good memory performance. We show that for several of these applications, simple prefetching schemes may alleviate the need to cache the large working sets.

1. Introduction

As the quantity of data collected each year has grown, the importance of data mining applications, which extract patterns from this data, has grown correspondingly. Data mining technology can be found in a wide variety of fields including medicine, marketing, finance, entertainment, and security. Data mining applications interest computer architects not only because of their growing relevance, but because they exhibit characteristics that distinguish them from existing workloads including integer (SPEC INT), floating point (SPEC FP), multimedia (MediaBench), and decision support (TPC-H) applications [8]. In general, data mining applications are unique because they combine high data demands with high computational demands.

Given that data mining applications exhibit these unique characteristics, the demands they will place on the resources of future chips, specifically multi-core chips, cannot easily be predicted. Consequently, recent work analyzes the different demands created by these applications in greater depth. Two of these studies, [3] and [5], conclude that these applications require large lower level caches to compensate for large, data working set sizes (32MB or more). However, this brute force approach to memory organization for data mining applications is not scalable given that working set sizes can be expected to continue to grow with problem size. Additionally, storing large quantities of data on-chip may not be feasible for systems where power is an important concern.

In this paper, we examine how data in the working set is used over program execution for a set of applications from the MineBench [7] data mining benchmark suite. MineBench consists of complete applications exhibiting a variety of algorithms used to learn from data sets ranging from grocery purchases to DNA sequences, with each algorithm potentially exhibiting different resource demands. Our goal is to better understand how data is used in order to suggest alternative memory organizations and/or data management policies to extremely large capacity, lower level caches. We use an architecture independent approach to show that while data may be reused throughout the majority of a program's execution, most data will be used for a small period of time (hundreds to thousands of instructions) before remaining unused for extended periods (millions or tens of millions of instructions). Thus, data usage is not exactly streaming in that data will be reused over a pro-

gram's lifetime, but the brief periods of active use suggest that there is little benefit to storing data beyond short periods of time if the entire working set cannot be retained. We show that some, but not all, of these applications are amenable to data prefetching as a mechanism for reducing latency and reducing storage capacity, assuming sufficient off-chip bandwidth. These results suggest that existing techniques may potentially be deployed to obtain good memory performance for data mining applications on chips with smaller on-chip storage capacities.

This paper proceeds by first discussing recent work on characterizing the resource demands of data mining applications. Section 3 then describes the methodology used for our analysis. We analyze how working set data is used in Section 4, analyze the potential uses of prefetching in Section 5, and then conclude.

2. Related work

The growing importance of data mining workloads has instigated examination of the hardware demands exhibited by these applications, particularly on shared memory multiprocessor systems. As part of this process, Narayanan et al. have provided a data mining application suite called MineBench [7]. The fifteen applications in this suite include algorithms for clustering, classification, association rule mining, optimization, and structured learning.

Several studies characterizing the resource demands of data mining applications established that these applications are both compute and memory intensive. Zambreno et al. found that most of the applications in MineBench execute a significantly high number of floating point operations [9]. Ghoting et al. similarly showed that the applications they studied were compute intensive, either issuing high numbers of floating point or integer operations [2]. These studies also found that these applications experience high level two data cache miss rates, primarily due to the use of large data sets. Ozisikyilmaz et al. show that this combination of high compute and memory intensity results in a clustering algorithm distinguishing data mining applications from existing workloads, including SPECInt, SPECFP, MediaBench, and TPC-H [8]. Additionally, they show that there is high variability in the characteristics displayed by the MineBench applications.

Although Choudhary et al. explore creating special hardware to handle the compute intensity of these applications [1], most subsequent studies have explored the implications of data mining applications' memory demands. The use of large data sets which cause high level two data cache miss rates resulted in Ghoting et al. concluding that data mining applications exhibit poor temporal locality [2]. Jaleel et al. discovered that temporal locality across threads could potentially be exploited to increase cache hit rates by increasing the total storage capacity of a system through the use of shared versus private caches [3]. Similarly, Li et al. used cache sensitivity analysis to show that cache hit rates can be improved by creating very large, lower level caches that can contain the very large working set sizes (up to 256MB) for these applications [5]. Finally, Ghoting and Li conclude that good spatial locality enables performance improvement via hardware prefetching for some data mining applications.

The work in this paper builds on prior work characterizing memory demands. Using architecture independent analysis, we reconcile the discrepancy between [2] and [3] and [5] regarding temporal locality. Specifically, we show that although temporal locality exists in these applications' data working sets, most data experience very short periods of use separated by very long idle periods; consequently, these applications can appear to have streaming access patterns, meaning little temporal locality, if the memory system cannot capture enough of the working set. We also consider whether prefetching can potentially be used instead of high capacity, on-chip storage to maintain good memory performance; our architecture independent analysis removes the specific system bandwidth availability issues experienced in earlier work from impacting this analysis.

3. Methodology

3.1. Simulator

Pin [6] is used to dynamically instrument the binaries used in this study. In order to characterize memory access behavior, we have Pin pass instruction addresses, memory access types, effective addresses, and data sizes to a Pin tool we created; this is done for user-level instructions executed by the application. This Pin tool accumulates access statistics for 64B memory lines in order to allow an architecture independent analysis of how data is used.

3.2. Workload

We have chosen to study eight of the applications in the MineBench suite compiled with gcc4.2. Table 1 specifies the applications and the parameters used. This subset includes applications that perform clustering, classification, and association rule mining. [1]

Although most of the MineBench applications can be run with multiple threads via OpenMP pragmas, we have limited our study to examining memory behaviors when only a single thread is executed. While this decision prevents observation of data sharing across threads, it makes it possible to understand the demands created by individual threads.

[1] This subset was chosen for run time considerations; the non-studied applications have run times an order of magnitude longer than these eight.

Application	Parameters
Apriori	-i data.ntrans_1000.tlen_10.nitems_1.npats_2000.patlen_6 f offset_file_1000_10_1_P1.txt -s 0.0075 -n 1
Bayesian	-d F26-A64-D250K_bayes.dom F26-A64-D250K_bayes.tab F26-A64-D250K_bayes.nbc
Eclat	-i ntrans_2000.tlen_20.nitems_1.npats_2000.patlen_6 -e 30 -s 0.0075
HOP	61440 particles_0_64 64 16 -1 1
K-means	-i edge -b -o -p 1
ScalParC	F26-A32-D250K.tab 250000 32 2 1
SVM-RFE	outData.txt 253 15154 30
Utility	real_data_aa_binary real_data_aa_binary_P1.txt product_price_binary 0.01 1

Table 1. Application execution parameters

Although individual threads in parallelized executions of these applications my work on smaller portions of the data set than a thread in a single-threaded version, we expect individual threads' data sets to grow as problem sizes grow.

4. Working set characterization

In order to gain a deeper understanding of how data is used throughout a program's lifetime, we examine memory usage patterns without considering a specific memory system. Table 4 confirms the data intensity of these applications shown in similar studies. For these applications, the average number of data references per instruction ranges from 0.29 to 0.73. While the division of these memory references between stack and non-stack locations varies, read accesses of non-stack data dominate write accesses (3-93 times as many reads as writes).

Application	Stack (R/W)	Non-stack (R/W)	Memory (MB)
Apriori	0.21 / 0.09	0.32 / 0.06	100.4
Bayesian	0.25 / 0.18	0.12 / 0.02	0.07
Eclat	0.33 / 0.06	0.30 / 0.04	22.6
HOP	0.17 / 0.04	0.13 / 0.02	3.4
K-means	0.12 / 0.04	0.25 / 0.11	1.5
ScalParC	0.25 / 0.15	0.12 / 0.04	113.3
SVM-RFE	0.04 / 0.02	0.23 / 0.002	44.9
Utility	0.15 / 0.10	0.22 / 0.004	503.0

Table 2. References/instruction and memory footprints

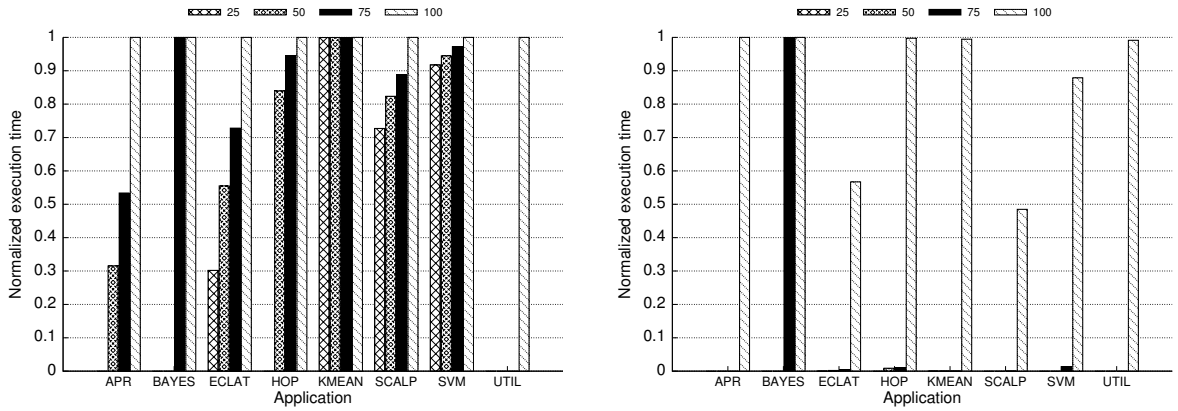
Table 4 also shows the memory footprints for these applications. The sizes vary greatly for these applications with the Bayesian application using less than 0.07MB of memory while the Utility application usage exceeds 500MB.

4.1. Working set sizes

We obtain an understanding of how the working set size compares to the total memory footprint by examining how long non-stack data are used during program execution. (The number of memory blocks used for stack data is minimal, so we do not analyze stack data use.) To determine the length of time data is used throughout a program, called its *lifetime*, we track the time between the data's first and last access. Figure 1(a) shows a chart of the percentiles of data lifetimes. For each application, we normalize the percentiles to the program duration. For ScalParC, the data lifetime for the 25th percentile is 72% of the program's execution, the 50th is 82%, the 75th percentile is 89%, and the 100th percentile is 99%. This means that more than 75% of the non-stack data in ScalParC is used for more than 72% of the program's duration.

From Figure 1(a), we observe that K-means, ScalParC, and SVM-RFE use more than 75% of their non-stack data for more than 70% of each program's duration. The working set sizes for these applications would therefore be nearly as large as the total memory footprint for these applications. In contrast, Apriori, Bayesian, Eclat, and HOP exhibit a great deal of selectivity in how long data continues to be used throughout program execution. For example, more than 25% (but less than 50%) of the non-stack data in the Bayesian application have lifetimes that span most of the program's execution while a separate 25% or more of the data have lifetimes that persist for less than 1% of the program's execution. Consequently, these applications will have working set sizes significantly smaller than their total memory footprints and working set sizes which will potentially change over time. Finally, most of the data in the Utility application have exceptionally short lifetimes with a small percentage persisting for the program's entirety; this application exhibits the memory characteristics of a streaming memory application where data is used briefly and then discarded.

This architecture independent analysis confirms that temporal locality does exist for this subset of MineBench applications. It also shows that several of these applications have very large working set sizes. If we combine information about total memory footprint sizes and working set sizes, we observe that some applications like ScalParC and SVM-RFE have large working sets. Despite their selectiv-



(a) Lifetimes

(b) Active usage

Figure 1. Data lifetimes(a) and total active use(b)

ity, Apriori and Eclat will have moderately large working set sizes. In contrast, Utility will have a small working set size despite its large memory footprint. Thus, some of these applications, but not all, can benefit from large capacity, on-chip storage.

4.2. Active Data Usage

In the previous section, we showed that each of these applications has a set of non-stack data that is used throughout the entire program execution. Depending on the application, the size of this set varies from a small portion of the non-stack data to practically all of the non-stack data accessed by the program. This data contributes directly to the working set size calculated when cache sizes are continually increased until no benefits are accrued from increased cache size as in [5]. However, this data may not be actively used throughout the program's lifetime; it may remain idle for long periods, using valuable storage resources. In this section, we examine the duration and frequency of periods of active use of working set data.

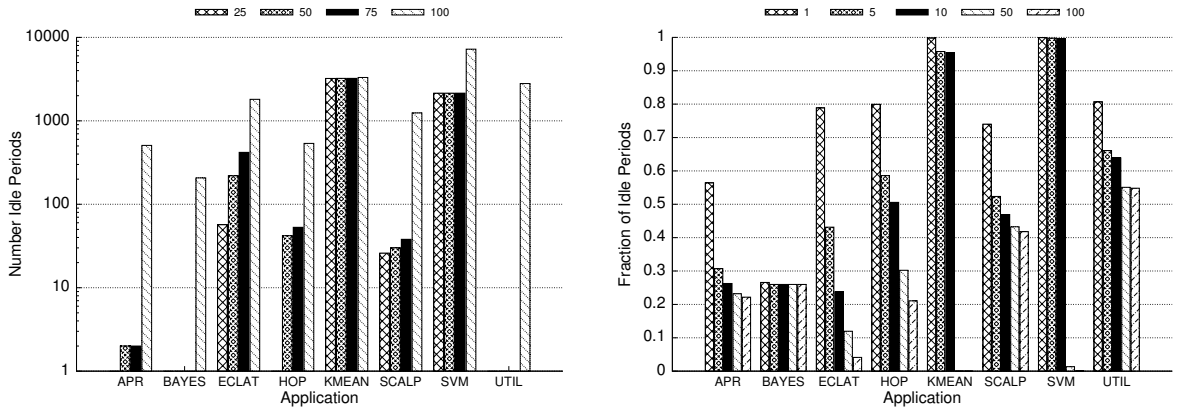
In order to quantify how data is used throughout a lengthy lifetime, we calculate the amount of time data is actively used before it remains idle for periods of 1 million cycles or longer. [2] For example, if data is accessed at times 1000; 30,000; 2,000,000; and 2,001,000, we would calculate its non-idle time as being (30,000 - 1000) + (2,001,000 -2,000,000) = 30,000 cycles. In contrast, if data is repeatedly used every 10,000 cycles starting at time 2,000 until time 3,000,000, it's non-idle time would be (3,000,000 - 2,000) = 2,998,000 cycles. This metric, therefore, allows us to distinguish data that is used continuously throughout its lifetime from data that is used repeatedly but for brief periods of time.

Figure 1(b) shows the percentiles of non-stack data's non-idle time as a fraction of total program execution time. For example, less than 25% of the data in HOP is used for the entire HOP program's lifetime; 50% of the data is used for between 0.008% and 0.01% of the program lifetime and the remaining data is used for less than 0.008% of the program lifetime. For all of the applications except Bayesian, we observe that less than 25% of the data are used continuously throughout the program lifetime. Most data remain idle for the majority of their lifetimes.

This non-idle time is actually spread over a number of reuses for most of the data with long lifetimes in these programs. Figure 2(a) shows the number of idle periods (periods of longer than 1 million cycles of data not being accessed) experienced by data. Again, the figure shows the percentiles for the number of idle periods experienced by non-stack data. More than 75% of the data in K-means experience approximately the same number of more than 3,200 idle periods. In contrast, less than 25% of the data in Apriori experience more than 2 idle periods while more than 50% of Apriori's data experience 2 idle periods. For some applications like Eclat, K-means, ScalParC, and SVM-RFE, large portions of the working set data will be repeatedly used. Depending on the length of time between these reuses of the data, it may or may not be worthwhile in a system to store this data on-chip to exploit this temporal locality.

We show the length of time of these idle periods in Figure 2(b). The chart shows the total percentage of idle periods lasting more than 1, 5, 10, 50, and 100 million cycles. (Note that all idle periods last at least 1 million cycles.) For applications like K-means and SVM-RFE, we see that long idle periods of more than 10 million cycles exist between the reuse of data. Given the long lifetimes of most data in these applications, most data in these applications remain

[2]This interval length was chosen based on results in [4] showing that using a 1024K cycle cache decay interval did not severely impact level two cache miss rates.



(a) Number idle periods

(b) Duration of idle periods

Figure 2. Frequency(a) and duration(b) of idle periods

idle for long periods. Since most data in ScalParC have long lifetimes, similar observations can be made although a fraction of idle periods last less than 5 million cycles. For applications like Apriori where some data is repeatedly reused over the program lifetime while other data is discarded after a small number of reuses, we observe a sharp drop in the number of idle periods lasting more than 1 million cycles but less than 5 million cycles and a stable fraction of idle periods lasting long periods of time. Thus, data that are reused in Apriori are reused before 5 million idle cycles and data that are not reused remain idle for extended periods of time. Similarly, more than 50% of idle periods in Utility persist for more than 100 million cycles because only a small set of data has long lifetimes; they are used and then remain idle for the remainder of the program.

4.3. Discussion

By examining the lifetimes and non-idle periods of non-stack data, we discover a complicated picture of how data are used in these data mining applications. While large portions of the total data accessed by some of these programs continue to be reused throughout the entire program, data lifetimes can be decomposed into short periods of use separated by potentially long idle periods. Given the number of idle periods experienced by data and the total non-idle times, we can calculate the average periods of non-idle periods lasting hundreds to thousands of cycles compared to idle periods of millions to tens of millions of cycles.

While the repeated use of data over the program lifetimes may encourage storing the entire working set in on-chip storage, the relatively short periods of active use require reconsideration of whether this approach is the best approach for data mining applications. There are negative consequences of storing idle data for long periods of time. As the working set sizes of data mining applications continue to grow, this solution will have trouble scaling. Additionally, as data centers become increasingly concerned about power consumption, having large on-chip storage capacity may be infeasible. Cache decay algorithms like [4] which decay lines after some preset number of idle cycles can have large negative impacts on the level two cache miss rates of these applications.

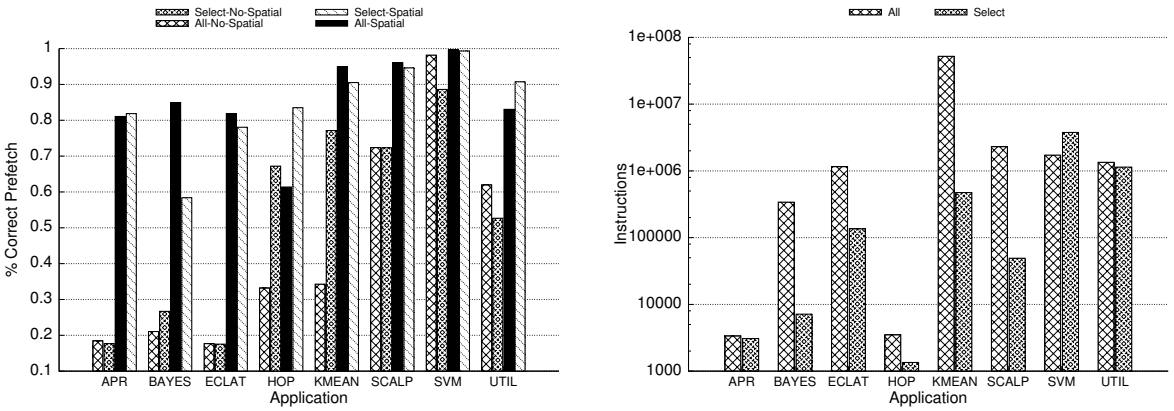
Given these concerns about the quantity of storage capacity needed by these applications' working sets and power consumption, it might be not be possible to retain the entire working set in a large on-chip cache. Techniques that can prefetch data to reduce latency will be necessary. In the next section, we analyze the predictability of memory accesses for these data mining applications.

5. Prefetching

In order to determine the likelihood that prefetching can be used successfully without modeling a specific memory hierarchy, we collect statistics for every instruction about the distances between subsequent non-stack memory references. If the distance between two memory references is repeated between the next two memory references, we consider the previous distance a correct predictor for what should be prefetched; the distance is the *stride*. If instructions frequently have the same stride, simple prefetching schemes may be useful in reducing cache miss rates.

For every instruction, we retain the frequency of its execution, the frequency of accesses to the same memory block as the previous memory access, the number of correct stride predictions (previous distance equals the next distance), and the total number of unique strides observed for that instruction. Contiguous accesses to the same data block represent a measure of spatial locality.

Figure 3(a) shows prediction accuracies for instructions



(a) Accuracy

(b) Time before prefetched block needed

Figure 3. Prefetch accuracy(a) and time before use after prefetch(b)

that access non-stack data. We calculate a single prediction accuracy number by weighting each instruction's prediction accuracy by its frequency. For each application, we show the prediction accuracy when consecutive accesses to the same data block are not included in correctness or frequency counts (No-Spatial) and when they are (Spatial). Because we are interested in determining how useful prefetching will be for instructions that access the large quantity of working set data, Figure 3(a) includes information about instructions that access large numbers of different data blocks (Select). Specifically, the 10% of instructions that access the largest number of unique non-stack data blocks are included. These instructions are expected to frequently access different data blocks and are therefore more likely to increase storage capacity needs. For comparison, we also show prediction accuracies when all instructions that access non-stack memory blocks are included (All).

Comparison of corresponding No-Spatial and Spatial bars in Figure 3(a) shows that all of the applications will benefit from spatial locality gained by an instruction successively accessing the same data block. However, not all applications have instructions with infrequently changing data access strides. ScalParC, SVM-RFE, K-means, and Utility exhibit strong prediction accuracies according to Select-Spatial, but Apriori, Bayesian, and Eclat do not. Consequently, a simple hardware prefetching scheme that uses a stride prediction table may be useful for some applications but not others.

Given that prediction accuracies for all instructions are similar to prediction accuracies for select instructions for all but two of the applications when spatial locality is included, simple prefetch hardware that prefetches on all memory requests should be sufficient. In the cases of HOP and Utility, prediction for all instructions is less accurate than the set of select instructions because slightly more than 10% of instructions access a large number of unique memory blocks. A selective prefetching scheme which prefetches memory only for specific instructions might therefore be beneficial.

Prefetching can only reduce latency if there is adequate time between initiation of a prefetch request and the requested data's use. For the same two sets of instructions analyzed above, we determine the average amount of time between accesses of different memory blocks by a given instruction. Figure 3(b) shows the weighted averages (based on instruction frequency) of these times for these applications. For all of these applications, these times are 1,000 instructions or greater, both when select and all instructions are examined; this implies that sufficient time may exist in these applications for prefetching to hide memory latency. Consequently, it may be possible to use prefetching for some of these applications as an alternative to storing large working sets on-chip.

6. Conclusions

The increasing importance of data mining applications requires future chips to be able to handle these computationally and memory intensive workloads. This paper examines the characteristics of the large data working sets documented in earlier studies. We show that data experience long idle periods, separated by brief periods of use. These long idle periods provide opportunities for handling data in ways that require smaller on-chip storage capacity than approaches that attempt to store entire working sets in large lower level on-chip caches. We also show that several of these applications are amenable to using simple stride prefetchers as one such alternative. These results suggest that structures like stream buffers for multimedia processing may be useful for improving the memory performance of these applications.

References

[1] A. N. Choudhary, R. Narayanan, B. Ozisikyilmaz, G. Memik, J. Zambreno, and J. Pisharath. Optimizing data mining workloads using hardware accelerators. In *Proc. of the Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, 2007.

[2] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y. Chen, and P. Dubey. A characterization of data mining workloads on a modern processor. In *DaMoN*, 2005.

[3] 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. In *Proc. of the 12th Intl. Symp. on High Performance Computer Architecture (HPCA)*, pages 250–260, 2006.

[4] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proc. of the 28th Annual Intl. Symp. on Computer Architecture (ISCA)*, pages 240–251, 2001.

[5] 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. In *IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 35–43, 2007.

[6] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[7] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. N. Choudhary. Minebench: A benchmark suite for data mining workloads. In *IISWC*, pages 182–188, 2006.

[8] B. Ozisikyilmaz, R. Narayanan, J. Zambreno, G. Memik, and A. N. Choudhary. An architectural characterization study of data mining and bioinformatics workloads. In *IISWC*, pages 61–70, 2006.

[9] J. Zambreno, B. Ozisikyilmaz, J. Pisharath, G. Memik, and A. Choudhary. Performance characterization of data mining applications using minebench. In *Proc. of the 9th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-09)*, 2006.