Understanding the Applicability of CMP Performance Optimizations on Data Mining Applications

Ivan Jibaja and Kelly A. Shaw
Department of Mathematics and Computer Science
University of Richmond
{*ij2gg, kshaw*}*@richmond.edu*

***Abstract*—A major challenge to the creation of chip multiprocessors is designing the on-chip memory and communication resources to efficiently support parallel workloads. A variety of cache organizations, data management techniques, and hardware optimizations that take advantage of specific data characteristics have been developed to improve application performance. The success of these approaches depends on applications exhibiting the presumed data characteristics.**

Data mining applications are a growing class of applications that discover patterns in large sets of collected data. Because these applications tend to be highly parallelizable, they represent an important workload for chip multiprocessors. However, the memory intensive nature of these applications means that they will stress these chips' memory and communication resources.

In this paper, we examine the data usage characteristics of a set of parallel data mining applications to determine the applicability of existing chip multiprocessor approaches to these applications. We show diversity of characteristics across and within these applications, making some techniques more applicable than others. We also discuss software approaches that could be used to either provide information to the hardware or assist the hardware in dynamically discovering data characteristics needed for the deployment of these techniques.

I. Introduction

Data mining applications extract useful information or patterns from large sets of raw data. They are currently used in a variety of fields including marketing, finance, scientific discovery, security, and entertainment. As the amount and type of collected data continue to grow, the importance and prevalence of data mining applications will grow. These applications tend to be both compute and memory intensive [1][2][3]. Their computation is largely parallelizable, making them a natural fit for the increasing number of processors being placed on single chips. Their tendency to have large working set sizes [4] and frequent sharing of data between parallel threads [5], however, has the potential to limit their performance on future chip multiprocessors.

One challenge for using chip multiprocessors is generating large quantities of parallel computations that can utilize the increasing number of processors. An equally important issue is designing a chip's supporting memory and communication infrastructure to achieve high performance. Research has generated a wide range of techniques that exploit specific data characteristics in order to address this issue. This body of work includes approaches that determine how to organize and manage on-chip caches [6][7][8][5] and how to reduce data access latencies through prefetching or reducing on-chip network overheads [9][10]. Other work has addressed how to exploit data characteristics to improve the performance of mechanisms which make it easier to program future chip multiprocessors [11].

This paper analyzes parallel applications from the MineBench data mining suite [12] to determine whether they exhibit characteristics exploited by a set of approaches developed for improving the performance of applications on chip multiprocessors. Our architecture independent examination of data usage enables us to observe the type, frequency, and predictability of data accesses and data sharing. Our approach also enables us to explore ways in which information about data characteristics can be conveyed by software to the hardware in order to help target the use of these optimizations.

The contributions of this work include an analysis of the data access characteristics of a set of parallel data mining applications. Our examination shows diversity in the data characteristics across applications as well as within applications. The benefits obtained by existing approaches, therefore, vary across and within applications. This suggests that future chips will need to dynamically choose which of a set of optimization techniques should be applied to a specific application based on the application's characteristics. We propose software techniques that can help identify or help hardware dynamically identify which data these techniques should be applied to.

The remainder of this paper is organized as follows. We describe a subset of the chip multiprocessor performance improving approaches in Section II and describe the existing characterization of data mining workloads in Section III. After describing our methodology in Section IV, we present our analysis in Section V. In Section VI, we discuss ways to help hardware identify data characteristics. Finally, we conclude in Section VII.

II. Existing Designs and Optimizations

A variety of techniques have been created for chip multiprocessors in order to improve the performance of the

memory and communication subsystems. Frequently, these optimizations are based upon some key attribute about how data is used. In this section, we briefly present a small subset of existing approaches in order to understand which features of data access patterns are being exploited. In Section V, we then characterize data mining applications with respect to these key characteristics.

A. Cache design and management

One challenge in designing chip multiprocessors is determining how to organize and manage the on-chip cache resources. Because physical proximity to data in an on-chip cache impacts access latency [13], designs and management strategies must balance the desire to minimize access latency and the need to store as much unique data as possible on the chip to avoid off-chip accesses. A set of studies have examined whether private or shared caches should be used for level two caches and lower on chip, with their results being largely dependent on the workload characteristics. Huh et al. explore how the fraction of shared versus private cache capacity impacts access latency and cache misses [6]. They find that sharing caches between a small number of processors provides the best performance for a set of commercial and scientific applications. In contrast, the high level of data sharing and large data set sizes in data mining bioinformatics workloads suggest that a single shared last level cache will perform better than private caches [5].

Other studies have explored how to manage data in shared caches via placement, migration, and replication of data. Again, the results for these techniques depend on workload characteristics. For example, Beckmann and Wood show that migration of data in shared caches results in heavily shared data being equally far from processors as it clusters in the center banks of the shared cache [7]. Thus, applications with large amounts of data sharing may encounter higher access latencies with this approach. Another technique, victim replication, provides replication of data in order to reduce access latencies by storing level one cache victims within local level two cache slices [14]. The replica is placed in a location that is either free, contains a block that has no sharers, or is an existing replica. Benefits of this approach depend on the availability of sufficient cache capacity to have space for replicas as well as access patterns that benefit from the retention of data evicted from the level one cache.

For commercial workloads, Beckmann et al. [15] find that a small number of shared read-only blocks account for a large fraction of level two cache requests. Consequently, they present a hardware approach that replicates shared read-only blocks in the level two cache. In contrast, Hardavellas et al. find that shared data is primarily read-write data in the server workloads they examine [8]. They show that blocks can be classified into three categories, namely instructions, private data, and shared data. In addition to being read-write data, shared data is generally shared across all processors. They present an approach where the OS classifies pages into these categories. They then place data on the chip in a way that minimizes access latency while maximizing storage capacity; private data is placed near the requesting core, instructions are replicated strategically, and shared read-write data is distributed evenly across the chip and not replicated in order to avoid coherence traffic.

B. Predictability of data accesses

1) Using prediction to reduce access latency: When applications' data accesses occur in predictable patterns, a variety of techniques for reducing data access latencies can be used. Beckmann and Wood establish that stride-based prefetching can be as effective as data migration in improving the performance of commercial workloads [7].

Wenisch et al. extend the idea of predictability to mean the predictability of streams of data accessed by successive processors [9]. They show that shared data tends to be accessed in sequences which are repeated at other processors. They find that proactively streaming data to a processor repeating an earlier pattern can reduce cache misses in a set of scientific, commercial, and server workloads.

Circuit-switched coherence is another technique that exploits this predictable pair-wise sharing of data [10]. In this router protocol design, virtual circuits are dynamically created to connect processor pairs. This reduces router latency overhead on remote accesses between the processor pairs connected via the virtual circuits.

2) Reducing storage capacity demands: Because on-chip cache resources are limited and potentially shared, techniques which limit the retention of data in caches if that data is unlikely to be reused either before being evicted or within a certain time frame can potentially improve cache performance. Recent work has shown that level two cache miss rates can be improved by preventing or limiting the amount of data retained for applications with streaming data accesses or working sets larger than the level two cache capacity [16]. For shared caches, different data retention policies can be applied to different processes in order to maximize the use of the shared capacity [17]. Other approaches like cache decay can target the same types of data in order to reduce the used capacity in a cache by turning off lines that have remained idle for long periods of time [18].

3) Improving performance of other mechanisms: One of the difficulties of writing code for multiprocessors is understanding the memory consistency model supported by a given machine. BulkSC provides a mechanism for providing the appearance of sequential consistency to programmers even though the hardware may provide a relaxed consistency model [11]. In this technique, large chunks of instructions are committed atomically, reducing the overhead of maintaining sequential consistency. One of the performance optimizations for this approach is to reduce or eliminate the amount of tracking of reads and writes to private data

Application	Parameters
Apriori	-i data.ntrans_1000.tlen_20.nitems_1.npats_2000.patlen_6 -f offset_file_1000_20_1_P8.txt -s 0.0075 -n 8
HOP	61440 particles_0_128 128 16 -1 8
SVM-RFE	outData.txt 253 15154 15
ScalParC	para_F26-A32-D125K/F26-A32-D125K.tab 125000 32 2 8
K-means	-i color -b -o -p 8
Fuzzy K-means	-i texture100 -o -f -p 8

Table I
APPLICATION EXECUTION PARAMETERS

when determining whether to commit chunks of instructions. This approach works for both statically declared private data and for data that remains used by a single thread for extended periods of time. Although not mentioned in [11], one presumes similar optimizations could be used on shared data that was known to be accessed in a read-only fashion.

This paper examines data mining applications with the intent of understanding how data is used in order to determine the applicability of these different designs and optimization techniques. Specifically, we examine

- sharing levels,
- read/write behavior,
- duration and frequency of idle periods,
- predictability of data accesses, and
- predictability of pairwise communication.

III. Related work

A. Data mining characteristics

The computationally intensive nature of data mining applications make them amenable to running on chip multiprocessors, but their memory intensive characteristics have the potential to limit their performance. In order to study these applications, Narayanan et al. have provided a data mining application suite called MineBench [12]. MineBench contains fifteen applications, twelve of which are parallel, that implement algorithms for clustering, classification, association rule mining, optimization, and structured learning.

Work on understanding the characteristics of data mining applications has categorized them as both compute and memory intensive. Ghoting et al. [2] and Zambreno et al. [1] found that data mining applications execute a large number of floating point or integer operations and experience high level two cache miss rates due to the use of large data sets. Ozisikyilmaz et al. show that the combined compute and memory intensiveness of MineBench applications make these applications distinctly different from existing workloads including SPECInt, SPECFP, MediaBench, and TPC-H [3].

The large working set sizes of data mining applications result in poor level two cache performance [2]. Li et al. show that increasing lower level cache capacities can improve the memory system performance of these applications [4]. Shaw shows that the benefits of increased cache capacity come from the repeated reuse of data after long periods of idleness [19]. Jaleel et al. show that bioinformatic data mining workloads benefit from the greater total capacity of a single shared lower level cache compared to private caches due to the large working set sizes [5]. They additionally show that these parallel applications include large quantities of data shared by multiple threads.

In this work, we extend these characterizations of data mining applications to include analysis of whether existing optimizations will be appropriate for data mining applications. While some of our examination overlaps with this earlier work, such as observing the sharing level for data, our architecture independent analysis enables us to study a larger set of data characteristics than explored in earlier work on parallel applications. Our approach also enables us to consider whether software can assist hardware in being able to exploit specific data characteristics.

IV. Methodology

A. Workload

We examine six parallel applications from the MineBench [12] data mining suite. [1] These applications are written in C/C++ and parallelized using OpenMP. We compile them using gcc 4.2 for a 64 bit x86 machine running Linux. Table I shows the applications executed as well as their corresponding command line arguments. As this work analyzes complete application executions, we chose medium size inputs when possible and small inputs when even the medium size inputs resulted in excessive instruction counts.

B. Simulation infrastructure

We use Pin [20] to dynamically instrument the application executables in order to collect information about instruction execution and memory accesses. The Pintool we created for analysis keeps track of a global time, attributing one cycle of time to each instruction executed. Threads execute in parallel with global times synchronized at the beginning and end of parallel sections of code as well as barriers.

The Pintool instruments user-level instructions, tracking information about data on a 64 byte memory block basis. Additionally, it collects information about how read and

[1] Of the 12 parallel applications in MineBench, two do not compile with gcc 4.2 and three execute hundreds of billions of instructions on even their smallest inputs, making analysis of complete application runs intractable. The remaining application fails to execute when run on Pin using Pintools that perform locking in the Pintool.

write system calls impact data being tracked. Data collected for each memory access includes the thread performing the access, the address and size, and whether the access is a read or write.

V. Results

We begin our analysis by examining general application characteristics. Table II shows the instruction count, execution time, frequency of reads and writes, total memory footprint, and shared memory footprint for these applications. As documented in other studies of data mining applications, the number of memory accesses per instruction is high for all of the applications with most of the applications exhibiting higher fractions of reads versus writes. The total memory footprint sizes range from less than one to hundreds of megabytes. On-chip storage capacity, therefore, has the potential to impact performance for some of these applications.

Application	Instrs. (bil.)	Time (bil.)	Acc./Instr. (R/W)	Memory (MB)	Shared (MB)
Apriori	21.68	2.83	0.36 / 0.11	199.0	55.6
SVM-RFE	42.67	13.09	0.25 / 0.03	60.0	58.9
HOP	3.19	0.52	0.26 / 0.04	10.5	2.9
ScalParC	12.02	1.57	0.27 / 0.14	285.3	57.7
KMeans	59.24	7.62	0.26 / 0.03	1.8	1.0
Fuzzy	12.65	1.87	0.28 / 0.10	0.8	0.6

Table II
Overall characteristics

As one of the goals of this work is to understand how data is shared among threads, we removed all instructions that deallocate memory (i.e. delete and free) in the applications. The reason for this is that the memory allocator does not necessarily account for previous usage of data by a specific thread when reallocating blocks of memory. Consequently, memory blocks that are reallocated will potentially appear to be shared data when in fact they are not. Figure 1 shows how the number of non-shared and shared blocks change with this modification. This modification reduces the number of shared blocks in Apriori, HOP, and ScalParC. It does, however, increase the number of non-shared data blocks for all applications. The results we observe for non-shared data, therefore, must be tempered with the understanding that some of those blocks would be reused when a thread-aware memory allocator is used. We use these modified applications throughout the remainder of this paper.

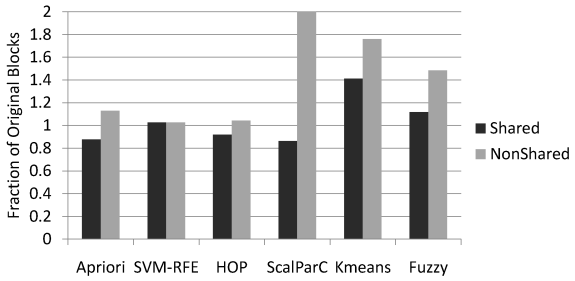


Figure 1. Impact of removing memory deallocation

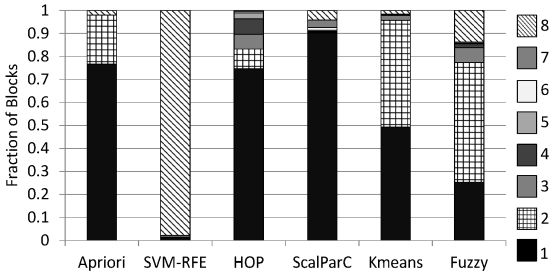


Figure 2. Levels of sharing

A. Levels of Sharing

Figure 2 depicts a breakdown of memory blocks in terms of the number of threads that access them. The fractions of blocks that are shared range from 10% of the blocks for ScalParC to 98% of all blocks in SVM-RFE. As seen in Table II, these fractions can amount to significant shared memory footprints depending on the total memory footprint size.

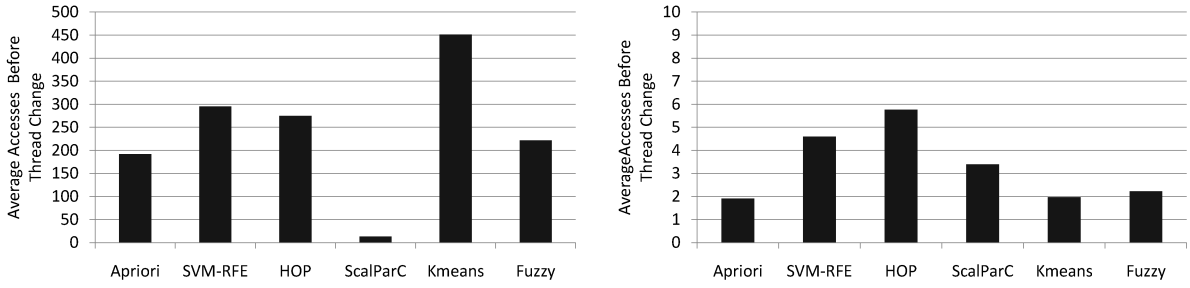
The distribution of the number of threads accessing each shared block also varies depending on the algorithm. Fuzzy KMeans has 52% of its blocks accessed by two threads where 96% of those blocks have the main thread as one of its accessors. Similarly, KMeans and Apriori have significant fractions of blocks accessed by two threads including the main thread. Applications where threads work independently and then the main thread combines their independent results will produce these types of sharing patterns. In contrast, SVM-RFE has most of its data accessed by all threads because most of the input set is used by all threads. HOP and ScalParC have a range of threads accessing their shared data.

The degree of sharing exhibited in the applications has implications for cache organization. For applications like SVM-RFE and ScalParC that have large amounts of memory shared between most threads, it may be better to have lower level cache resources organized as a single shared cache. Data replication in private caches may place too high of demands on total cache capacity for these applications. Other applications that have fewer numbers of threads sharing most shared blocks and smaller shared memory footprints like HOP may benefit from caches shared by small numbers of threads or data replication. As much of the sharing between two threads in KMeans and Fuzzy KMeans involves the main thread, caches shared by small numbers of processors may provide little benefit.

In the remainder of our analysis, we will group blocks into three groups - non-shared (*1*), shared between two threads including the main thread (*2 with 0*), and shared by many threads indicating all other thread combinations (*Many*).

B. Access Type and Frequency

In order to understand how data is shared by these applications, we determine which of our three groups of blocks generate the largest fraction of accesses. Figure 3



(a) 2 with 0 (b) Many

Figure 4. Thread access interleaving

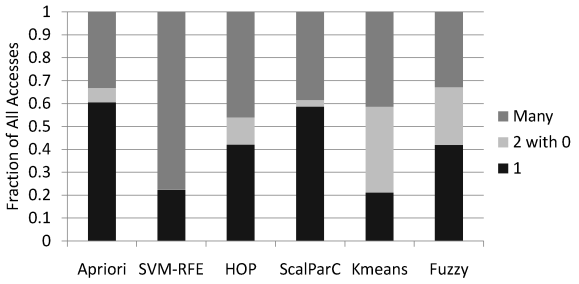


Figure 3. Breakdown of accesses

depicts the fraction of accesses to each block type. We observe that between 33 to 78% of accesses are to data that is accessed by our *Many* classification. This is true even for applications like KMeans and Apriori with small fractions of data shared by more than two threads. Thus, accesses to data shared among many threads are significant for all applications.

To understand whether accesses to shared blocks by different threads are finely or coarsely interleaved, we examine the average number of accesses to a block of data by a single thread before another thread makes an access to that block of data. Figure 4 shows these numbers for both groups of shared data. Data shared by many threads tends to be only accessed a few times before another thread starts accessing that block. In contrast, data shared between two threads (where one is the main thread) is accessed by a single thread for significant numbers of accesses before an intervening access by the other thread for all of the applications except ScalParC. This category of blocks in ScalParC represents less than 1% of blocks; based on the application's algorithm, most of these blocks would more appropriately be categorized in the Many category, which explains the lower number of accesses between thread changes.

These results have implications for caching and management strategies. The interleaving of accesses to blocks shared by many threads means that techniques like migration, which work best when there is some form of localized or temporary exclusive access to data, may not function well for these applications; shared data may simply migrate to the center of shared cache resources. Additionally, replication strategies may be useful depending on the number of blocks that need to be replicated, the number of times they are replicated, and how long they need to be replicated. The fine interleaving of accesses may also limit performance enhancing optimizations in modeling sequential consistency that rely on data being unshared for extended periods of time. However, this optimization may work well for data that is only shared between two threads where one is the main thread since the granularity of sharing is coarser.

Since several of the optimizations discussed in Section II make assumptions about whether data is read-only or read-write, we present information about writes. Figure 5 shows that most writes are to non-shared data. However, the number of writes to both classifications of shared data can be significant.

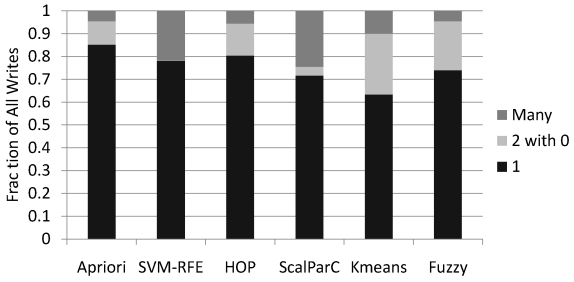


Figure 5. Breakdown of writes

As only writes to shared data will result in coherence traffic, we examine writes to shared data in more detail. As most of the input data is read in from files, just about all of the data blocks will be written at some point in time. Consequently, we try to determine the fraction of shared data that becomes read-only after initialization. To calculate this, we determine the lifetime of each block, meaning the amount of time between a block's first and last accesses. Figure 6 shows the fraction of shared data that is read-only after the first 10% of their lifetimes.

For some applications like Apriori, a large fraction of data shared by many threads becomes read-only after the first 10% of its lifetime. However, for many applications, a large fraction of the shared data is written during later portions of its lifetime. This suggests that optimizations that demand data be read-only may not be applicable to much of the shared data in these data mining applications. These optimizations include those that only replicate read-only

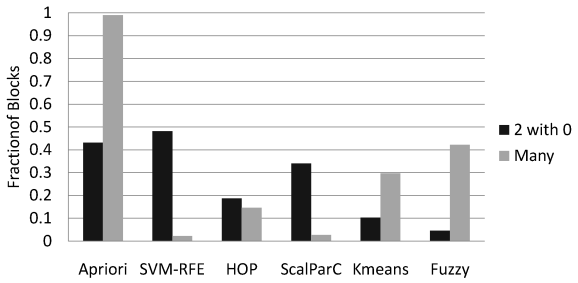


Figure 6. Read-only after initialization

data and potential performance optimizations for modeling sequential consistency which rely on data being read-only.

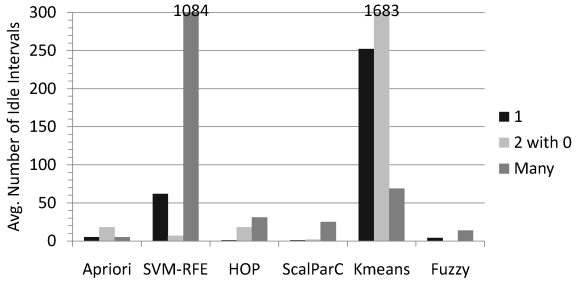


Figure 7. Idle Frequency

C. Data reuse

Techniques that selectively retain data in caches rely on data going unused for long periods of time. To gain an understanding of how data is used throughout the duration of the program, we analyze how often blocks go unaccessed for more than one million cycles and the duration of time during which blocks remain unaccessed. Combined, these metrics will help determine whether data tends to be used continuously throughout a program's execution or if data is used cyclically with long periods of idleness.

Figure 7 shows the average number of times the different groups of data blocks remain unaccessed for more than one million cycles. (We choose this time frame to designate idleness based on [18] which showed that it might be worthwhile to decay cache lines in a level two cache if they remained idle for more than one million cycles.) Data that goes idle multiple times is data that is used cyclically throughout its lifetime, going idle for the period of time between uses. This data is likely to be part of a large working set. We observe that data shared by many threads tends to go idle frequently for all applications except Apriori. Data that is shared between two threads including the main thread in Apriori, HOP, and KMeans also tends to go idle frequently.

We now examine the duration of those idle periods. Data that goes idle for longer periods of time may potentially be data that can either quickly be removed from a cache or may potentially never be inserted into a cache in order to deal with storage capacity issues. Figure 8 depicts the fraction of idle intervals that are at least one, five, or ten million cycles long. The number of idle intervals is normalized to the number of idle intervals of one million cycles.

Our first observation is that shared data accounts for a larger number of idle intervals than non-shared data. Additionally, while the fraction of idle periods extending longer than ten million cycles is significantly reduced from the number of idle periods lasting one million cycles, several applications including ScalParC, HOP, and Fuzzy KMeans still exhibit significant fractions of idle periods lasting longer than ten million cycles. In particular, more than 50% of ScalParC's idle periods last at least ten million cycles. In contrast, the idle periods in SVM-RFE generally last less than 5 million cycles, making it a less likely candidate for selective storage. We also note that the number of idle periods for non-shared data do not decrease as significantly as those for shared data for applications like Fuzzy KMeans and Apriori.

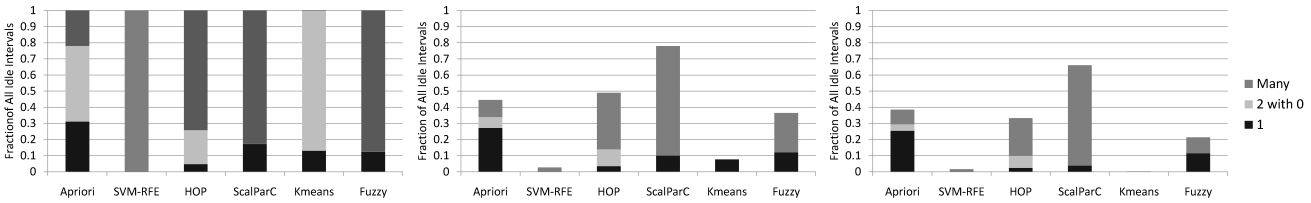
The implications for these results are that it may be possible to limit the total storage capacity needed on chip by using techniques like cache decay or adaptive insertion policies for some applications like ScalParC and HOP. As the different types of data exhibit different characteristics with respect to idle frequency and idle duration, it may also be beneficial to tailor these techniques to specific types of data as suggested as future work in [17]. For example, data in ScalParC that is shared by many threads goes idle frequently and for long periods of time, so it might be worthwhile to prevent it from being cached. In contrast, the non-shared data in ScalParC does not have many idle periods and less than 25% of these idle periods last longer ten million cycles. Consequently, you might want to insure that non-shared data in ScalParC is retained in lower levels of cache.

D. Access predictability

The optimizations used for reducing latency involve either prefetching data blocks to a given processor, proactively streaming data to a processor expected to use the blocks, or reducing the router overhead latency of communication between pairs of processors that use the same data blocks. All of these techniques rely on data accesses being predictable.

To first understand how predictable data accesses are within these applications without modeling caches, we determine the predictability of data accesses for individual instructions. For each instruction, we determine the predictability of data accesses using a simple data stride prefetcher [21].

Table III shows statistics for instructions that access more than one block of data per thread (Multiblock) and that have a data address prediction accuracy greater than or equal to 90%. We show the percentage of all program data accesses covered by these instructions as well as the percentage of all accesses made by instructions that may access multiple blocks. We also show the percentage of block changes covered; block changes occur when an instruction switches



(a) 1 Million Cycles (b) 5 Million Cycles (c) 10 Million Cycles

Figure 8. Idle periods

to using a different data block.

Application	Num. Insts.	% Accesses (All/Multiblock)	% Block Changes
Apriori	66	9 / 24	3
SVM-RFE	206	72 / 93	96
HOP	163	33 / 44	40
ScalParC	75	8 / 10	22
KMeans	52	35 / 44	34
Fuzzy KMeans	41	31 / 40	57

Table III
PREDICTABILITY OF DATA ACCESSES

The predictability of these applications varies. The set of instructions that have stride prediction accuracies of 90% or higher can account from between 3 to 96% of block changes and between 8 and 72% of the applications' accesses. Given the long periods of time between reuses of some shared data discussed in Section V-C, the use of a stride-based prefetcher may be worthwhile for these applications.

The results in Table III do not provide any information about whether or not the data blocks being accessed are likely in use by another processor that could potentially forward those data blocks in a streamlined fashion. To determine whether data may potentially be forwarded between pairs of processors, we keep track of the last thread to access each block of data. On a block change, we determine whether the new block was last accessed by the same thread as the previous block requested by this instruction (for a given thread). Table IV shows statistics for instructions that had prediction accuracies for both the address and the previous owner of at least 90%.

In addition to showing the percentages of accesses and block changes covered by these instructions, we also show the percentage of the block changes covered by these instructions which are data blocks that were last accessed by a thread that is not the current accessor. Applications like HOP and ScalParC that have large fractions of their data blocks that are non-shared have very few data blocks previously accessed by a thread different than the current accessor. Since KMeans and Fuzzy KMeans have large fractions of data shared by two threads including the main thread, most of their accesses will be to data the current thread last accessed. Streaming of data between pairs of processors is unlikely to help these applications. In contrast, SVM-RFE has 91% of these data blocks accessed by another thread. Since most data in SVM-RFE is shared by all threads, repeatable patterns exist and would benefit from pairwise streaming.

In summary, we observe that sharing levels for data vary for these applications and that most shared data is read-write. Additionally, some applications will benefit from capacity limiting approaches and most applications will benefit from some form of prefetching.

VI. DISCUSSION

The analysis from Section V shows that the applicability of any given optimization varies across different applications and sometimes even within a given application. For example, one type of shared data may be predominantly read-only after initialization while the other is not. This variability implies that the hardware will need to determine when certain approaches should be applied. Hardware can either dynamically learn this information or it can be provided with hints by software. Given that chip complexity and power consumption are concerns for future multi-core architectures, we ask whether software can potentially convey information to the hardware or somehow make it easier for hardware to dynamically classify data to determine when optimizations can be applied.

A. Conveying information via instructions

Some optimizations can be easily tied to specific instructions in an application. For example, some information could be tied to specific instructions to indicate when data prefetching should be used or to indicate when streaming of data should occur between multiple threads. Many of the approaches presented in Section II, however, rely on general characteristics of data which are established over time.

An important classification of data is based on its sharing level. We consider whether or not instructions can be used to indicate whether data blocks are non-shared, shared by two threads including the main thread, or shared by many threads. For every instruction, we track which data blocks it accesses. For every data block, we track which threads access it. We then determine if there is a set of instructions that indicates the degree of sharing predictably. Meaning, do certain instructions only access data of a certain type? Table V shows the number of instructions used to indicate each category of sharing and also what the coverage of blocks is for that category using that set of instructions.

Application	Num. Instrs.	% Accesses (All/Multiblock)	% Block Changes	% Block Changes Diff. Thread
Apriori	61	7 / 19	2	13
SVM-RFE	190	72 / 93	95	91
HOP	160	33 / 44	40	0
ScalParC	72	7 / 9	21	7
KMeans	50	9 / 7	5	0
Fuzzy KMeans	38	22 / 29	44	1

Table IV
PREDICTABLE STREAMING ACCESSES

Application	Shared By 2 Num. Instrs	Coverage of Blocks	Many Num. Instrs	Coverage of Blocks
Apriori	1	0.38	3	0.11
SVM-RFE	0	0	15	0.97
HOP	0	0	0	0
ScalParC	0	0	22	0.93
KMeans	0	0	13	0.36
Fuzzy KMeans	0	0	8	0.34

Table V
USING INSTRUCTIONS TO DESIGNATE SHARING

We observe that sets of instructions can be used to predict the degree of sharing for some applications. For example, a set of fifteen instructions can predict 97% of the data blocks shared by many threads in SVM-RFE and 22 instructions predict 93% of these same blocks in ScalParC. However, the coverage achieved for other applications is not as high. Part of the reason for this is that some instructions may predominantly access shared data, but if they access any non-shared data, they are disqualified as indicators of shared blocks. Allowing flexibility in our classification increases coverage of blocks (as well as the number of instructions) but comes at the cost of possibly miscategorizing some data. Still, this technique could be used for some applications where a small set of instructions was applied uniformly to all data blocks for a certain sharing level.

B. Using data addresses to convey information

Many of the existing approaches for determining how to manage data make their decisions on a block by block basis. However, Hardavellas et al. [8] and Ceze et al. [11] both assume the characterization of data is either dynamically discovered or provided at the page level. The problem with this approach is insuring that all data within a page has the same characteristics.

We suggest that the dynamic memory allocator (i.e. malloc) can assist with insuring that data with similar characteristics is placed in contiguous memory locations, enabling classification of data on a page level. The key to this approach is in realizing that different invocations of the memory allocator (i.e. different calls to malloc) allocate different types of data with different characteristics. However, if a memory allocation call is in a loop or in a method, the data dynamically allocated on each iteration of the loop or each call of the function is likely to have data characteristics (e.g. sharing level) similar to the data previously allocated at that same code location. If the memory allocator insures that data allocated within that loop or function is given contiguous memory addresses, then entire regions of memory will contain data with similar characteristics.

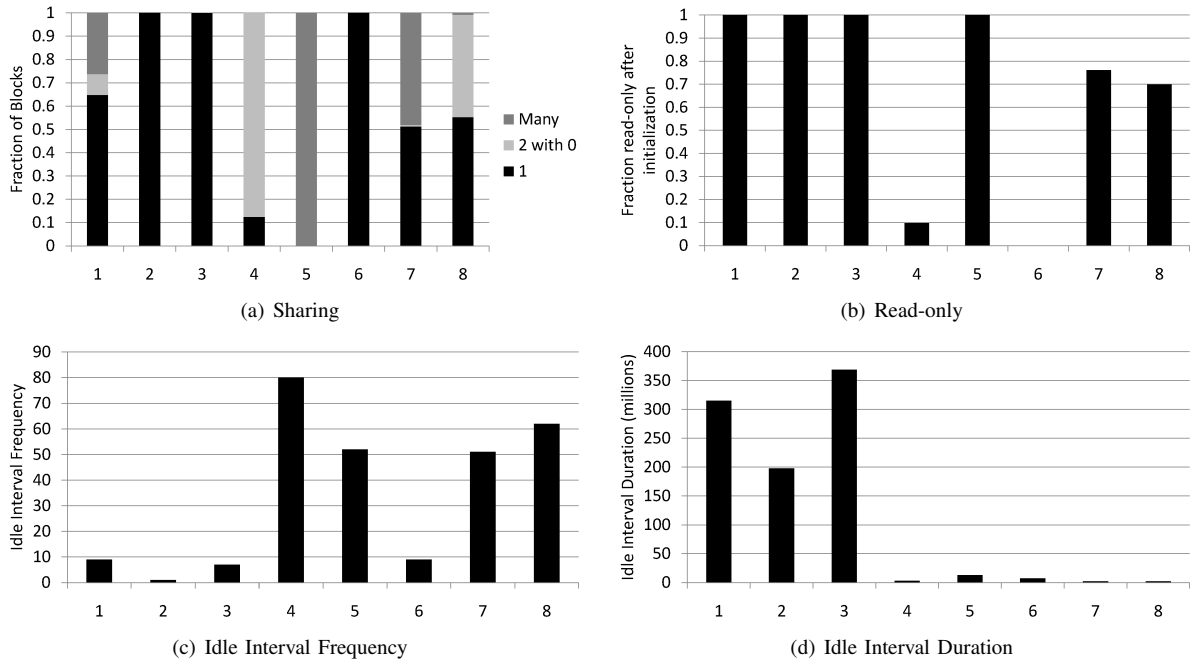
We look at dynamic memory allocation points in Apriori and HOP to illustrate this idea. We examine whether data allocated at the same source code locations have similar characteristics. Figures 9 and 10 show information for the eight memory allocation points that allocate the largest number of blocks in Apriori and HOP. For the data allocated at each allocation point, we show the sharing level, the fraction of data that becomes read-only after 10% of its lifetime, and the idle frequency and duration.

We observe that the allocation point can be indicative of sharing level. For example, the sharing level of data allocated at the fourth allocation point in Apriori will clearly be between two threads including main. In contrast, points 2, 3, and 6 in Apriori clearly indicate that data will be non-shared. Data allocated by points 4, 7, and 8 in HOP should clearly be considered shared by many threads.

Similarly, several allocation points in Apriori and HOP indicate that their corresponding data will become read-only after initialization. For example, data allocated at points 1, 2, 3, and 5 in Apriori and 1, 2, 5, and 7 in HOP can be considered read-only after initialization.

Finally, we observe that data allocated at different allocation points have different characteristics in terms of the number of times they go idle for more than one million cycles and their idle durations. For example, allocation point 7 in HOP goes idle frequently and has an average idle period of eight million cycles. Allocation point 4 in Apriori has data that goes idle frequently but remains idle for short periods of time. In contrast, data from allocation points 1 and 3 go idle for long stretches of time.

One possible approach to managing data is to restructure how memory is allocated so that data allocated at the



(a) Sharing (b) Read-only

(c) Idle Interval Frequency (d) Idle Interval Duration

Figure 9. Apriori

same instruction in the source code is actually allocated contiguously in memory. The results above suggest that doing so would result in contiguous regions of data that have similar characteristics. Information about the data characteristics for given pages could potentially be supplied by the application via directives to the hardware in order to inform the hardware of potential optimization strategies that would work on each page's data. Alternatively, the clustering of similar data into contiguous regions could make it easier for hardware to dynamically learn data characteristics at a page level instead of at a block level as there would be less variance of characteristics within a page. We intend to explore the performance impact of this approach in future work.

VII. Conclusions

Data mining applications are a growing class of applications that can take advantage of the increasing numbers of processors on chip multiprocessors. Their memory intensive nature, however, implies the memory system can significantly impact their performance.

In this paper, we examined the characteristics of data in parallel data mining applications in order to determine the applicability of existing memory system designs and performance enhancing techniques. We showed that the degree of sharing for data varies considerably across applications, suggesting the need for flexible cache organizations. We also showed that most shared data is read-write data and that most applications can benefit from some form of data prefetching. As some of these applications have large working sets, they can also benefit from approaches for reducing the retention of idle data in caches.

We also proposed a way to use the dynamic memory allocator (i.e. malloc) to help cluster data with similar usage characteristics into contiguous memory addresses. This approach would help enable the classification of data usage at the page level, making it potentially easier for hardware to determine which performance-enhancing techniques to apply to data.

Acknowledgements

We thank the anonymous reviewers for their feedback. We also thank Dan Upton, Barry Lawson, Carla Ellis, Margaret Martonosi, and Marianne Shaw for their comments on earlier drafts of this paper. This work was supported in part by the National Science Foundation under grant CCF-0702689.

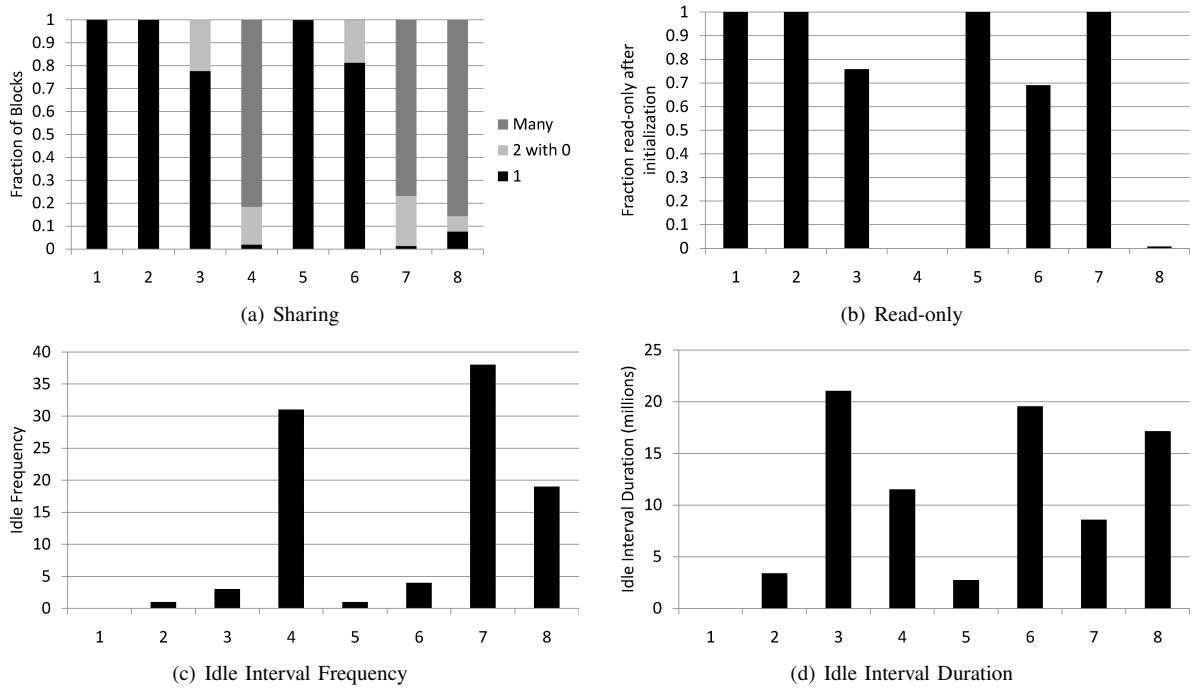
References

[1] 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.

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

[4] 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)*, 2007, pp. 35–43.



(a) Sharing (b) Read-only

(c) Idle Interval Frequency (d) Idle Interval Duration

Figure 10. HOP

[5] 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)*, 2006, pp. 250–260.

[6] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A nuca substrate for flexible cmp cache sharing," in *Proc. of the 19th Annual Intl. Conf. on Supercomputing*, 2005, pp. 31–40.

[7] B. M. Beckmann and D. A. Wood, "Managing wire delay in large chip-multiprocessor caches," in *Proc. of the 37th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, 2004, pp. 319–330.

[8] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive nuca: Near-optimal block placement and replication in distributed caches," in *Proc. of Intl. Symp. on Computer Architecture*, 2009.

[9] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Temporal streaming of shared memory," in *Proc. of the 32nd Annual Intl. Symposium on Computer Architecture*, 2005, pp. 222–233.

[10] N. D. E. Jerger, L.-S. Peh, and M. H. Lipasti, "Circuit-switched coherence," in *Proc. of the 2nd ACM/IEEE Intl. Symp. on Networks-on-Chip*, 2008, pp. 193–202.

[11] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: bulk enforcement of sequential consistency," in *Proc. of the 34th Annual Intl. Symp. on Computer architecture*, 2007, pp. 278–289.

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

[13] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 211–222.

[14] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *In Proc. of the 32nd Annual Intl. Symp. on Computer Architecture*, 2005, pp. 336–345.

[15] B. M. Beckmann, M. R. Marty, and D. A. Wood, "Asr: Adaptive selective replication for cmp caches," in *Proc. of the 39th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, 2006, pp. 443–454.

[16] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. of the 34th Annual Intl. Symp. on Computer Architecture*, 2007, pp. 381–391.

[17] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proc. of the 17th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2008, pp. 208–219.

[18] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power." in *Proc. of Intl. Symp. on Computer Architecture*, 2001, pp. 240–251.

[19] K. A. Shaw, "Understanding the working sets of data mining applications." in *11th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-11)*, 2008.

[20] 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 Conf. on Programming Language Design and Implementation*, 2005, pp. 190–200.

[21] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Proc. of the 25th Annual Intl. Symp. on Microarchitecture*, 1992, pp. 102–110.