

**Proceedings for
The Eleventh Workshop on Computer Architecture Evaluation using
Commercial Workloads (CAECW-11)**

**Immediately preceding the 14th International Symposium on High Performance
Computer Architecture (HPCA-14)**

Salt Lake City, Utah
February 17th, 2008

The function of this workshop is the discussion of work-in-progress that utilizes commercial workloads for the evaluation of computer architectures. By discussing this ongoing research, the workshop will expose participants to the characteristics of commercial workload behavior, provide an understanding of how commercial workloads exercise computer systems and help establish methodologies for measuring, modeling and analyzing the execution time characteristics of these workloads.

Final Program

8:00 – 8:30 am

Registration

8:30 am - 8:40 am

Opening Remarks

8:40 am - 9:30 am

Session 1: Keynote

Commercial Computing and High Performance Computing: How different are they?

Abstract: This talk provides an overview of application characteristics and the decision criteria for commercial computing and HPC. It covers general conceptions for HPC and commercial and why those are gross generalizations. It then compares and contrasts the different segments for each pointing out which segments have similarities and also the differences between those segments. It concludes by showing how systems that are designed for HPC can do well in the commercial space.

Don DeSota
IBM

9:30 am - 10:00 am

Coffee Break

10:00 am – 11:00 am

Session 2: Workload Characterization

Trends in Legacy and Emerging Commercial Workloads

Bill Maron, Bret Olszewski, Mala Anand, Steve Kunkel, and Thomas Chen
IBM

Understanding the Working Sets of Data Mining Applications

Kelly A. Shaw
University of Richmond

11:00 am – 12:00 pm

Session 3: Systems Modeling and Architecture

Write Update Optimizations for CC-NUMA Systems

Liqun Cheng and John B. Carter
Intel and University of Utah

An M/G/1 Queue Model for Multiple Applications on Storage Area Networks

Emmanuel Arzuaga and David Kaeli
Northeastern University

12:00 am - 1:30 pm

Lunch

1:30 pm – 3:00 pm

Session 4: Sensitivity and Scalability

Varying Memory Size with TPC-C: Performance and Resource Effects

Jay Veazey and Blaine Gaither
Hewlett-Packard

An Analysis of the Sensitivity of SPECjAppServer2004 to the Memory Sub-System

Rajeev Garg, Kumar Shiv, Andrew Sun, Mahesh Bhat, and Michael Jones
Intel

Scalability Study of a Java Application Server on Two Multi-core Systems
Yohei Ueda, Hideaki Komatsu, and Toshio Nakatani
IBM Toyko Research Laboratory

3:00 pm - 3:30 pm

Coffee Break

3:30 pm - 5:00 pm

Session 5: Memory Characterization

Memory Characterization of Emerging Recognition-Mining-Synthesis Workloads for Multi-Core Processors

Yu Chen, Aamer Jaleel, Wenlong Li, Junmin Lin, and Zhizhong Tang
Tsinghua University, Beijing, China and Intel

Characterizing Memory Behavior of XML Data Querying on CMP

Hong Liu, Rubao Li, Qiang Gao, Bihui Duan, and Taoying Liu
Graduate School of the Chinese Academy of Sciences, Beijing, China

Memory Characterization of SPEC CPU2006 Benchmark Suite

Junmin Lin, Aamer Jaleel, Yu Chen, Wenlong Li, and Zhizhong Tang
Tsinghua University, Beijing, China and Intel

5:00 pm

Participant Feedback

Closing Remarks

Workshop Organizers

- Adrian Moga, Intel (adrian.c.moga@intel.com)
- Russell Clapp, AMD (russell.clapp@amd.com)

Trends in Legacy and Emerging Commercial Workloads

Bill Maron Bret Olszewski Mala Anand Steve Kunkel Thomas Chen

IBM Corp.
Austin, TX

Throughout the history of computing the confluence of hardware and software trends has produced inflection points which proceeded changes in the information industry ecosystems. After a period of relative stability, an explosion of new hardware capabilities, software techniques, and business imperatives are spawning a more diverse set of applications. This paper examines some emerging workloads, their characteristics, and their relationships to traditional workloads.

1. Introduction

Each historic shift in application environment has been preceded by improving hardware performance and cost performance. These underlying trends have enabled environments such as wide-spread Java-based application infrastructures, which would have been unthinkable twenty years previous. Today's application emphasis points are no longer focused on improving the performance of simple, transactional tasks. Rather, focus has been placed on three primary areas:

- Reducing the cost of application development, delivery, and operation.
- Improving access and delivery of information.
- Leveraging existing information in optimization of business processes.

At the same time, the trend of ever increasing, complex single core microprocessors is at an end. Realities of technology scaling and power consumption have forced a rethink of priorities in hardware development. How should emerging workloads affect future system design?

2. Hardware development trade-off processes

Processor development has long used models to optimize proposed designs for performance, cost, power, chip area, schedule, complexity and resources [8]. However, it is obvious that these models are no better than the input with which they are driven. Hence, there is great interest in understanding new and emerging workloads. Because the processor development process takes several years and that there are usually derivative designs that ship for

several years after the initial design, the performance modeler would like to understand workloads many years in the future. This poses a serious problem because emerging workloads usually are understood only a few years in advance and often workloads don't even exist for emerging market segments. Industry standard benchmarks take years longer to become available.

Normally, traces or full system simulators are used to drive processor performance models. These provide a wealth of information to accurately drive the models of the caches, pipelines, and multiprocessor memory nest. However, these require a representative workload to execute. Often a representative, executable workload is not available. In this case, the characteristics of what is available can be compared to known workloads to find ways in which the emerging workload is more stressful or to find a surrogate workload that can be used (perhaps with some adjustments)[6]. If the emerging workload is not more stressful than known workloads, a robust design can still be achieved. If a surrogate workload can be found it can be weighted more heavily in the optimization process[7].

3. Methodology

While detailed instruction and data reference traces are irreplaceable for making low-level hardware decisions, they are expensive to collect and maintain currency. Additionally, they are frequently impractical to collect on customer systems. Those, coupled with the fact that trade-off's are easier to make with a small set of representative traces, leads to the obvious conclusion that a higher level filtering of workload characteristics is a necessity to cover a wide range of market segments and application domains.

Our approach collected detailed hardware performance monitor data on POWER5 systems which includes both micro-architecture "dependent" characteristics such as cache misses, as well as some of the "independent" characteristics such as number of loads/stores/branches of the instruction mix etc., of the workloads. This allowed efficient collection on a wide-variety of industry standard workloads, independent software vendor workloads, customer workloads and emerging workloads.

Following are a list of Micro-architecture dependent and independent characteristics collected for each workload:

- **CPI – Cycles Per Instruction, measured at the hardware thread. As there are two threads per core, the core CPI is approximately one-half of this value, depending on utilization**
- **IL1 – Instruction cache misses from the L1 cache per 100 instructions executed. The L1 instruction cache size is 64kB on POWER5**
- **DL1 – Data cache misses from the L1 cache per 100 instructions executed. The L1 data cache size is 32kB on POWER5**
- **IERAT – Instruction ERAT (Effective to Real Address Translation) misses per 100 instructions executed**
- **DERAT – Data ERAT (Effective to Real Address Translation) misses per 100 instructions executed**
- **ITLB – Instruction TLB (Translation Look-aside Buffer) misses per 100 instructions executed**
- **DTLB – Data TLB misses per 100 instructions executed**
- **Branch Misprediction Condition Register – The number of branches dispatched which are miss predicted on the basis of condition register values per 100 instructions executed.**
- **Branch Misprediction Target Address – The number of branches dispatched which are mispredicted on the basis of target address values per 100 instructions executed.**
- **Branch Instructions Dispatched – The number of branches dispatched per 100 instructions executed.**
 - ▶ Note: this metric as well as branch Misprediction due to Condition Register and branch Misprediction Target Address over count the actual completed instruction values due to speculation
- **Instructions per group – The number of instructions per completed group. The POWER5 architecture can complete 1 to 5 instructions per group. Groups do not necessarily complete each cycle**
- **IL2 – Instruction cache misses from the L2 cache per 100 instructions executed. The L2 cache is shared between instructions and data on POWER5 and is 1.9MB in size. Additionally, the L2 cache is shared between the two cores on each chip.**
- **DL2 – Data cache misses from the L2 cache per 100 instructions executed**
- **MM – Accesses to memory (local+remote) per 100 instructions executed. Since POWER5 systems have external L3 caches, the rate of memory accesses is less than the number of L2 cache misses**
- **Loads – Number of loads dispatched per 100 instructions executed**
- **Stores – Number of stores dispatched per 100 instructions executed**

4. Trends

This paper compares some emerging workloads to legacy ones, allowing us latitude to determine if the emerging workloads are fundamentally different than the current workloads used for hardware development trade-offs.

The categorization used for the collected workloads falls in one of the following segmentations:

- Application server - This group includes Java-based application serving solutions. The IBM Websphere product family is heavily represented in this segment.
- Business intelligence – This group includes both DB2-based TPC-H queries as well as other ISV solutions.
- OLTP database – This section includes traditional OLTP solutions from several major database vendors, including IBM's DB2.
- ERP – This group includes solutions from a number of ISV's, running quite a range of solutions from CRM, ERP, to hospital management.
- Infrastructure - This group includes webserving, file serving, mail serving, and a customer portal.
- Other – This group includes a traditional UNIX workload, a stand-alone Java benchmark, and IBM storage microcode.
- PHP – This group includes several PHP applications.
- Search – Indexing a search operations from IBM and 3rd parties.
- SOA – This group includes early service oriented architecture solution applications developed at IBM.
- SPECint_2006 – This group includes the individual workloads from the SPECcpu2006 suite. These measurements were not specifically tuned for POWER5.
- XML – XML parsing and database operations.

This paper addresses only commercial application segments and does not include high performance computing (HPC) applications.

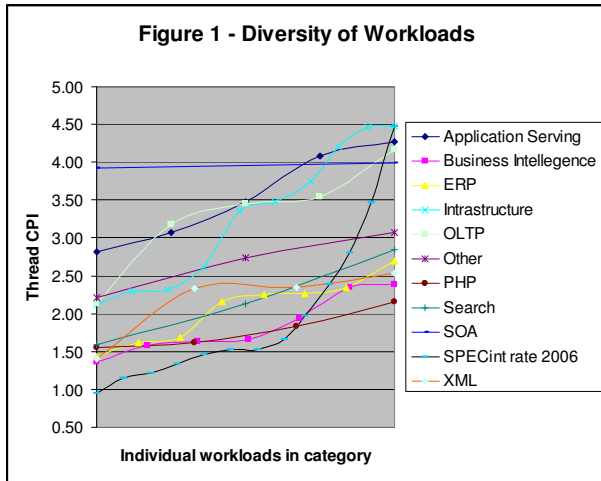
5. Comparisons

The diverse set of workloads that are analyzed in this study spans from application servers to XML workloads. To map all the workloads in a single domain, a single property that encompasses many of the workload characteristic namely CPI (Cycles per Instructions) is used. As shown in Figure 1, the workload domain consists of applications that have a wide range of CPI at

the same time some of the workloads clustered around similar CPI levels.

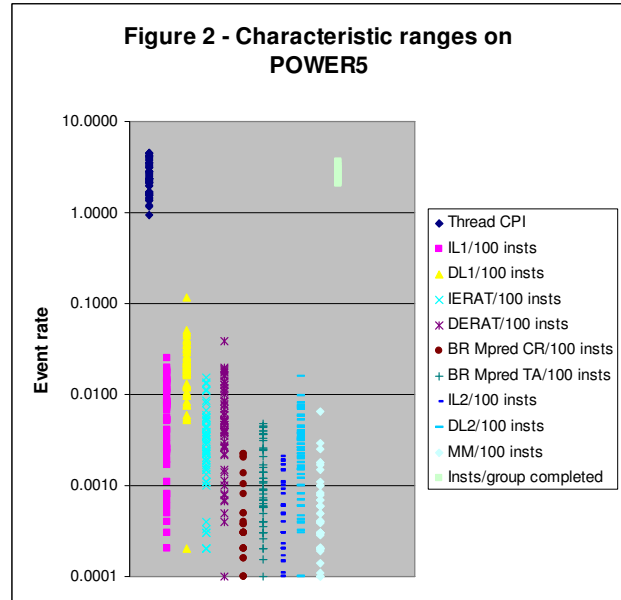
Industry Standard Benchmark SPECcpu2006 suite has the lowest and the highest point in the workload domain space covering a wide range of CPI. This indicates that SPECcpu2006 does represent a wide range of commercial workloads.

The emerging PHP workloads tend to have fairly low CPIs, XML based workloads cover wide ranges of CPI and application servers, SOA and infrastructure workloads tend to be at the upper end of the CPI range.



The CPI metric is just the tip of the iceberg with respect to variations in workload characteristics. Figure 2 shows that the range measured across our tracked characteristics is often on the order of two orders of magnitude. However, CPI experiences only a range of approximately one half order of magnitude. Thus, it is necessary to drill down much deeper than just CPI to identify characteristic similarity and dissimilarity.

In current hardware implementations, some workload characteristics such as instruction and data L1 cache misses may have small overall impacts to CPI if low latency L2 and L3 caches shield the processor from memory latency. Other characteristics, such as the average instruction level parallelism, as evidenced by instructions per group completed, tend to have a narrow range on POWER5.



6. PHP, scripting languages and Web 2.0

The term Web 2.0 is clearly ill-defined and overused, however it is impossible to ignore the explosion in the workloads created around open source software. While a diversity of software is available, the largest population of users engages the so-called LAMP stack which includes Linux, Apache, MySQL, and PHP. In LAMP based implementations, the bulk of the CPU usage is in applications designed and implemented with the PHP scripting language.

Grasping the nature of PHP applications requires confronting the confounding permutations of the stack. Table 1 shows some of the variables considered during measurement and analysis on POWER. Interestingly, the factors of PHP level, operating environment, and compiler turned out to be mostly irrelevant from the perspective of characteristics of the workload on the hardware. However, significant differences occur between non-accelerated PHP and accelerated PHP. As a general rule accelerated PHP results in higher event rates and higher CPI. This is partially a consequence of that accelerated PHP spent more execution time in the operating system and memory management.

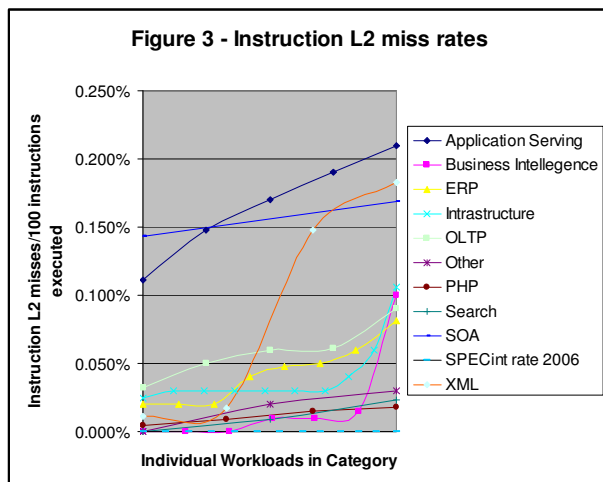
Application	PHP level	Accelerator	Operating environment	Compiler
ezPublish	PHP4	None	Linux	gcc
Wikipedia	PHP5	Xcache eAccelerator APC Zend	AIX	IBM XLC/C++

Table 1 – PHP application permutations

A rich variety of other scripting languages also participate in this space, including Python, Ruby, and Perl. We haven't had time to explore these with the level of detail applied to PHP.

7. Do emerging workloads have unique characteristics?

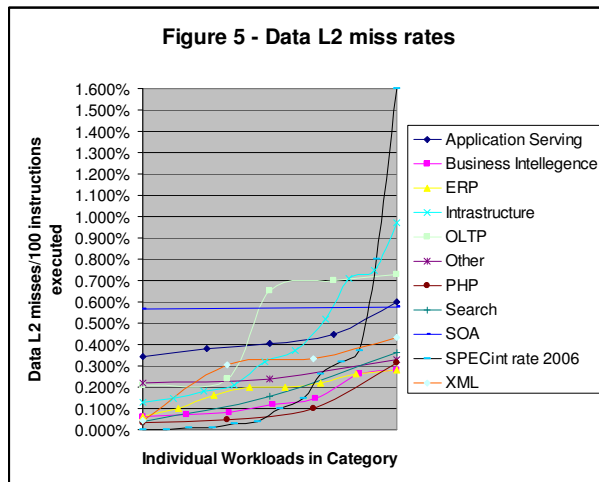
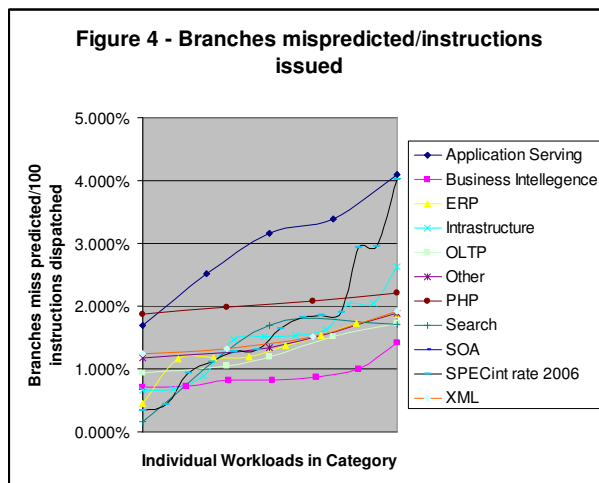
An item of interest in workloads is their effective footprint in cache. Given the reasonably generous on-chip L2 cache of POWER5, the L2 instruction miss rate is fairly indicative of code footprint size. Figure 3 shows that our application serving and SOA applications have consistently high L2 instruction cache miss rates. The database oriented XML applications also have relatively large code footprints. PHP and search have fairly low L2 instruction cache miss rates and the SPECint rate 2006 applications have negligible L2 instruction cache miss rates.



Interestingly, instruction L2 miss rates are not necessarily a good predictor of branch prediction complexity. Figure 4 shows that the application serving segment tends to have high branch misprediction rates, but so does PHP, which had very low instruction L2 miss rates. Also, while most of the SPECint 2006 suite has good branch prediction performance on POWER5, a few programs are problematic.

The L2 data cache miss rate for an application tends to have more distinct triggers than the instruction cache footprint. Diverse factors including data footprint, data access pattern, and context switch rate can all come into play. Figure 5 shows the L2 data cache miss rates for our workload samples. The highest point is represented by the SPECint rate 2006 mcf1 program. As expected, OLTP also drives relatively high rates, a result of the high multiprocessing levels and context switches for I/O. Also interestingly, high volume web-serving

infrastructure based Java servlet also can push high miss rates.



Memory accesses on POWER5 are filtered by the 36MB per chip L3 cache. Generally, workloads that have high memory hit ranges on POWER5 have either very large footprints or very little cache reuse. Figure 6 shows the memory hit rates for our workload samples. Interestingly, the same SPECint 2006 code that afflicted the data L2 miss rate samples, mcf1 has by far the highest memory access rate. Clearly, if systems are designed to the most pathological cases, the memory bandwidth required for a system with a POWER5 sized L2 would be about 50% higher than required for any of the other samples. The situation is even worse for the larger 36MB cache, with the peak memory bandwidth being more than thrice the requirements of any other workload sample. Not surprisingly, OLTP, Search, and Business intelligence demand relatively more bandwidth than the other workload sets.

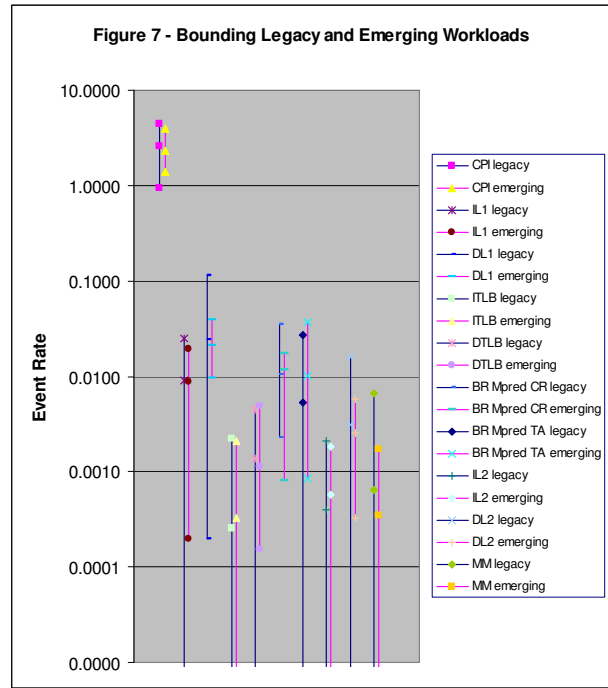
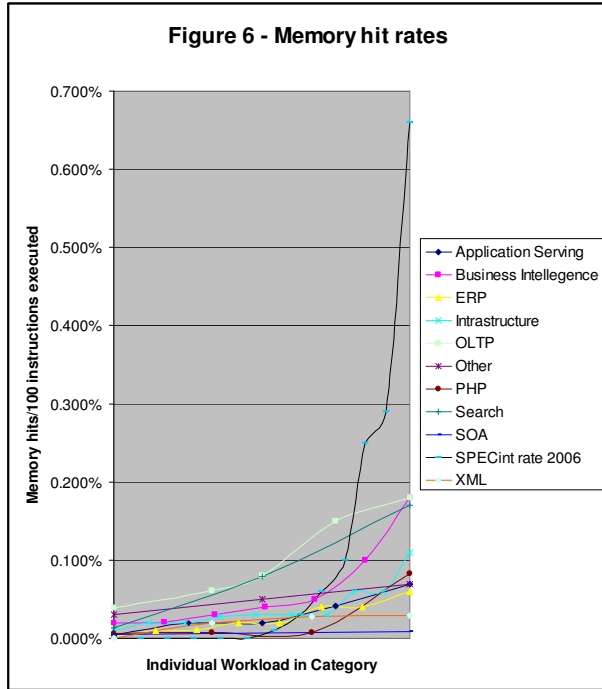


Figure 7 shows a comparison of low, average, and high values across a set of our workload characteristics for the fifty-one legacy and thirteen emerging workloads used for this study. Bounding by categories by low and high values we find that legacy workloads have more extreme characteristics than the emerging workloads evaluated in every category but two. And those categories, the high values for emerging workloads on DTLB misses and branch miss predict for target addresses are not much higher than the highest legacy values. Table 2 summarizes the difference in average values for the emerging workloads compared to the legacy workloads. Though averages are very difficult to use profitably, it appears that emerging workloads instruction footprints and branch behavior are generating higher event rates than on the legacy workloads. However, the memory bandwidth requirements of the emerging workloads is less than legacy workloads.

Metric	Average %change in emerging workloads
CPI	-7.44%
IL1	+21.73%
DL1	-9.87%
ILTB	+22.11%
DTLB	-14.82%
BR Mpred CR	+10.97%
BR Mpred TA	+58.62%
IL2	+31.43%
DL2	-15.66%
MM	-39.45%

Table 2 – Change in average metrics

Though we must remain vigilant, it appears that for the time being the legacy workloads we've been using to evaluate future systems are more or less relevant to the emerging workloads we've investigated.

8. Virtualization Technologies

Characterization of workloads in a virtualized environment using technologies such as Advanced Power Virtualization (APV) [1,4] and AIX workload partitions [5] (WPAR) brings out a diverse set of additional properties which in turn highlights a different set of system requirements. APV allows multiple partitions to time share the underlying microprocessors, cache, memory, and I/O subsystems. WPAR virtualization is done at Operating System level (AIX) and enables multiple workload partitions to share a single operating system image that manages the underlying hardware resources. In both cases, the context switching of workloads on the microprocessors leads to less reuse of cache and a corresponding increase in cache misses. However with WPAR virtualization, the single operating system image leads to some natural sharing of pages across multiple workloads, reducing the impact of context switches.

Figure 8 compares the increase in each of the measured hardware events when seven homogenous workloads are consolidated first using APV and then WPAR technologies over single workload measurements on a single system image. For these measurements, we did not enable direct sharing of pages between APV partitions and did not share middleware

and workload code among WPAR partitions. This approach was primarily to understand impacts to heterogeneous workload environments. In the APV case, every measured hardware event is increased and the rate of completed instructions reduced. It is interesting to note that in WPAR case, consolidated workloads cause a decrease in L1 instruction and data cache misses, translation cost and the branch prediction cost while L2 caches misses and memory accesses are increased. The access patterns of the application serving Trade6 workload on seven Workload partitions (WPAR) affect the MRU ranking of the lines in the L1 cache leading to a reduction in the L1 miss rates. However, this change in MRU ranking is not reflected in the L2 and L3 caches. Also, partitions lead to more distinct (less sharing) data working sets increasing L2 and L3 miss rates. Therefore L2 cache misses increase even though L1 misses are reduced. As the number of workloads are increased the pressure on L2, L3 caches increases in both cases resolving misses in memory. For brevity reasons data from one to seven partitions incremental data is not shown here. Even though the L2 and L3 on POWER5 are reasonably large, the large footprint workloads, and its access patterns could fill up the caches quickly at every partition switch causing memory access to increase. Moreover POWER5 has shared L2 and L3 caches shared by both the threads and cores in the chip which exacerbate the pressure on these caches.

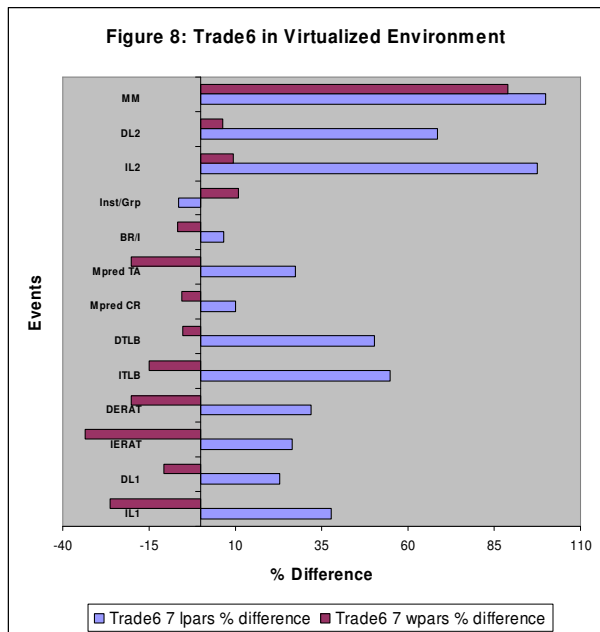
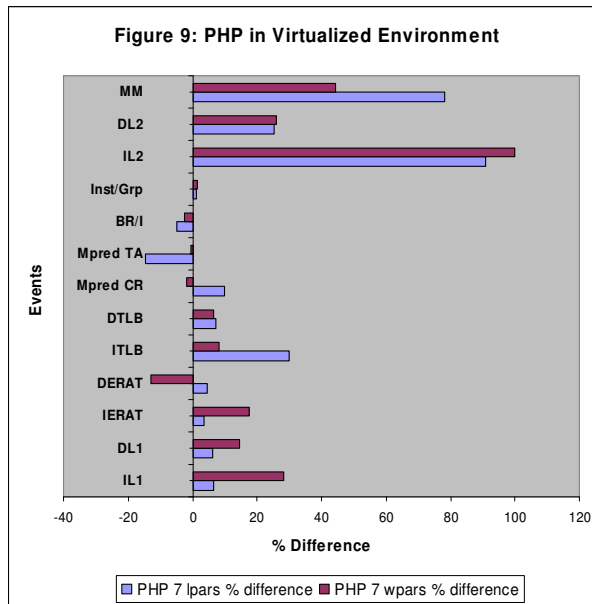


Figure 9 characterizes a PHP workload in both virtualization technologies. PHP which is a low instruction footprint workload shows higher cache miss rates both instruction and data in virtualized environment. It is interesting to note that instruction misses are higher

in both L1 and L2 but the memory access is lower in case of WPAR indicating L3 is the point of resolution for most of the L2 misses. That is, the workload's instruction footprint is still contained within the caches for WPARs, but less so for APV. This again points out that WPAR shared pages have an edge over non-shared page model. Most importantly, both forms of virtualization tend to increase the memory bandwidth required of systems, something critical to consider in system design.



9. Conclusion

This paper has shown a way to grasp the characteristics of emerging workloads during their early evolution by taking advantage of some of the key components of the CPI stack. This provides a great deal of insight into the system level behavior of these workloads at a fraction of the investment of the long drawn process of simulation and modeling which is impossible or non-existent. The emerging workloads examined run a gamut of characteristics, however they are fairly well bounded by legacy workloads, allowing surrogate workload strategies for future system evaluation. A bigger factor in future system design is likely the impact of various virtualization technologies, which will present a heavier load on memory subsystems.

References

- [1] Scott Vetter, Bill Adra, Annika Blank, Mariusz Gieparda, Joachim Haust, Oliver Stadler, Doug Szerdi: *Advanced POWER Virtualization on IBM eServer p5 Servers: Introduction and Basic Configuration*, IBM Redbooks
- [2] Joel M. Tandler, Steve Dodson, Steve Fields, Hung Le, Balam Sinharoy: *IBM eServer Power4 Microarchitecture*, Technical White Paper (<http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.pdf>)
- [3] Bret Olszewski, Octavian F. Herescu: *Performance Workloads in a Hardware Multi Threading Environment*, Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads, February 2002 (<http://tesla.hpl.hp.com/caecw-02/s3p2.pdf>)
- [4] Benn Gibbs, Dr. Balaji Atyam, Frank Berres, Bruno Blanchard, Lancelot Castillo, Pedro Coelho, Nicolas Guerin, Lei Liu, Cesar Diniz Maciel, Ravikiran Thirumalai, Dr. Carlos Sosa, *Advanced POWER Virtualization on IBM eServer p5 Servers: Introduction and Basic Configuration*, IBM Redbooks
- [5] Bruno Blanchard, Pedro Coelho, Mary Hazuka, Jerry Petru, Theeraphone Thitayanum and Chris Almond: Introduction to Workload Partition Management in IBM AIX Version 6.1, IBM Redbooks <http://www.redbooks.ibm.com/redbooks/pdfs/sg247431.pdf>
- [6] H. Vandierendonck and K De Bosschere, "Experiments with Subsetting Benchmark Suites" in WWC-7, Oct, 2004, pp. 55-62.
- [7] L. Eeckhout, J.Sampson, and B. Calder. Exploiting program microarchitectre independent characteristics and phase behavior for reduced benchmark suite simulation. In Proceedings of the 2005 IEEE International Symposium on Workload Characterization (IISWC), pages 2-12, Oct 2005
- [8] S. Kunkel, R. Eickemeyer, M. Lipasti, T. Mullins, T. O'Krafta, H. RosenBerg, S. VanderWiel, P. Vitale, and L. Whitley, *A Performance Methodology for Commercial Servers*, IBM Journal of Research and Development, November 2000, Vol. 44, No. 6, pp. 851-872.

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, SPEC FP, 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 increas-

ing 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.

¹This 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-

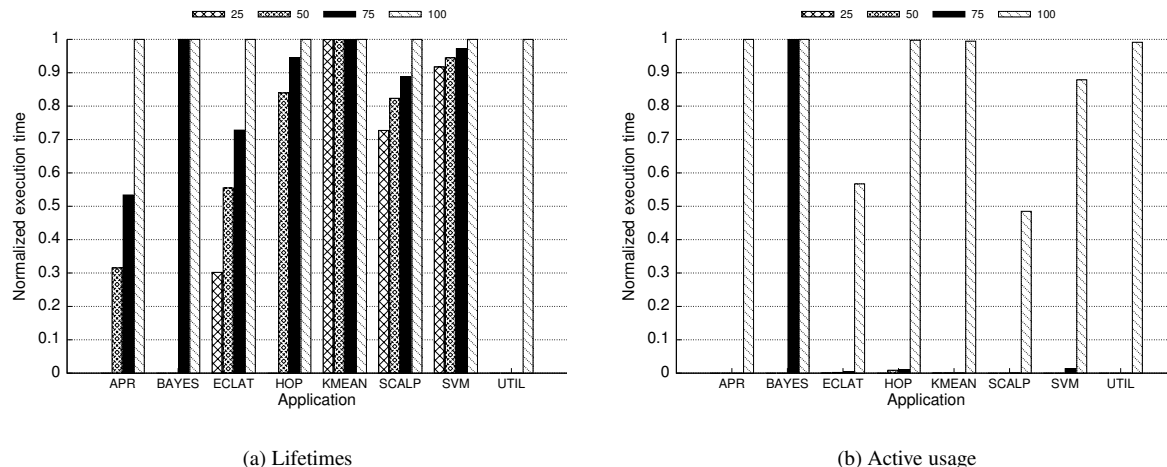


Figure 1. Data lifetimes(a) and total active use(b)

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

²This 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.

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

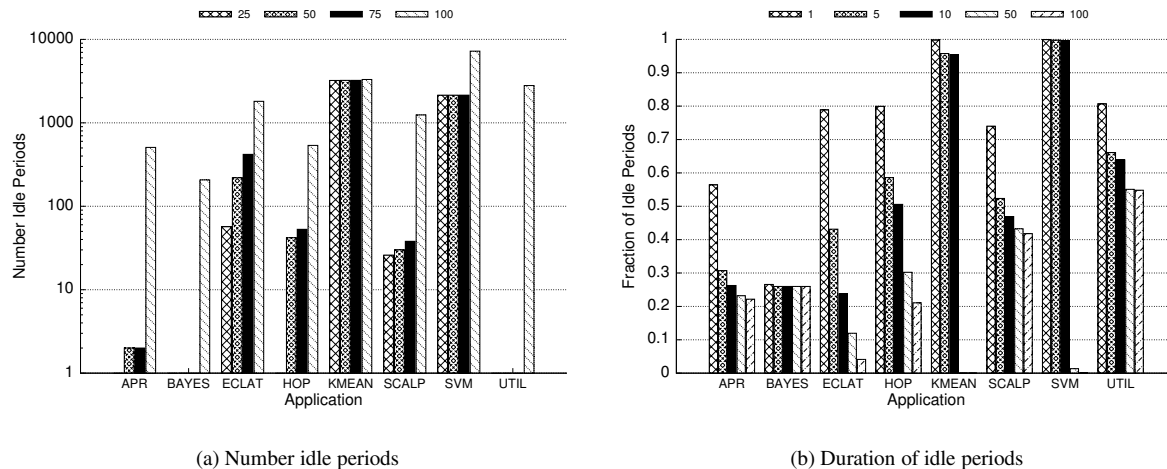


Figure 2. Frequency(a) and duration(b) of idle periods

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 con-

tinue 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.

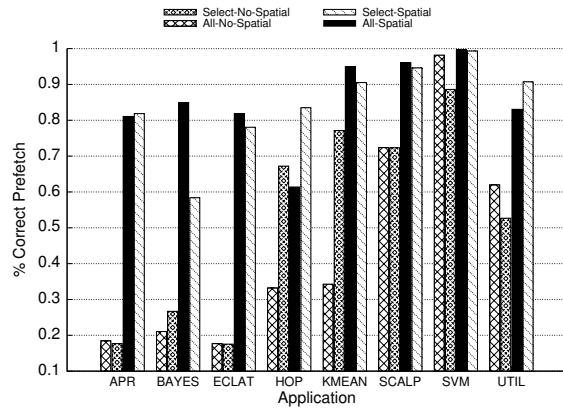
Given these concerns about the quantity of storage capacity needed by these applications' working sets and power consumption, it might be not be possible to retain the entire working set in a large on-chip cache. Techniques that can prefetch data to reduce latency will be necessary. In the next section,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

5. Prefetching

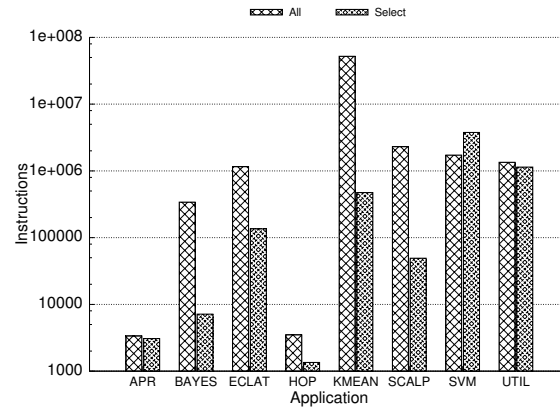
In order to determine the likelihood that prefetching can be used successfully without modeling a specific memory hierarchy, we collect statistics for every instruction about the distances between subsequent non-stack memory references. If the distance between two memory references is repeated between 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 in-

structions 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.

Write Update Optimizations for CC-NUMA Systems

Liqun Cheng
Intel DEG
liqun.cheng@intel.com

John B. Carter
University of Utah
retrac@cs.utah.edu

Abstract

Conventional CC-NUMA machines employ write-invalidate protocols, which induce read misses to ensure coherence. Previous research has shown that an update protocol can be much more efficient than invalidation when the data is in predictable patterns. However, update protocols are difficult to build on a scalable interconnect and often generate excessive network traffic.

We propose a speculative sequentially consistent write-update mechanism on top of a write-invalidate protocol. To ensure correctness, a processor wishing to write obtains exclusive ownership through a traditional write-invalidate protocol. To improve performance, the writing processor can later self-downgrade the modified block and flush it back to its home node, which forwards the latest value to processors that are likely to consume the data. We also present two hardware-efficient mechanisms to detect access patterns that benefit from the update optimization, stable reader set and stream.

We evaluate our update mechanisms on a wide range of scientific benchmarks and commercial applications. Using a cycle-accurate execution-driven simulator of a future 16-node SGI multiprocessor, we find that the mechanisms proposed in this paper reduce the average remote miss rate by 30%, reduce network traffic by 15%, and improve performance by 10%.

1 Introduction

Advances in semiconductor technology have led to a tremendous increase in the clock speed and transistor counts of single-chip devices. However, decreases in interprocessor communication and memory latency have not kept pace with increases in processor speed. This growing processor/memory performance gap has a significant impact on shared memory multiprocessor, where a coherence miss often requires traversing multiple cache hierarchies and incurs several round trip message latencies.

Existing CC-NUMA systems maintain coherence using directory-based write-invalidate protocols. Write-invalidate

protocols are efficient for situations where data is used exclusively by a single processor or mostly read-shared, but is inefficient for other sharing patterns, like producer-consumer and widely-shared data. For example, many scientific applications use successive over-relaxation (SOR), in which a large array is partitioned amongst cooperating processors. During alternating iterations of the SOR algorithm, the boundary data elements are modified by one processor and subsequently consumed by neighboring processor(s). When a consumer accesses the data, it incurs an expensive 3-hop miss: (1) a message to the home node (2) a snoop message from the home node to the producer node (3) a data forward message from the producer to the consumer.

We propose a lightweight write-update mechanism on top of a traditional write-invalidate protocol to reduce the number of remote misses. Our speculative write-update optimization works as follows. In the above SOR example, when the producer wishes to modify a boundary element, it obtains an exclusive ownership through a traditional write-invalidate protocol. Meanwhile, the home node determines whether the data being written should be forwarded to (likely) consumers before they suffer a read miss. If the home node determines that this write should trigger the speculative update mechanism, it indicates this as part of its response to the producer. Upon receiving this response, the producer write-through the new value by *delayed intervention*, which first downgrades the new contents of the cache line, then writes back the data to the home node. Finally, the home node can send *speculative update* messages to nodes that it predicts are likely to consume the data.

Traditional write-update protocols often send updates that are no longer useful. Excessive updates lead to serious performance problems by consuming precious interconnect bandwidth. To mitigate this problem, we employ simple hardware mechanisms to detect two access patterns likely to benefit from updates, *stable reader set* and *stream*. Recent research has shown that update protocols can improve the performance of the producer-consumer sharing pattern [3, 20]. Our *stable reader set* detects when the data is accessed in a producer-consumer fashion, even when there are multiple producers. Other recent research iden-

tifies sets of cache blocks that are accessed in consistent streams [17, 19, 20]. Data in the same stream often exhibit similar sharing behavior, so if one block benefits from an update protocol, the rest are likely to benefit also.

2 Protocol Implementation

2.1 Write Update Framework

We decouple coherence into a (write-invalidate) correctness substrate and a (speculative write-update) performance substrate. The correctness substrate employs the write-invalidate protocol to guarantee coherency invariants like *write atomicity*. The performance substrate exploits the write-update protocol to push (write through) modified data to where it is likely to be consumed. The decision to employ update optimization is determined by the home node.

2.1.1 Delayed Intervention

In collaboration with SGI, we are investigating performance optimizations that can be deployed in near-term commercial systems. Therefore, we only consider designs that require no processor modifications. This restriction makes it tricky to initiate updates. To get around this problem, we employ a *delayed intervention* mechanism. Figure 1(a) shows the coherence operations induced by the producer-consumer sharing pattern in a traditional write-invalidate protocol, where both the producer and consumer incur 3-hop misses. Figure 1(b) shows the coherence operations after we introduce delayed interventions. New coherence messages are indicated with bold lines and coherence messages eliminated are indicated with dashed gray lines. When the producer wishes to modify a cache line, it acquires the exclusive ownership using the baseline write-invalidate protocol. However, if the home node chooses to trigger the write-update optimization upon receiving the read exclusive request, its reply to the requesting node includes a delayed intervention flag. Upon receiving the reply message, the producer’s memory controller treats it as an exclusive reply, but also registers a delayed intervention. After the producer’s memory controller collects all acknowledgement messages and grants the processor exclusive ownership, it activates the delayed intervention operation, which entails waiting a short period and then sending an downgrade intervention request to the processor. In response to the local intervention message, the producer downgrades the cache line from *exclusive* to *shared* state and sends an implicit write back (IWB) message to the home node with the cache line’s new contents. Subsequently, when consumers read the data, the home node can serve the request directly without snooping the producer.

2.1.2 Speculative Updates

When the home node receives the data written back from the producer node, it can use *speculative updates* to propagate the data to where it is likely to be used. If we correctly push data to where it will be consumed, the consumer will suffer a local miss instead of an expensive remote miss. To support speculative updates, we need to first provide a means to push unrequested data to a processor, then determine when and where to push the data (Section 2.2). Researchers often assume the ability to push data into processor caches, but this capability is not supported by modern processors. To support pushing unrequested data, we augment each node with a remote access cache (RAC). In our design, RACs can hold victimized remote data [10], but are primarily used as a target for remote pushes.

Figure 1(c) shows the flow of speculative updates. Bold lines represent new coherence messages sent in the speculative update protocol that are not present in the baseline protocol. Solid gray lines represent messages saved compared to employing only the *delayed intervention* mechanism. Dotted gray lines together represent coherence messages saved compared with the baseline write-invalidate protocol. The protocol is identical to the delayed intervention (only) design until the home node receives the IWB message, at which time it forwards the data to the RACs on nodes where the data is likely to be consumed. The home node keeps track of the set of nodes to which it has pushed the data by adding them to the corresponding sharing vector. If the home node correctly predicts which node(s) subsequently consume the data, the consumers read the shared copy directly from their local RACs.

2.1.3 Verification

We used the Murphi model checker [5] to verify that our performance optimizations do not violate any coherence correctness invariants. We extended the DASH protocol model provided as part of the Murphi release and performed an exhaustive reachability analysis for a small configuration. We found that none of the invariants provided in the DASH model are violated by our changes. Moreover, we extended our architecture (performance) simulator to check two invariants, *single writer exists* and *consistency within the directory*, at the completion of each transaction that incurs a L2 miss. These invariants held for all of our experiments.

2.2 Determining When/Where to Push Updates

2.2.1 Stable Reader Set

Adaptive coherence protocols have been proposed to optimize migratory [4, 18], producer-consumer [3], and wide [9] sharing. However, many applications have multiple dominant signatures that tend to fool adaptive protocols designed to identify specific access patterns. Unstructured

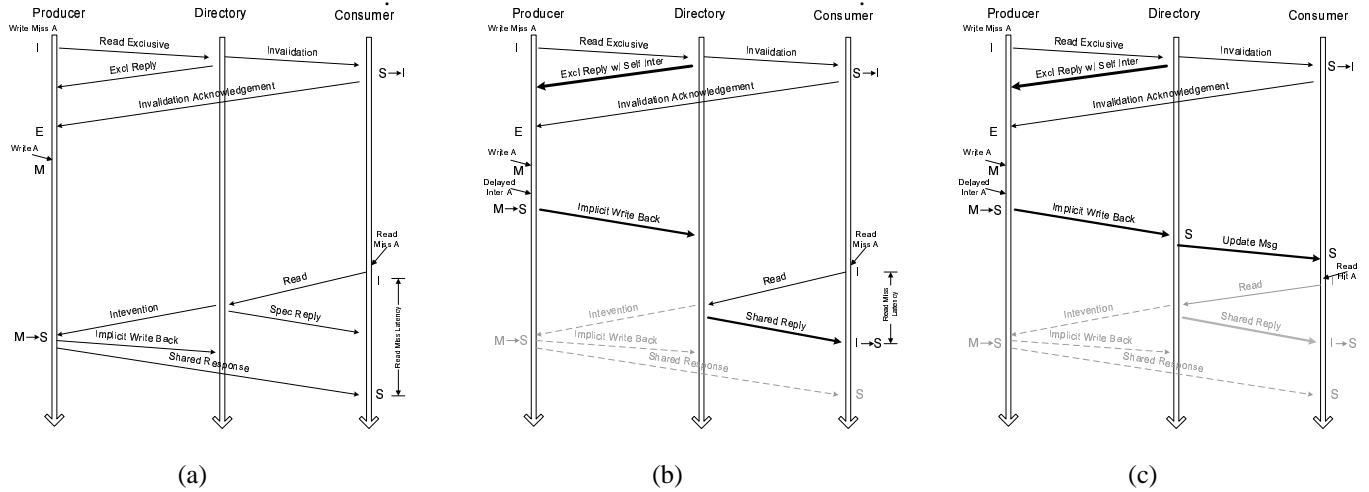


Figure 1. Flow of base case (a) , delayed intervention (b), and speculative update (c).

State	Sharing Vector	Stable Counter	Stable Reader Set
5b	16b	3b	16b

Directory cache entry (40bits)

Figure 2. Format of directory cache entries

is a computational fluid dynamics (CFD) application that models an airplane wing using an unstructured mesh. Its access pattern oscillates between migratory and producer-consumer sharing. To exploit this pattern, we propose a hardware mechanism to detect a *stable reader set*.

We define stable reader set sharing as a repetitive pattern wherein multiple processors belonging to a stable set can write and/or read the block arbitrarily, which corresponds to the following regular expression:

$$\dots(W_i)(R_{\forall j:j \neq i})^+(W_k)(R_{\forall m:m \neq k})^+ \dots i, j, k, m \in S \quad (1)$$

R_i and W_i represent read and write operations by processor i , respectively.

To detect stable reader sets, we extend each node's directory controller to track the access histories. We minimize space overhead by only tracking the access histories of blocks whose directory entries reside in the directory cache. A typical directory cache might contain 8K directory entries. We find that tracking only their access histories detects the majority of the available sharing patterns, which agrees with similar results reported by Martin [12].

We add two fields to each directory cache entry: a *reader set* and a *stable counter*, as shown in Figure 2. This extra state is not saved if the directory entry is evicted and flushed to main memory. The *reader set* field employs the same data structure as the sharing vector, a full bitmap structure. Figure 3 shows an example in which processors 1, 2, and 3 share the data at address A. Each processor modifies

Access history of address A: (W1, R2, R3, W2, R1, R3, W3, R1, R2)*

	Tag	State & sharing vector	Reader set	Stable counter
1) proc 1 write	A	E/M 1	1	000
2) proc 2 read	A	S 1,2	1,2	000
3) proc 3 read	A	S 1,2,3	1,2,3	000
4) proc 2 write	A	E/M 2	1,2,3	001
5) proc 1 read	A	S 1,2	1,2,3	010
6) proc 3 read	A	S 1,2,3	1,2,3	011
7) proc 3 write	A	E/M 3	1,2,3	100
...				
N) proc 3 read	A	S 1,2,3	1,2,3	111
N+1) proc 1 write	A	E/M 1	1,2,3	111

Figure 3. Stable reader set prediction process

the data in turn, and each modification is immediately consumed by the other two processors. When a processor is added to the sharing vector, it is also added to the reader set. However, when a processor writes the data, e.g., step 4 in Figure 3, the sharing vector is overwritten by the current owner's ID, while the reader set is not modified. Thus, the reader set is always a superset of the sharing vector. The *stable counter* is a 3-bit saturating counter that is incremented each time the home receives a new request from a node in the current reader set and decremented each time it receives a request from a node not in the current reader set. A memory block is regarded as having a *stable reader set* whenever the stable counter is saturated. When the home node receives a read exclusive request while the block has a *stable reader set*, it sets the delayed intervention bit in the reply message. When the newly written data is flushed from the producer (processor 1 in this case), the home node pushes the data to the RACs of all other processors in the *reader set* (processor 2 and 3), as described above.

2.2.2 Streaming

Memory accesses in commercial benchmarks display less temporal and spatial locality and cannot be accurately pre-

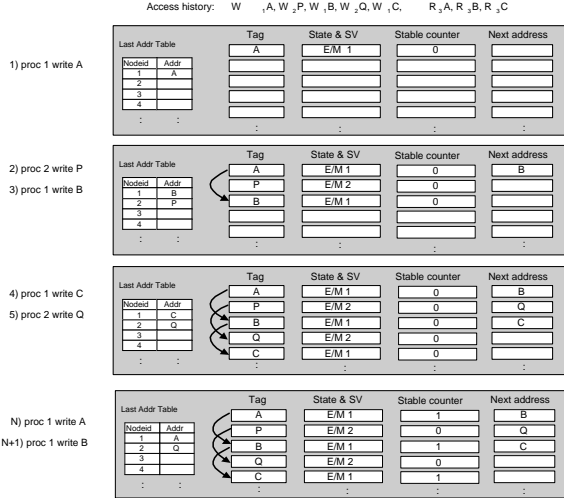


Figure 4. Streaming prediction process

dicted based on the access history of a single block. Recent research advocates fetching data in the form of streams, sequences of blocks that are accessed in a correlated fashion [17, 19, 20].

To detect streams, we add a Last Address Table (LAT) to each directory controller to track the address of the last write request from each processor and augment each directory cache entry with a *next address* field that points to the next element in the current stream. Figure 4 depicts the operations involved in detecting stable streams. In this example, processor P1 produces a stream {A, B, C} that will be consumed by processor P3. Meanwhile, P2 produces another stream P, Q. These two streams are interleaved, and arrive at the home node in the order: $W_1A, W_2P, W_1B, W_2Q, W_1C...$ When the home node receives the read exclusive request of address A from P1, it first checks its LAT. As there is no previous write request from P1, the directory controller updates P1’s LAT entry with address A (step 1). Then, the home node receives a second write request from P1, this time asking for address B. Since the directory controller knows that the address of the last write request from P1 is A, it sets the *next address* field in A’s directory cache entry to B and updates the LAT (step 3). After receiving two more write requests, W_2Q and W_1C , the directory controller has built two linked lists that record two independent streams, {A, B, C} and {P, Q} (step 5). However, the value of stable counter of all directory entries remains 0 to indicate that these streams have not repeated, and thus are not stable.

The value of the *stable counter* is updated when the same address is re-requested by a particular processor, e.g., when P1 re-requests address A (step N).

A stream is considered stable if all of its elements have saturated their stable counters. If a processor reads data belonging to a stable stream, we predict that it will request

Parameter	Value
Processor	4-issue, 48-entry active list, 2GHz
L1 I-cache	2-way, 32KB, 64B lines, 1-cycle lat.
L1 D-cache	2-way, 32KB, 32B lines, 2-cycle lat.
L2 cache	4-way, 2MB, 128B lines, 10-cycle lat.
DRAM	200 processor cycles latency
Network	100 processor cycles latency per hop

Table 1. System configuration.

other data in the stream in the near future. In the previous example, when P3 requests address A (the start of stable stream {A,B,C}), the home node adds P3 to the reader set of A, B, and C. It then sends request that the current owner (P1) self-downgrades and flushes any elements of the stable stream that it has cached. After the home node receives the written-back data, it pushes the data into P3’s RAC. In our current design, we limit the length of a stream to 4, as suggested by previous work [20]. An important area of future work is to study the latency/bandwidth tradeoff with streams of varying length.

3 Evaluation

3.1 Simulator Environment

We evaluate our update mechanisms on a wide selection of benchmarks (19 scientific applications and 3 commercial application) using two simulators, UVSIM and Simics. UVSIM is a cycle-accurate execution-driven simulator [22]. It models a hypothetical 16 node future-generation SGI Altix architecture that we are investigating along with researchers from SGI. Table 1 summarizes the major parameters of our simulated system.

Among the 19 scientific applications, Barnes, Ocean, FMM, Watersnq, Waterspa, Cholesky, FFT, Radio, Radix and Volrend are from the SPLASH-2 benchmark suite [21]; EM3D is a shared-memory implementation of the Split-C benchmark [7]; Unstructured is a computational fluid dynamics application; Gauss, ASP and TC were used by Kaxiras to evaluate wide sharing sharing [9]; Appbt, CG, LU, MG, are from the NAS Parallel Benchmark suite (NPBs) [6].

To study commercial workloads, we model the processor described in Table 1 with the Virtutech Simics full system execution-driven simulator [11] and GEMS timing infrastructure [13].

3.2 Results

Figure 5 presents the execution time speedup, and Figure 6 presents the network message reduction. For each application, we also show the results for a baseline write-invalidate (only) with no RAC, a write-invalidate (only) system with a 32KB RAC, a system with a 32KB RAC that employs our write update optimization, and a system with

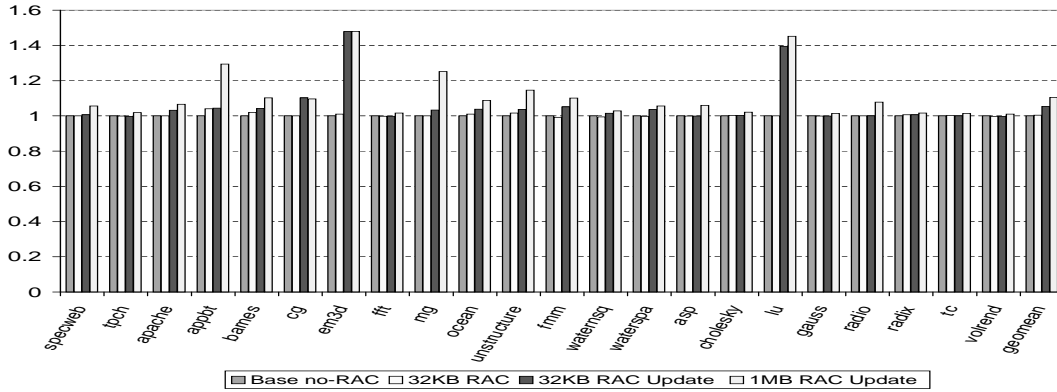


Figure 5. Application Speedup

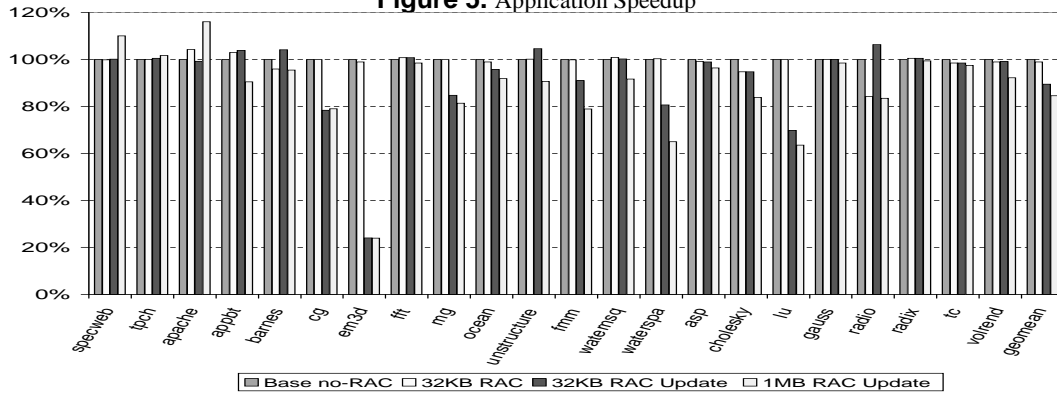


Figure 6. Network Traffic

a 1MB RAC that employs our write update optimization. All results are normalized to the baseline system’s performance. Other than the baseline and RAC-only systems, the results include both write-update performance policies: *stable reader set* and *stream*.

Rather than report the results of only the subset of applications that demonstrate large benefit, we report on all of the programs that we ran on our simulator. Our speculative update mechanisms never hurt performance, even in the few instances where they increased network traffic, and often led to significant performance benefits.

Our results corroborate previously published results [16] showing that scientific applications exhibit stable and highly repetitive sharing patterns. Some applications display one dominant signature like migratory (Cholesky, Water), producer-consumer (Barnes, Ocean, LU, CG, Appbt, Em3d, Mg) or wide sharing (Tc, Asp, Gauss). Others, e.g., Unstructured, exhibit different dominant signatures during different program phases. The *stable reader set* mechanism can detect these patterns and trigger updates when data is produced. Generally speaking, for applications with stable sharing patterns, the more often we detected opportunities to employ updates, the more remote misses we were able to eliminate and the better the performance speedup.

The percentage of data that can benefit from update optimization, and thus the required size of RACs to exploit our design, is application-dependent. In some applications, only boundary elements benefit from updates. Since the percentage of boundary elements are usually small relative to overall data size, a small RAC is able to achieve good performance for these applications. For example, for LU we only need a 32KB RAC to reduce remote misses by 22%, reduce network traffic by 30%, and improve performance by 39% when we employ updates. However, other applications, like Appbt, have a large number of blocks that can benefit from the updates. For these applications, a small RAC cache is able to capture only a fraction of the performance benefit of the large configuration. For Appbt, using a 32KB RAC limits the benefits of using updates to a 10% reduction in remote misses and 5% performance improvement. Increasing the RAC size to 1MB improves the performance benefit to 29% and eliminates an additional 26% of remote misses.

For commercial workloads, our proposed mechanisms reduce the number of remote misses by 5%-10% while increasing network traffic by 2% to 16%, which results in a performance improvement ranging from 2% to 6%.

4 Related Work

Write-update and hybrid update-invalidate protocols have been extensively studied [1, 15]. Producer-initiated communication and prefetching are two commonly used techniques to hide long miss latencies [2]. To guarantee *write atomicity* and *write serialization*, researchers often assume a weak/relaxed memory consistency model or rely on a bus interconnect to provide a total order.

Recent research has advocated decoupled coherence protocols. Token coherence [14] breaks a protocol into separate performance and correctness protocols. Coherence decoupling [8] breaks a protocol into speculative and correctness protocols. Similarly, we employ a write-invalidate correctness substrate and a speculative write-update performance substrate.

A different approach to reducing coherence overhead is to support speculative coherence operations, which can be classified into *directory value prediction* [16] and *coherence operation prediction* [9]. All of these techniques require changes to the processor core. In contrast, our simple, and likely less accurate, predictor can be implemented in an external directory controller with little hardware overhead.

Stream detection can enable sequences of blocks that are accessed in consistent orders to be streamed between nodes [19, 20, 17]. Streaming increases the accuracy of coherence prediction by correlating a recurring sequence of addresses. However, previously proposed stream detectors require significant storage overhead in the form of large history tables and/or processor modifications. To meet our design constraints, we propose a simple hardware mechanism to detect streams that uses only information available in the directory controller.

5 Conclusions and Future Work

There are many ways to extend and improve our work. We plan to investigate the value of more sophisticated predictors, like spatial streaming and temporal streaming. In addition, we plan to investigate the potential performance benefits of non-sequentially consistent versions of our mechanisms.

References

- [1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *International Symposium on High-Performance Computer Architecture*, 1997.
- [2] G. Byrd and M. Flynn. Producer-consumer communication in distributed shared memory multiprocessors. *Proceedings of the IEEE*, 1999.
- [3] L. Cheng, J. Carter, and D. Dai. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *International Symposium on High-Performance Computer Architecture*, 2007.
- [4] A. Cox and R. Fowler. Adaptive cache coherence for detecting migratory shared data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.
- [5] D. L. Dill. The Murphi verification system. In *Computer Aided Verification*, 1996.
- [6] D.H. Bailey, et al. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, Fall 1994.
- [7] B. Falsafi, A. Lebeck, S. Reinhardt, I. Schoinas, M. Hill, J. Larus, A. Rogers, and D. Wood. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing '94*, pages 380–389, Nov. 1994.
- [8] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: Making use of incoherence. In *ASPLOS*, 2004.
- [9] S. Kaxiras and J. R. Goodman. Improving CC-NUMA performance using instruction-based prediction. In *International Symposium on High-Performance Computer Architecture*, 1999.
- [10] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.
- [11] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.
- [12] M. Martin, P. Harper, D. Sorin, M. Hill, and D. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared memory multiprocessors. In *Annual International Symposium on Computer Architecture*, 2003.
- [13] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 2005.
- [14] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *Annual International Symposium on Computer Architecture*, 2003.
- [15] M. R. Marty and M. D. Hill. Coherence ordering for ring-based chip multiprocessors. 2006.
- [16] S. S. Mukherjee and M. D. Hill. Using prediction to accelerate coherence protocols. In *Annual International Symposium on Computer Architecture*, 1998.
- [17] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. *Annual International Symposium on Computer Architecture*, 2006.
- [18] P. Stenström, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.
- [19] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *Annual International Symposium on Computer Architecture*, 2005.
- [20] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, C. Gniady, A. Ailamaki, and B. Falsafi. Store-ordered streaming of shared memory. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA, 2005.
- [21] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.
- [22] L. Zhang. UVSIM reference manual. Technical Report UUCS-03-011, University of Utah, May 2003.

An M/G/1 Queue Model for Multiple Applications on Storage Area Networks

Emmanuel Arzuaga and David R. Kaeli
Northeastern University Computer Architecture Research
Electrical and Computer Engineering Department
Northeastern University
360 Huntington Avenue, Boston, MA 02115
{earzuaga,kaeli}@ece.neu.edu

Abstract

Storage Area Networks (SAN) have become one of the most widely used solution for high performance enterprise storage applications. However, the infrastructures of SANs are very complex and varied among different vendors. It is important to understand the behavior of these SANs in order of enhancing performance of the applications that run on them. The main idea of this paper is to understand the behavior of a server-SAN system that manages multiple applications stored in the same volume. We aim at modeling a typical workload of a small company which consists of one license of a database system software and shares that same license for multiple databases. These multiple databases may be used potentially for diverse applications. The objective is then to see how the system would behave when managing queries from different types of databases at the same time. In this paper we will use an M/G/1 Queueing Model to analyze the system and will report our first set of experiments which are composed of two different database workloads based on the TPC-C and TPC-H benchmarks. We show that the model can be used to anticipate the impact of growth in the arrival rate on the behavior of the system. This model has the potential of predicting the system behavior when adding more workloads to the mix.

1. Introduction

SANs typically consist of large disk arrays controlled by local storage processors (SPs) and made available to the server as different virtual partitions or logical unit numbers (LUN) through a fast attached media such as fibre channel. In these experiments we want to address the lack of characterization of concurrent workloads running on a server system. A considerable amount research has focused traditionally in characterizing the behavior of streaming data (video/audio) or servers dedicated completely for a single

type of application [1]. In real life this may not necessarily be the common case. Due to high cost of database software licensing, small companies often use same licenses to create multiple databases that they will need. For example, companies may have a Web Server that handles online transaction processing (OLTP) type transactions and also internally may need to process decision support systems (DSS) type queries. One possible implementation of their system could be thought as a set of workloads consisting of multiple applications interacting with a single database system. In order of enhancing performance for this type of services, there is the need to fully understand the characteristics of these multiple instance workloads running both single and concurrently.

Analytical Modeling aims at describing a collection of measured and calculated behavior of different computer system components such as workloads, hardware, software, cpu and the storage medium. The measurements are done in a finite period of time, thus we would like this time window to capture the behavior we are interested in. Analytical models can be built for different purposes. Some models are created as a basis for prediction of behavior of certain elements within a system. The impact of moving to faster or slower hardware, for example, can be measured by changing different parameters of each component into the model. Other models, as in our case, are mainly built to gain an understanding of the current activity on the system of interest. We can then measure performance and analyze the behavior of the workloads and the hardware within it.

In this paper we will use an M/G/1 Queueing Model to characterize our multiple application system. We can take the advantage that Queueing Theory can be applied to evaluate the efficiency of a computer system that handles requests and provides services. Figure 1 depicts such scheme. The implemented workloads are based on TPC-C (OLTP) and TPC-H (DSS) benchmarks. These benchmarks are specified by the Transaction Processing Council [2]. We used our model with combination of these two types

of workloads as well as each running separately. The rest of

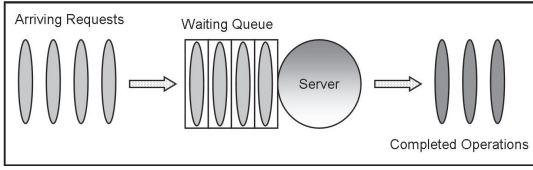


Figure 1. Example of a Queueing System

the paper is organized as follows. In section 2 we start with a description of the $M/G/1$ model. Section 3 will then discuss the characteristics of the implemented workloads. Results of the model simulation and a comparison with real data will be covered in section 4. Section 5 will analyze the results. Section 6 will cover some related work in the area. Then section 7 presents the conclusions and will propose future work.

2. $M/G/1$ Queueing Model

There has been a considerable amount of work analyzing the $M/G/1$ queue system [3, 4, 5, 6]. The $M/G/1$ queue has customers or transactions arriving according to a Poisson process with rate λ , but with service times having a general distribution. This provides us with a more general case when comparing it to the $M/M/1$ queue which has Poisson arrival rates and exponential service times. There is a drawback though: the $M/G/1$ queue does not have a general, closed form distribution for the number of transactions in the queue in steady state. However, it provides a generalization based on average values. That is, it gives a general solution for the average number of transactions in the queue, and the application of Little's Theorem provides us the corresponding result for the average time spent in the queue. Collectively, these results are known as the Pollaczek-Khinchin (P-K) formula:

$$T_W = \frac{\rho(1 + C_S^2)}{2(1 - \rho)} T_S \quad (1)$$

Where $\rho = \lambda T_S$ is the utilization of the system, defined in terms of arrival rate λ and T_S , the average service time. T_W is the average waiting time and C_S^2 is the coefficient of variation squared of T_S . The P-K formula for the $M/G/1$ queue assumes first come first served (FCFS) policy. Using the P-K formula, the average number of transactions in the system Q_L is:

$$Q_L = \rho + \rho^2 \frac{(1 + C_S^2)}{2(1 - \rho)} \quad (2)$$

Table 1. OLTP vs DSS Comparison

Characteristic	OLTP	DSS
Description	Transaction Processing	Informational Processing
Orientation	Transactions	Analysis
Function	Day to day operations	Decision Support, Long-Term informational requirements
Data	Up to date	Consistency maintained over time
Accesses	Read/Write	Read (mostly)
Operations	Index/hash on primary key	Large amount of scans
Performance metric	Transaction Throughput	Query Throughput, response time

and the average waiting line size Q_W is:

$$\begin{aligned} Q_W &= Q_L - \rho \\ &= \rho^2 \frac{(1 + C_S^2)}{2(1 - \rho)} \end{aligned} \quad (3)$$

With these equations, we can build a model that captures the behavior of a system composed of a server attached to a SAN which manages multiple applications in the same volume. We can use input parameters λ , T_S and C_S^2 to estimate system average queue length, utilization and average waiting time. We can then characterize the performance of the system based on these model output.

3. Workload Characteristics

We are using in our experiments two types of workloads; an OLTP and DSS system. These workloads were implemented based on TPC-C and TPC-H [2]. Table 1 shows a comparison of both types of workloads. TPC-C models a wholesale supplier managing orders. Order-entry provides a conceptual model for the benchmark with the underlying components being typical of any OLTP system. The workload consists of five transaction types. Table 2 shows the transaction types and their read/write characteristics. The transactions operate against a relational database composed of 9 tables. These are: Warehouse, District, Customer, History, Order, New-order, Order-Line, Stock, and Item. Transactions will make reads, writes, and rollbacks. The application uses primary and secondary key access. Our OLTP workload consists of warehouses each containing 10 terminals as the TPC-C specification states. The terminal interface consists of a Linux terminal that displays information about the current transaction being performed. The delivery transaction is executed as any other transaction so there is no deferred mode. The TPC-H workload is a complex

Table 2. OLTP Transactions

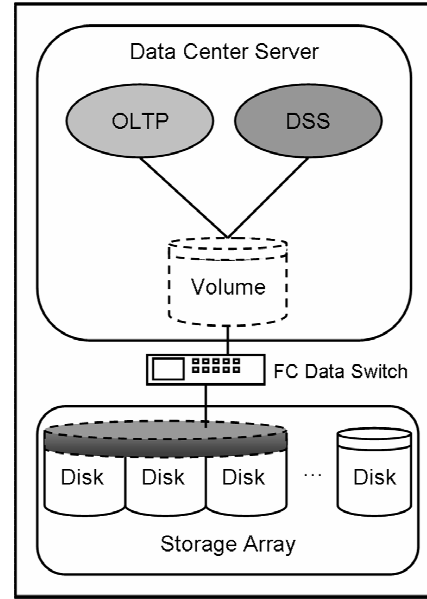
Name	Access Pattern
New-order	read/write
Payment	read/write
Delivery	read/write
Order-status	read
Stock-level	read

Decision Support workload. TPC-H Benchmark models ad hoc queries. The benchmark consists of queries that perform read accesses to the database and a pair of database update functions (add, delete). The database size is determined from fixed Scale Factors (SF): 1, 10, 30, 100, 300, 1000, 3000 which correspond to the nominal database size in GB. For example, SF 10 means a database with a size of approximately 10 GB. Tools for generating data and queries for the database are provided with the benchmark specification: DBGEN and QGEN. The use of these tools is strongly recommended and if not used, the mechanism for creating the data and queries must produce exactly the same output as those tools. TPC-H contains 22 well defined queries. Our DSS workload uses 17 out of the 22 queries. Sequential execution of these queries produce a query stream. The used queries are: q1-q16 and q19. Refresh functions perform database update between tests. The workload consists of three parts: Load Test, Power Test and Throughput Test. The Load Test just measures the time it takes for the system under test to build the database. The Power Test performs a pair of refresh functions and queries submitted in a single query stream. The test is performed sequentially (i.e., no concurrency). The Throughput Test performs multiple concurrent query streams and refresh functions.

4. Experiments

4.1. Single Class Model

The experiment consisted on running the workloads in a Data Center Server-SAN attached system using a same volume and DBMS. The implementation was done using a MYSQL 5.0 DBMS [7], C++ and MYSQL++ [8] tools. The experimental setup for this experiment depicted in Figure 2. The server is a DELL PowerEdge 1950 server with dual processor, dual-core, 2.0 GHz Intel Xeon EM64T processors with 8GB memory configuration connected to an EMC CLARiiON CX300 storage array via fibre channel with Red Hat Enterprise 4 (Linux 2.6.9-34.ELsmp, 64-bit kernel). The fibre channel connection is made through a using a McData Sphereon 1440 fibre channel switch. The server has a dedicated LUN of 1 TB consisting of 7 HDDs with a RAID 5 configuration. OLTP and DSS

**Figure 2. Experimental Setup****Table 3. OLTP workload description**

Warehouse Number	Number of Terminals
10	100

implementations have the configuration shown in Tables 3 and 4. The databases were previously created and loaded. We decided to keep the applications size small to capture transaction behavior without stressing the system, specially when running concurrently. In the case of the DSS system we ran a complete Power and Throughput test sequence. We collected disk I/O information of execution using linux command iostat [9] and EMC Navisphere CLI application [10]. These commands return information about the SAN treating it as a large disk. In this paper we will follow the same approach. We selected our model input data after the first 10 minutes of execution to allow time for the system to arrive at a near to steady-state condition. From this time interval, We will obtain the input parameters for our model such as average service time and arrival rate and compare the output of the model with the real results measured by

Table 4. DSS workload description

Scale Factor	Throughput Streams
1	10

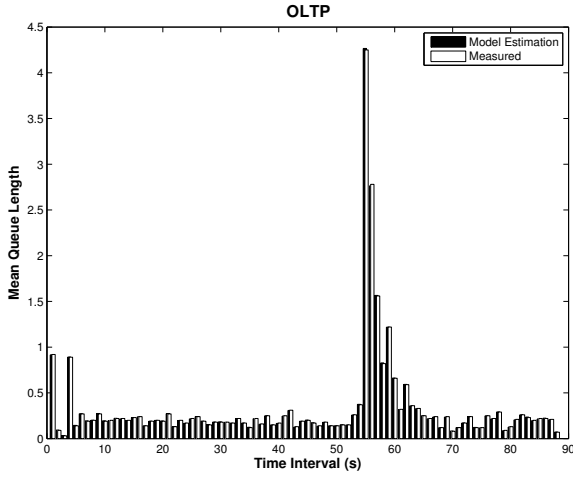


Figure 3. OLTP Queue Length Estimated by model vs Measured

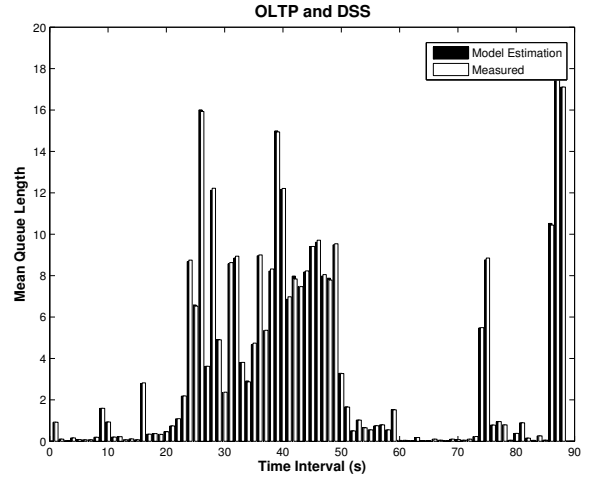


Figure 5. OLTP and DSS Queue Length Estimated by model vs Measured

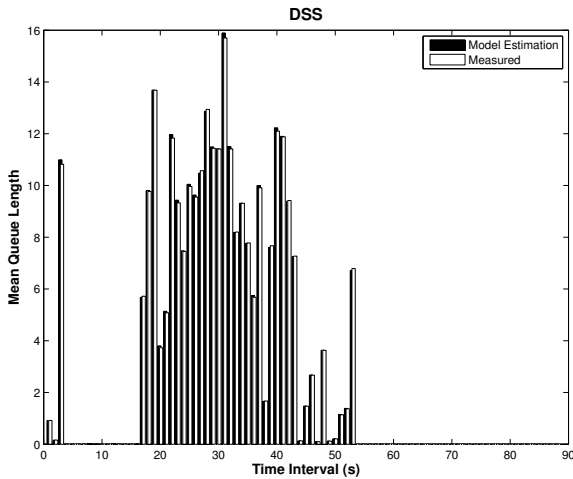


Figure 4. DSS Queue Length Estimated by model vs Measured

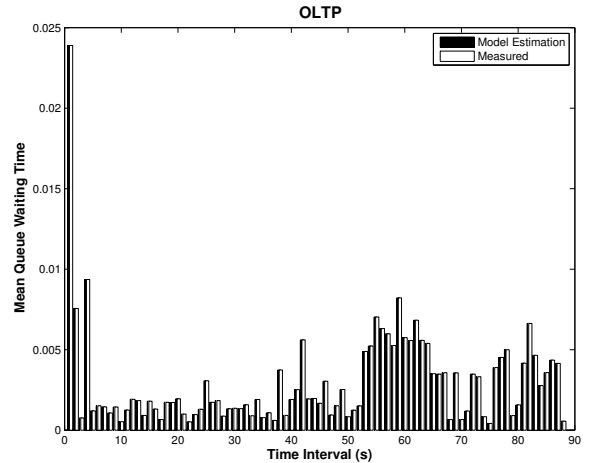


Figure 6. OLTP Queue Waiting Time Estimated vs Measured

these tools. The procedure is as follows:

1. Select a steady-state region in our simulation time (middle). We will collect the average service times, arrival rates, and response times.
2. Construct our model inputs, with these values and with the estimation of the coefficient of variance C_S^2 .
3. Estimate our model outputs, the average queue length Q_L and average waiting time T_W .

We will display the estimated values of Q_L and T_W

using our model and their respective measured results obtained during simulation.

Table 5 shows basic characteristics of the workloads. From these results, we can see that the multiple application (OLTP & DSS) is dominated mostly by the behavior of the DSS workload which has an I/O activity much larger than OLTP. This I/O activity is due to writes performed in the update functions. We can see from Figures 3 to 5 that the $M/G/1$ model predicts the average queue length with high accuracy. The error between measured results and the model estimates range from 0.01% to 4.26%. In the case of

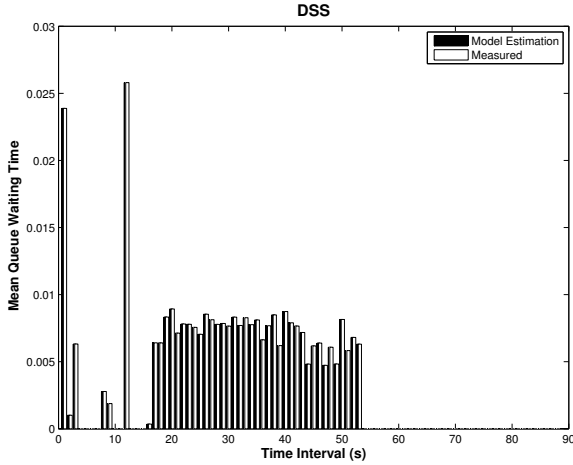


Figure 7. DSS Queue Waiting Time Estimated vs Measured

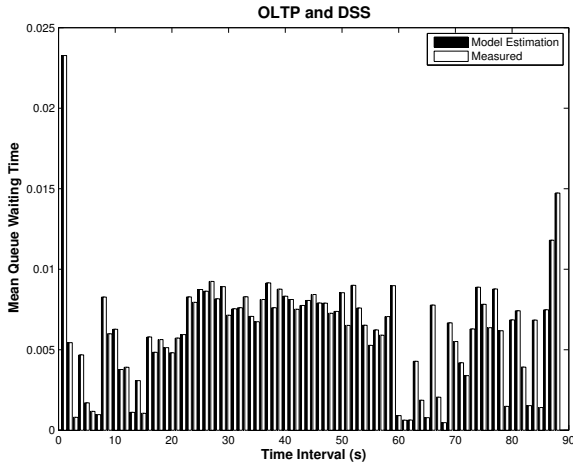


Figure 8. OLTP and DSS concurrent Queue Waiting Time Estimated vs Measured

the average waiting time, (Figures 6 to 8) the behavior is the same with error ranges from 0.03 to 16%. The 16% error is from DSS and is associated with the fact that few disk I/Os were generated with it. In the case of the concurrent the maximum error was 3.42% and OLTP error was 4.26%. The previous results give us confidence that the model has captured the behavior of the system. In the case of the multiple application results (Figures 5 and 8), we only have aggregated values at this moment, so we would have to model it with a single class model. In the next section we will analyze this model and how it behaves in terms of modifying these components to create a two class model.

Table 5. Workloads I/O Characteristics

Workload	R/W ratio	% reads	% writes	total I/Os
OLTP	0.461	0.316	0.684	4,254.2
DSS	0.001	0.001	0.999	35,392
OLTP & DSS	0.005	0.005	0.995	34,309

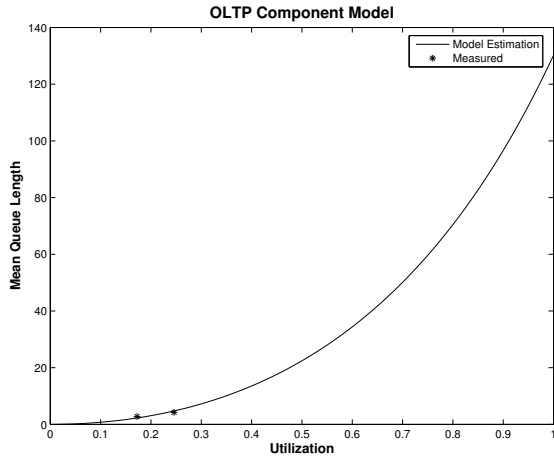
4.2. Two Class Model

The application of the previous models following a single class model approach produced excellent results. However, to fully understand the contribution of each workload, the multiple application experiment had to be modeled using a two class approach. Each input parameter representing the load of the classes were taken from the measured values (total). They were split in a manner such that the aggregation of both components resulted in the total measured values, that is, $\lambda = \lambda_{OLTP} + \lambda_{DSS}$, $\rho = \rho_{OLTP} + \rho_{DSS}$ and so on. We selected λ_{OLTP} to be around 20% the size of λ in order of satisfying the proportion we observed from Table 5. The service times ($T_{S_{OLTP}}$ and $T_{S_{DSS}}$) of this two class model are independent of each other. For simplicity we chose both to have the same average value, which was the measured one. We computed $C_{S_{OLTP}}^2$ and $C_{S_{DSS}}^2$. The output of our model will be the sum of the two components, that is: $Q_L = Q_{L_{OLTP}} + Q_{L_{DSS}}$. Since time values have the same average value, $T_{W_{OLTP}} = T_{W_{DSS}} = T_W$. The output of this model is the same as shown in figures 5 and 8 so we are confident that this approach can be used to analyze each component independently.

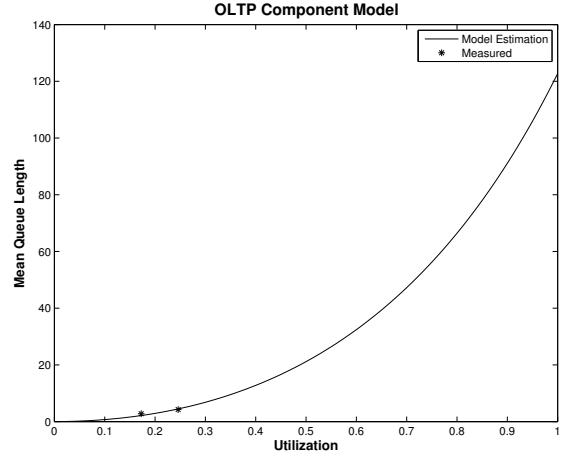
5. Analysis of Results

After validating our model, we can then proceed to analyze it. In this case we have selected a time interval inside the steady state region. The first step was to select similar measured values of T_S and their corresponding calculated $C_{S_{OLTP}}^2$ and $C_{S_{DSS}}^2$. Q_L values were calculated using the selected (T_S, C_S^2) pairs and varying ρ . ρ values were selected from 0-1 range so that we can see system behavior when modifying the load of each application component. The two class model was the only model used for this analysis. We split the model into the two components and draw the graphs of ρ vs Q_L . We also provide results from both components as a whole.

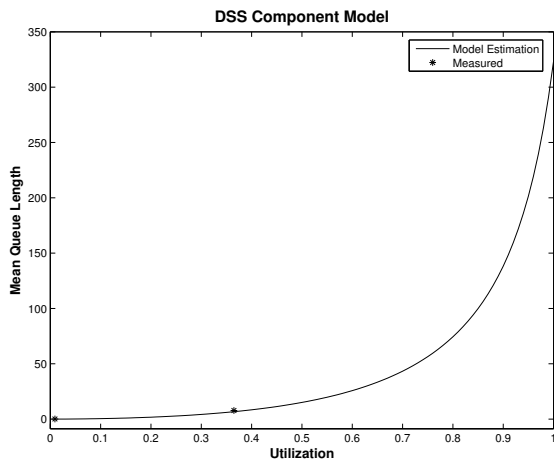
Figure 9 shows the obtained results. In (a) there is the OLTP component on the model. In this graph we let $\lambda_{OLTP} \gg \lambda_{DSS}$, specifically $\lambda_{OLTP} = 0.9 * \lambda$. The graph shows a saturation point at around $\rho = 0.6$, $Q_L = 30$. Looking at the measured data for the single OLTP execution, we found two values with the same T_S for this anal-



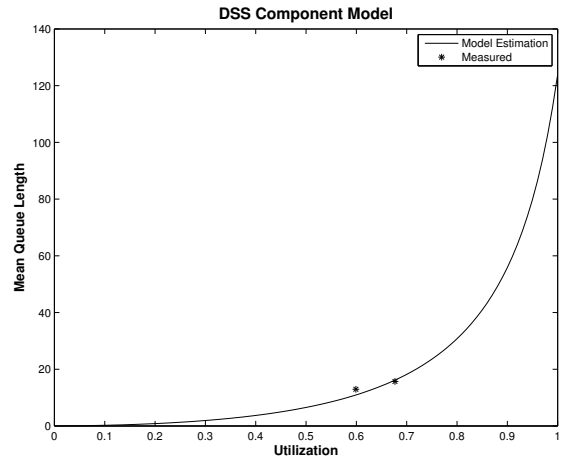
(a) $\lambda_{OLTP} \gg \lambda_{DSS}$



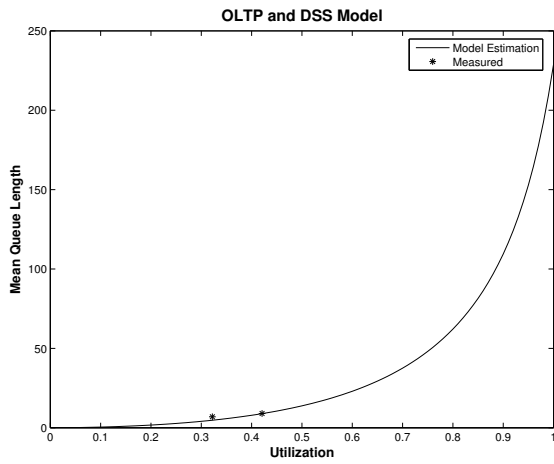
(a) $\lambda_{OLTP} \gg \lambda_{DSS}$



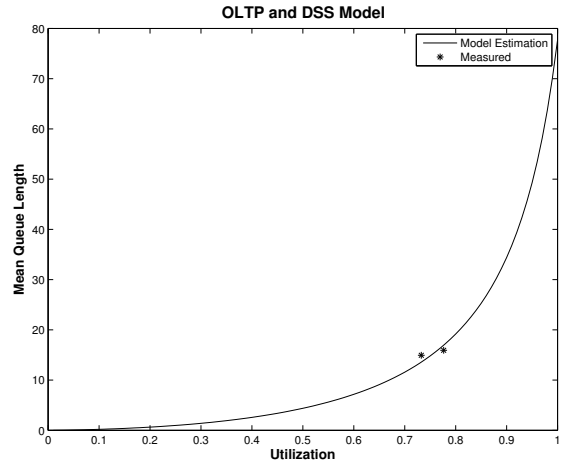
(b) $\lambda_{DSS} \gg \lambda_{OLTP}$



(b) $\lambda_{DSS} \gg \lambda_{OLTP}$



(c) Unmodified Model



(c) Unmodified Model

Figure 9. M/G/1 Two-Class Model: $T_S = 0.40$ ms

Figure 10. M/G/1 Two-Class Model: $T_S = 0.50$ ms

ysis. When the values for the measured ρ and Q_L are plot, they fall inside the model graph. Figure 9 (b) shows the same type of plot associated with the DSS component of the two class $M/G/1$ model. In this case, we let $\lambda_{DSS} \gg \lambda_{OLTP}$, with $\lambda_{DSS} = 0.8 * \lambda$. This component behaves very different than the OLTP component. The saturation point is located at around $\rho = 0.8$, $Q_L = 50$, which shows that this component saturates at a later utilization than that of the OLTP. Data values from the DSS single execution were also found and superimposed in the graph. We can see that the values from the DSS single experiments fit in the model's ρ vs Q_L curve. Finally, in (c) we have graphed both components of the model with values from the measured OLTP-DSS simulation. These values are also inside the model path as expected.

Figure 10 shows the same experiments but with different a value of T_S . The purpose of this figure is to analyze how the model behaves when the overall utilization is higher, specifically larger than 50%. We can see that also in this case, the model provides results that are close to the measured data. When analyzing both figures, we note that the combined two class model shows a behavior similar to that of DSS which is consistent with the measured results summarized in Table 5. This makes sense since DSS is the larger contributor in the mix. This analysis suggests that the two class model can be used to predict system behavior with certain similarity of actual measured data. In other words, the analysis shows that there is a contribution of each workload to the total mix and that these contributions can be separated while maintaining their respective behavioral attributes.

6. Related Work

The study of Storage Area Networks has gained popularity as this technology has emerged as one of the most commonly deployed storage solutions. Our work implements a queue model and uses it to model a particular system using obtained measurements. We are interested in modeling multiple complex database applications running concurrently.

Most of the previous efforts for SAN research have been in introducing theoretical models. Chao, Li-zhu and Chun-xiao [11] proposed the use of an $M/M/Q$ model for Performance simulation of SANs. The authors construct a software simulator and show how this model is general enough to be fitted to different SAN systems. Their work shows an error of around 10%. Zhu, Zhu and Xiong [12], presented a Queueing Network Model to analyze SAN structures. They create a theoretical model that details all mayor components of a SAN as a network of queues. Besides the theoretical model, the authors provide analysis for the host server, fibre channel network, disk array subsystem, fibre channel protocol and disk units. Molero et al [13] developed a tool

for the design and evaluation of fibre channel SANs. This tool is based on the CSIM C/C++ [14] extension to create simulations.

Menasc, Pentakalos and Yesha [15] present an analytical Queueing Network Model for mass storage systems. They consider both host attached and network attached devices. They provide a modified Mean Value Analysis (MVA) algorithm that can handle multiple classes of requests for a striped RAID array. Their workloads were a set of ftp get and put for a ten day period. Nijim, Xie and Qin [16] propose the use of an analytical model to model the performance of a self-managing computer systems under mixed workloads conditions. They perform a trace driven simulation where their workload mix consisted of one cpu intensive and one I/O intensive. Uргаonkar et al [17] present an analytical Queueing Model for multi-tier Internet applications. Their model predicted average response times within the 95% confidence intervals of the observed average response times.

7. Conclusions

In this paper we have shown results from modeling a single server connected to a SAN. We used a single $M/G/1$ Queueing Model. The experiments deal with characterizing the behavior of multiple workloads running in the same system. The results show how the model manages to represent the system of interest. This model is able of generating output parameters that yield low error when compared to experimental data. We learned that the two class $M/G/1$ model is able of predicting system behavior that is actually similar to measured data. The model can be used to anticipate the impact of growth in the arrival rate on the behavior of the system. In the case of adding more workloads to the mix, the model has the potential of predicting the system behavior once the new mix has been characterized.

As future work, we plan to extend these experiments by both increasing the workload number and size. We are also interested on quantifying the impact of each instance independently as a component of the whole disk utilization. This will require us to extend the model to a multi (more than two) class model. This would help us understand better the overall performance of the system given a particular set of desired applications to be run concurrently. The use of Queueing Network Models to characterize the complete system in terms of its components components as the fibre-channel switch, the storage processors and host bus adapters is the long term goal. This approach will provide an even greater level of understanding of all the characteristics this server-SAN system has.

8. Acknowledgments

This work was supported in part by an NSF Major Research Instrumentation Grant (Award Number MRI-0619616), the Institute for Complex Scientific Software, and from the Gordon-CenSSIS, the Bernard M. Gordon Center for Subsurface Sensing and Imaging Systems, under the Engineering Research Centers Program of the National Science Foundation (Award Number EEC-9986821).

References

- [1] L. Huang, G. Peng and T.C. Chiueh. Multi-dimensional storage virtualization. In *SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 14–24, ACM Press, New York, NY, USA, 2004.
- [2] Transaction Processing Council, TPC-C and TPC-H benchmarks specification (URL), <http://www.tpc.org>; accessed August 12, 2007.
- [3] L. Kleinrock. *Queueing Systems. Volume I: Theory*. Wiley, New York, 1975.
- [4] D. Bertsekas and R. Gallager. *Data Networks*, second edition. Prentice Hall, New Jersey, 1992.
- [5] G. Bolch, S. Greiner, H. deMeer and K.S. Trivedi. *Queueing Networks and Markov Chains*. Wiley, New York, 1998.
- [6] E.D. Lazowska, J. Zahorjan, G.S. Graham and K.C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice Hall, New Jersey, 1984.
- [7] MYSQL AB, *MYSQL Database* (URL), <http://www.mysql.com>; accessed August 12, 2007.
- [8] TangentSoft.net, *MYSQL++ library* (URL), <http://tangentsoft.net/mysql++/>; accessed August 12, 2007.
- [9] S. Godard, *Iostat man file, systat utilities home page* (URL), http://perso.wanadoo.fr/sebastien_godard/; accessed November 23, 2007.
- [10] EMC² Corporation, *Navisphere SAN Management Suite* (URL), http://www.emc.com/products/storage_management/navisphere.jsp; accessed July 15, 2007.
- [11] L. Chao, Z. Li-zhu, X. Chun-xiao. Using MMQ Model for Performance Simulation of Storage Area Network. In *ICDE 2005: Proceedings of the 21st International Conference on Data Engineering*, IEEE Computer Society, Washington, DC, USA, 2005.
- [12] Y. Zhu, S. Zhu and H. Xiong. Performance Analysis and Testing of the Storage Area Network. In *MSST 2002: 19th IEEE Symposium on Mass Storage Systems and Technologies*, IEEE, Maryland, USA, 2002.
- [13] X. Molero, F. Silla, V. Santoja and J. Duato. A Tool for the Design and Evaluation of Fibre Channel Storage Area Networks. In *SS 2001: Proceedings of the 34th Annual Simulation Symposium*, IEEE Computer Society, Washington, DC, USA, 2001.
- [14] Lockheed Martin Advanced Technologies Laboratory, *The CSIM Simulator* (URL), <http://www.csim.com>; accessed November 18, 2007.
- [15] D.A. Menasc, O.I. Pentakalos and Y. Yesha, An Analytic Model of Hierarchical Mass Storage Systems with Network-Attached Storage Devices, In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ACM Press, Philadelphia, May 1996.
- [16] M. Nijim, T. Xie and X. Qin, Integrating a Performance Model in Self-Managing Computer Systems under Mixed Workload Conditions, In *Proceedings of the IEEE International Conference on Information Reuse and Integration*, IEEE, August, 2005.
- [17] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer and A. Tantawi, Analytic modeling of multitier Internet applications, In *ACM Transactions on the Web (TWEB)*, Volume 1, Issue 1, ACM Press, New York, May, 2007.

Varying Memory Size with TPC-C --- Performance and Resource Effects

Jay Veazey and Blaine Gaither
Hewlett-Packard

Jay.Veazey@hp.com and Blaine.Gaither@hp.com

Abstract

Using TPC-C™ as an evaluation tool, we examine the effects of varying memory size on throughput, I/O rate, bus utilization and cache utilization. Unexpected relationships between memory size and resource utilizations are revealed and quantified. The platform studied was an HP Integrity rx6600 with two dual-core Itanium 2 CPUs, running at 1.6 GHz with 24 Mbytes cache memory per socket.

1. Introduction and Background

It's general knowledge that increasing memory size in a server can improve performance. However, it's a long way from this "general knowledge" to quantifying the gain in performance, much less to an understanding of the underlying cause-and-effect relationships between memory size and throughput.

The question of how much memory to design into a server is an important consideration for R&D teams, involving as it does scheduling, parts cost, board layout, thermal and power tradeoffs, as well as performance.

In this paper, we will quantify this gain in performance and discuss the changes in resource utilizations associated with varying memory size. The workload used is TPC-C [1], the DBMS is SQLServer, and the server is an HP Integrity rx6600 with 2 CPUs.

TPC-C, for those unfamiliar with it, is a commercial benchmark developed by the Transaction Processing Council. It represents an on-line transaction procession (OLTP) environment. As the Transaction Processing Council website (<http://www.tpc.org/tpcc/detail.asp>) describes it:

"As an OLTP system benchmark, TPC-C simulates a complete environment where a population of terminal operators executes transactions against a database. The benchmark is centered around the principal activities (transactions) of an order-entry environment. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses.

However, it should be stressed that it is not the intent of TPC-C to specify how to best implement an Order-Entry system. While the benchmark portrays the activity of a wholesale supplier, TPC-C is not limited to the activity of any particular business segment, but, rather, represents any industry that must manage, sell, or distribute a product or service." [2]

When properly implemented, TPC-C will heavily stress system resources. For example, CPU utilization usually approaches 100%, and the I/O workload is similarly intense---even a small server may require hundreds of disk spindles. For example, an rx6600 with 4 CPUs---Itanium 9050s---requires 766 disk spindles, as detailed in http://www.tpc.org/results/individual_results/HP/hp_sq_lsapphire_win64_exsum.pdf

TPC-C rules require that the initial size of the database be scaled in proportion to throughput---that is, the faster the system, the larger the database required. For example, the system cited above has an initial database size of 2,433 GBytes, with a throughput of 344,928 tpmC, while the system described in www.tpc.org/results/FDR/TPCC/hp_sqldiablo_win64_060327_fdr.pdf achieves a throughput of 290,644 tpmC and has an initial database size of 2,036 GBytes. However, this system also had 766 disk drives configured, despite the smaller database. (The links above include "Fair Use" data on price-performance and availability.)

The capacity of the I/O subsystem, in GBytes, is usually much larger than the TPC-C rules require. Extra disk spindles are needed to achieve maximum throughput while satisfying the response time requirements. In the case of the system detailed in http://www.tpc.org/results/individual_results/HP/hp_sq_lsapphire_win64_exsum.pdf, 8,965 GBytes are required (including capacity for 60 days of database growth), while the disk array contains 25.642 GBytes of formatted space.

The exact ratio of total disk space to space used is a matter of engineering judgment.

The system used in these experiments was an rx6600 with 2 Itanium-2 9050 CPUs. The I/O system contained approximately 750 disk drives.

2. Data and Analysis

The memory configuration was varied in these experiments from 32 GBytes to 192 GBytes, the latter being the maximum system capacity. These were lab experiments and all results are unofficial. The goal here was to determine the effect of memory size on:

- *Throughput*; in particular, to find the memory configuration that yielded maximum performance.
- *Resource utilization*, including I/O rates, cache miss rates, and memory bandwidth.
- *OS behavior*, particularly the effect on OS scheduling and context switching.

Figure 1 shows the effect of adding memory on throughput.

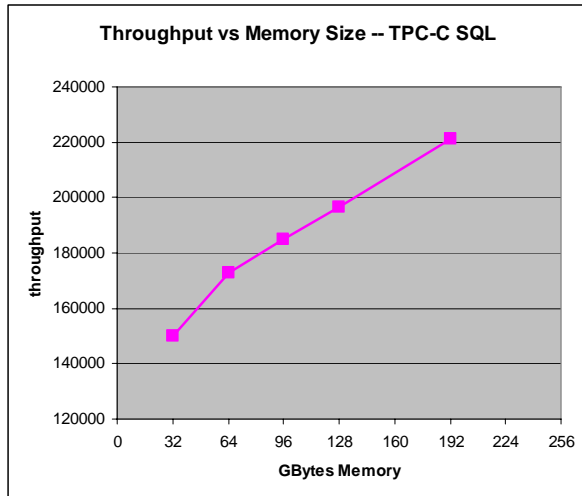


Figure 1. Throughput vs. memory

While it may seem intuitive that increasing memory would increase performance, exactly what resource or resources are conserved by additional memory? What precisely causes this improvement?

This is a question of practical importance---in designing new servers, we are frequently confronted with choices between design strategies. To take a single example, should we optimize a given system for memory latency, or for memory capacity? In other words, are faster memory accesses more valuable than greater memory capacity? The “more valuable” qualifier is critical---everyone in the industry seems to

have an opinion, and these opinions are often valuable so far as they go---but they are seldom *quantified* beyond the precision of a “rule of thumb,” and without quantification we cannot make economically rational design tradeoffs.

In this case, one underlying mechanism is that increasing memory reduces the number of physical I/Os. Physical I/Os do not cause the CPU to fall idle, unless the benchmarking team has set up too few concurrent threads or too few disk drives, but they do require a significant number of instructions to execute. On this platform, each I/O requires about 18K instructions. The exact relationship between I/O rate and memory size is detailed in [3].

Table 1 details CPU utilization and the effect of decreasing I/O on pathlength.

Table 1. Disk I/O and CPU utilization

GB Mem	throughput	CPU Util.	I/Os / sec	Relative thrupt	approx. % insts. devoted to I/O
32	149,934	99.7%	71,068	1.00	31%
64	173,017	99.0%	58,907	1.15	24%
96	184,716	99.7%	50,574	1.23	20%
128	196,521	99.5%	44,397	1.31	17%
192	221,289	99.9%	29,422	1.48	11%

The reader will note that a decrease of instructions devoted to I/O---the rightmost column---is equivalent to a decrease in total pathlength of 20%, as we proceed from 32 GBytes to 192 GBytes. But relative performance increases by 48% over the same interval. We conclude that reduced I/O pathlength accounts for about 42% of the performance benefit.

What factors account for the other 58%? In general terms, a CPU-intensive application will run faster in two cases: when the instructions per transaction is reduced, or when there’s an increase in the instructions per second executed.

Since CPU utilization is already 100%, instructions executed per second can be increased only by reducing CPI (cycles per second). Leaving aside for the moment the issue of how CPI is reduced, Figure 2 shows the effect of adding memory on CPI.

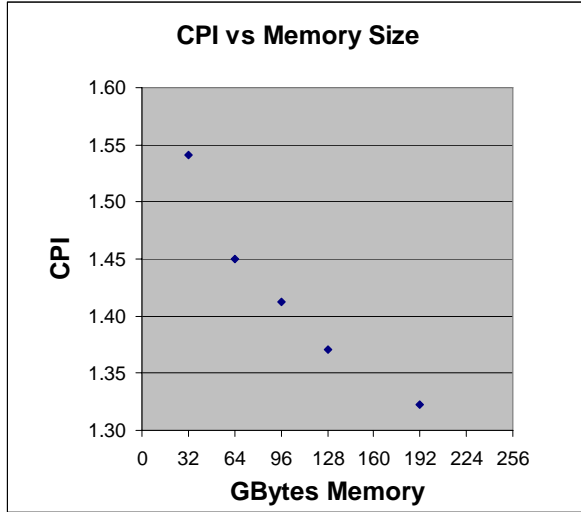


Figure 2. CPI vs. memory

CPI is sensitive to several factors, including CPU micro-architecture, compiler optimizations and cache design, but most of these are constant for the data in this study. One factor that isn't constant is memory latency. Whenever a load instruction stalls awaiting a memory access, overall CPI increases.

Figure 3 shows that memory bandwidth decreases as memory is added, by about 15% from 32 GBytes to 192 GBytes, despite a 48% increase in performance. Some of this decrease in bus bandwidth is due to less I/O. Disk I/O requires DMA, and in a system that uses a snoopy cache coherency scheme, the memory controller queries the caches to ensure that memory and caches have the same copy of the data.

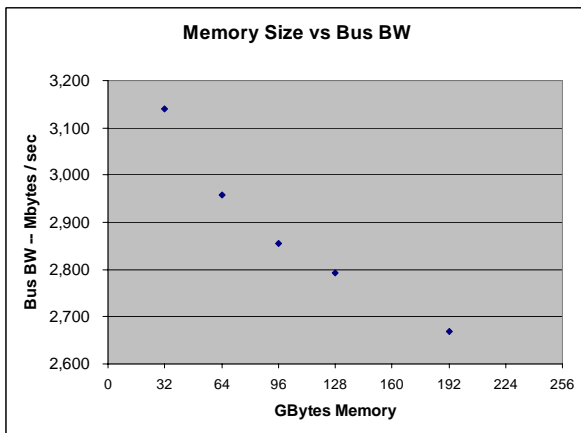


Figure 3. Bus bandwidth as a function of memory size

However, less than 20% of the reduction in bus bandwidth is due to a decrease in DMA-related coherency traffic. Most of the effect is due to a reduction in cache activity---the caches stabilize as memory is added. Table 2 details cache activity. Note that cache accesses are normalized per CPU per second per tpmC. The "per tpmC" normalization is particularly important, allowing us to focus on cache activity per unit of throughput.

Table 2. Cache activity as function of memory

memory	L1 accesses	L1 misses	L2 misses	L3 misses
32	6901	1549	183	22
64	6219	1377	155	19
96	5943	1297	139	17
128	5683	1232	127	16
192	5122	1095	109	14

Units in Table 2 are accesses (or misses) / sec / CPU / tpmC.

(The Itanium 2 9050 has an L1 cache that is split between instruction and data; likewise for the L2. The unified L3 cache is 12 Mbytes. The L1 cache is closest to the CPU, while the L3 interfaces with the front-side bus.)

Note that the L1 access rate falls by 26% as memory size is increased to 192 GBytes. This implies that the register set is stabilized by the addition of memory.

The most obvious explanation for the stabilization of the cache hierarchy and register set is that OS thread switches are reduced with additional memory, as shown in Figure 4.

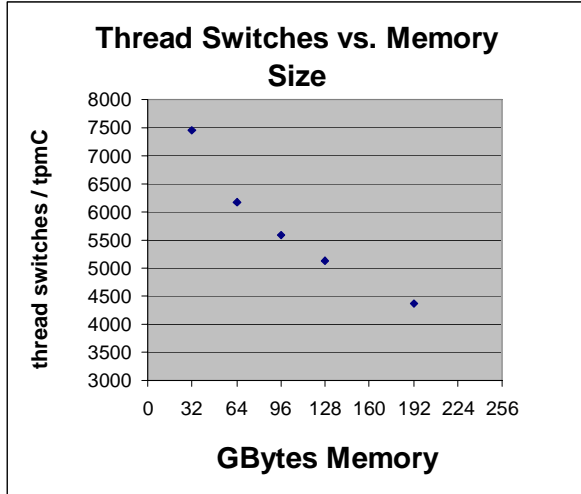


Figure 4. Thread Switches vs. Memory

Operating systems make context switches (either of OS threads or processes---whatever the unit of multi-programming may be), either because the current thread can no longer execute or as a result of fairness scheduling policies, such as time-slicing.

For TPC-C, time-slicing is unimportant. A thread will need to do I/O long before it would be time-sliced out.

Thread switches are positively correlated with L1 accesses (correlation coefficient = .996). While correlation does not necessarily prove causation, it seems highly probable in this case that the decline in

thread switches causes the register set (and with it, the cache hierarchy), to stabilize.

3. Conclusion

The addition of memory to a system running TPC-C increases performance significantly. More memory results in less I/O, as more data and code are cached in memory. Less I/O implies a shorter instruction pathlength, as well as fewer context switches. The decline in context switches stabilizes the cache hierarchy, resulting in fewer memory accesses and lower CPI.

4. Acknowledgment

Many thanks to Gunter Zink and team for their invaluable help in generating and reviewing this data.

5. References

- [1] "TPC Benchmark C," Standard Specification Revision 5.9, Transaction Processing Council, June 2007, http://www.tpc.org/tpcc/spec/tpcc_current.pdf
- [2] F. Raab, W. Kohler, A. Shah, "Overview of the TPC Benchmark C: The Order-Entry Benchmark," <http://www.tpc.org/tpcc/detail.asp>
- [3] J. Veazey, B. Gaither, "An I/O-Memory Model for TPC-C (SQLServer)," *Journal of Computer Resource Management (CMG)*, Issue 112, 2004.

An Analysis of the Sensitivity of SPECjAppServer2004¹ to Memory Sub-System

Rajeev Garg, Kumar Shiv, Andrew Sun, Mahesh Bhat, Michael LQ Jones
Intel Corporation

Abstract

The performance of application servers and other such memory-bound applications is limited by the memory sub-system, which is improving at a much slower rate than the processor sub-system. Until recently, most research was in the area of instruction-level parallelism and methods to extract more performance from the processor. To truly enjoy the benefits of processor architectural improvements, an application and system level understanding is required. Specifically, insight into the sensitivity to memory sub-system can lead to better decisions and trade-offs in system and processor design. In this paper, we present a methodology to gain a deeper understanding of the sensitivity to the performance of the memory sub-system. We then use this trace-simulation based method to understand the implications of memory sub-system performance to the leading industry-standard application server benchmark, SPECjAppServer2004¹. We define and measure stall-factor, the portion of memory latency that stalls the processor across several dimensions including cache size, cache sharing, memory latency, code read miss rate, data read miss rate, and data write miss rate.

1. Introduction

Various trends in computer architecture have resulted in a growing imbalance between the performance of the processor sub-system and the performance of the memory sub-system. This has been exacerbated lately by the push to the multi-core architecture as a way to utilize the ever increasing number of transistors available; thanks to Moore's law. The larger number of transistors has also resulted in larger caches as a way to alleviate the pressure on the memory, but the effect of the memory sub-system on the system performance continues to worsen. Because of this, architects and designers have been paying a lot

of attention on ways to mitigate this. Their job is made more difficult by the fact that memory affects the performance and cost of the system in different ways for different applications. Thus, having a good understanding of the effect of the memory sub-system on a large suite of applications is vital for optimal design of future systems.

In this paper, we study the industry-standard Enterprise Java benchmark SPECjAppServer2004. In particular, we characterize the benchmark performance on current multi-core systems and then investigate the effects of various components of memory traffic. For example, we study the impact on processor CPI (cycles per instruction) of code misses in the last-level cache (LLC).

SPECjAppServer2004 is a good proxy for a large class of enterprise middleware applications. It is a 3-tier workload. Figure 1 shows a representative configuration. It is an end-to-end application, which exercises all major J2EE technologies implemented by compliant application servers including the web container, servlets and JSPs, the EJB container, EJB2.0 Container Managed Persistence, JMS and Message Driven Beans, transaction management, and database connectivity. It heavily exercises all parts of the underlying infrastructure that makes up the application environment, including hardware, JVM software, database software, JDBC drivers, and the system network. The benchmark metric is JOPS, Java operations per second.

The application has a large total code footprint of several hundred megabytes, and is made up of approximately 10,000 Java methods. The methods are very small, and call chains are very deep. The usage profile is flat with no method consuming more than 2% of application time. However, in spite of this, the footprint of frequently used code is not very big. The data footprint for a Java workload is artificially imposed by the prescribed heap size, and is about 2GB. The workload has significant network traffic between the 3 tiers.

¹ SPECjAppServer is a registered trademark of the Standard Performance Evaluation Corporation (SPEC)

In the next section, we describe our methodology. In the third section, we present measurement data on current systems to provide us with a baseline. Section 4 presents the results of simulations using traces. Section 5 provides a discussion of essential next-steps, and in the final section we provide our concluding thoughts.

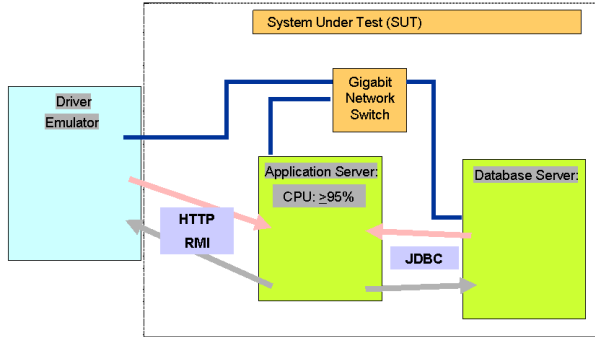


Figure 1: Typical SPECjAppServer2004 configuration

2. Methodology

We first use measurement data on current systems to provide us a firm foundation for our work. We then use a well-tested and validated suite of traces with cycle accurate simulators to carry out a wide sweep of simulations. On the general observation that the effect of the memory sub-system is unlikely to be uniform and constant for all types of memory traffic, we then attempt to deduce the individual impact each memory transaction type has on the system performance. Parameters of interest include LLC size, effective memory latency, the number of processors (hardware threads) sharing a cache, the code miss rate, the data read miss rate, and the data write miss rate. We do this by holding several of the parameters fixed and varying the others in a controlled way in a suite of simulations. The parameter values that are fixed are set to appropriate values derived from our baseline data. In each case, we compute a stall factor, which is the portion of the relevant memory latency that actually affects the application CPI. This factor is between 0 and 1, and a higher value would indicate a greater impact on the CPI of each memory request of that type.

3. Measurement Data

Figure 2 presents measurement data from a 2-socket Core 2 Duo based system. The pie chart shows the components of cache misses for a platform with 4M cache per socket. It can be seen that cache misses respond well to cache size increase, but that there are

substantial cache misses due to object creation (compulsory) and coherence (increase with cache size). The ~5% increase in pathlength in 4M case is within run-to-run variation limit.

System Level Metrics	2S4C-1M	2S4C-2M	2S4C-4M
Relative Performance (JOPS)	1	1.24	1.46
Total CPU Utilization	94%	84%	79%
Interrupts/JOP	26	21	20
CtxRate/JOP	52	55	53
Allocation Rate (KB/JOP)	737	757	798
Performance Metrics	2S4C-1M	2S4C-2M	2S4C-4M
CPI	4.4	3.17	2.89
Pathlength	6557163	6533215	6808556
MPI	0.0148	0.0088	0.0056
HIT	0.0133	0.0051	0.0030
HITM	0.0008	0.0010	0.0010
LLC Read Miss Latency (core clocks)	404	350	327

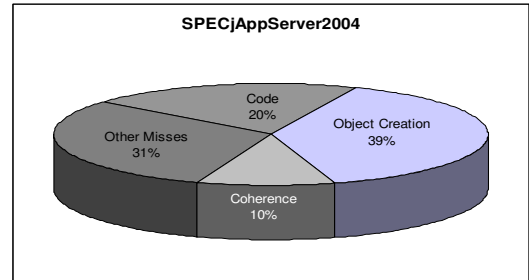


Figure 2: SPECjAppServer2004 Baseline Data – Core 2 Duo Systems

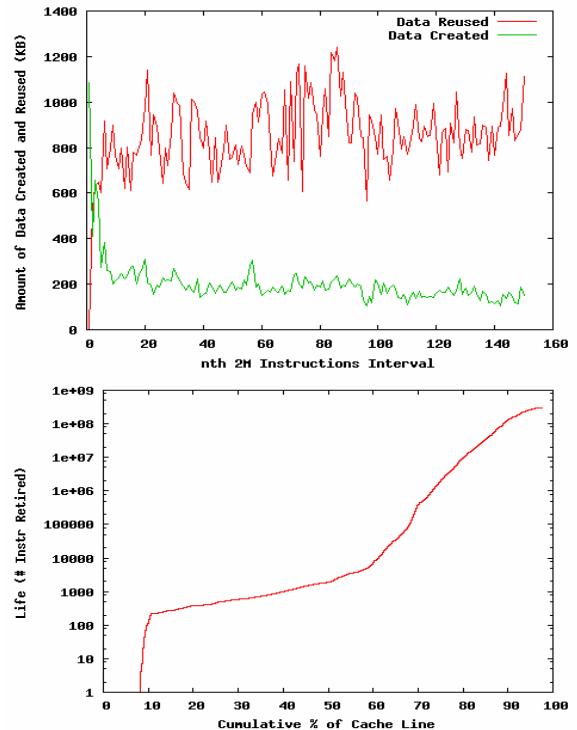


Figure 3: Data Usage Rate and Object Lifetime Profiles

Figure 3 profiles the data usage rate and lifetimes. “Reused Data” is the data accessed in a 2M instruction interval that is also been accessed in all the previous

intervals. “Created Data” is the data accessed in a 2M instruction interval that is never been accessed before. As evident from the upper graph of Figure 3, the application creates data at a constant rate. Also, the amount of reused data remains between 800 KB – 1000KB. This shows that the created data does not have long life; otherwise, the amount of reused data would have increased with the number of instruction retired. The lower graph shows that 60% of objects/data are short-lived. Figure 4 shows that while total footprint is large, most accesses are to a small footprint.

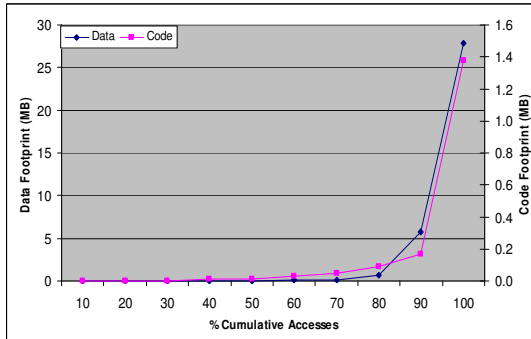


Figure 4: Code and Data Footprint Usage

4. Trace-based Simulations

Code read, data read, and data write misses have different impact on the application performance. Code and data read misses can be expected to stall the pipeline more aggressively (stall factor is higher) as compared to data write misses. This is especially true for server applications, which write significant amount data that is never read again, and, therefore, have no dependencies. For such applications, hardware can be optimized in a number of ways – for example, streaming writes, allocating cache line as least recently used, and optimizing for memory reads - to exploit the differences. Therefore, it becomes essential to study the effects of each type of cache miss. In this section, we discuss the performance impact of each miss type on SPECjAppServer2004 performance. All the results presented are based on the trace-driven cycle accurate simulation model.

4.1 Code Miss Sensitivity

Even though SPECjAppServer2004 has a very small code footprint (90% of accesses are to less than 256KB of code), 30 software threads per hardware thread and a constant rate of newly created objects drives out the code lines from the cache. This is especially true at cache sizes less than 4MB. For larger

caches, the older objects become least recently used and are, instead, replaced to bring in the new objects, but even here 20% of all cache misses are code misses.

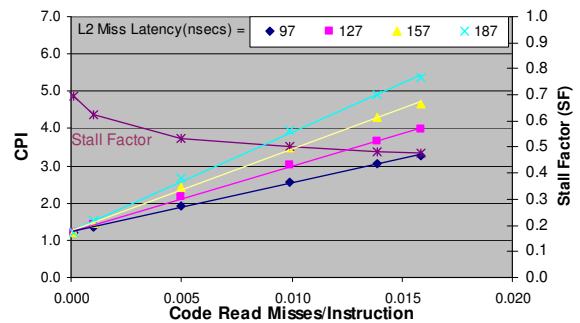
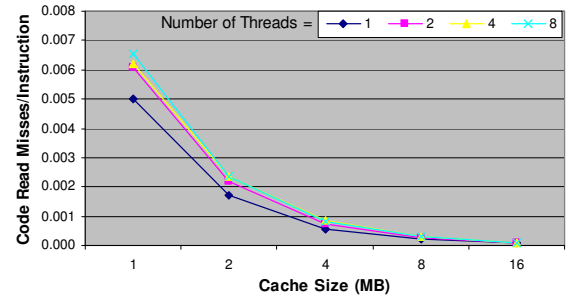


Figure 5: Code Read Miss Impact

Multiple threads sharing a large cache perform better as compared to smaller private caches per thread because all threads share the same code. The stall factor for code misses is very high (0.5 – 0.7), which indicates high performance sensitivity to the code misses. This is not surprising given the way code is laid out for this application by JVMs. The total code footprint is many times larger than the hot-code footprint. Code misses thus tend to be scattered around the address space limiting the advantage of hardware prefetchers. Very little of the penalty can therefore be hidden by prefetching. Additionally, since Java methods used in this benchmark tend to be very small, there are fewer code-misses simultaneously outstanding to memory, which again inflates the stall factor. This suggests that systems with smaller caches should optimize the code read data path. Code inlining by the compiler will help, and hints provided by the hardware to allow profile guided optimizations will be desirable. The stall factor decreases with the increase in the number of misses per instruction because of memory level parallelism (MLP), which reduces the average latency in the presence of multiple outstanding miss requests. Even though the stall factor decreases with the increase in miss rate, the CPI increases. This is due to the fact that the decrease in stall factor is overshadowed by the increase in miss rate.

4.2 Data Read Miss Sensitivity

As evident from the high stall factor (0.7 – 0.9) in figure 6, SPECjAppServer2004 performance is highly sensitive to the data read misses per instructions (DRMPI). At lower DRMPI, the stall factor is around 0.9, which indicates that 90% of the memory access time will be exposed as pipeline stalls. Due to a highly random access pattern, traditional stride-based prefetchers are not effective in hiding the miss latencies. Therefore, it becomes imperative to design more intelligent prefetchers and/or optimize the data read path. Misses drops sharply with the cache size and reaches an asymptote, indicating a certain fraction of compulsory misses. In addition to this, a shared cache performs much better than a private cache - 8 threads sharing a 16M cache have a quarter of the misses as compared to a 1 thread having its own 2M cache. This shows significant sharing of data among software threads. Again, due to MLP, stall factor decreases with the increase in miss rates.

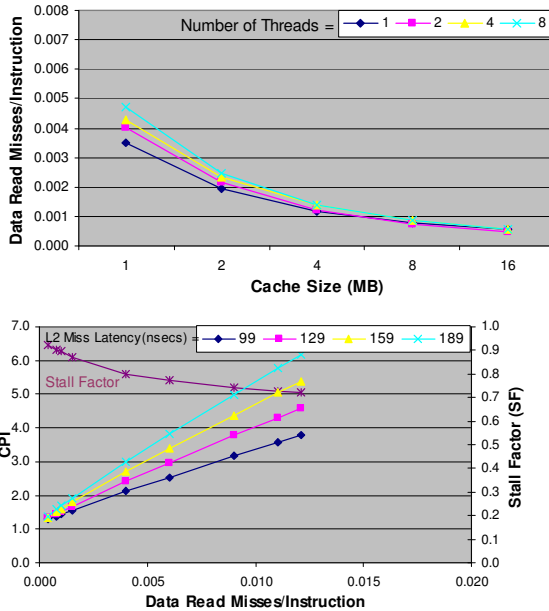


Figure 6: Data Read Miss Impact

4.3 Data Write Miss Sensitivity

Figure 7 shows the impact of the data write misses. The application performance is least sensitive to the data write misses (stall factor is between 0.4 and 0.5). Also, the data write misses quickly reaches an asymptote of 0.0015 misses per instructions. This is because of the constant object creation rate of ~100KB per million instructions. A newly created

object in the Java heap causes compulsory misses, have very short lifetimes, and tend to perturb the caches acting almost like a streaming workload. A replacement policy that brings these lines as less than most recently used would, probably, reduce the perturbation and decrease the DRMPI and IRMPI.

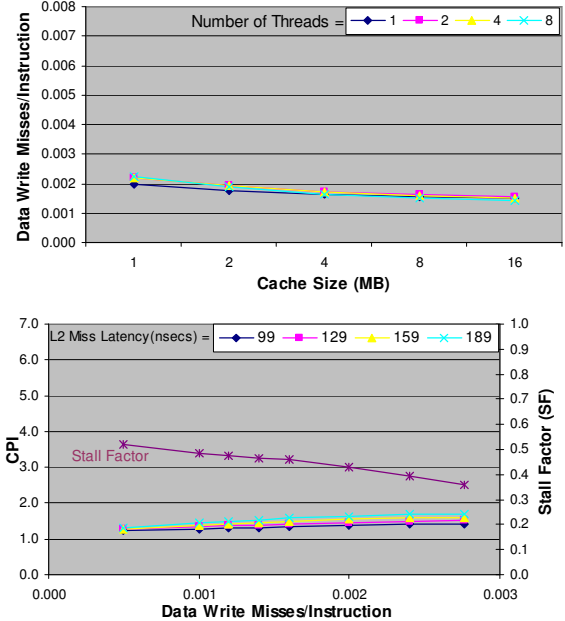


Figure 7: Data Write Miss Impact

5. Analysis

	<i>MPI</i>	<i>% MPI</i>	<i>SF</i>
IR MPI	0.00225	36%	0.55
DR MPI	0.00214	35%	0.83
DW MPI	0.00181	29%	0.40
Weighted SF			0.60
Actual SF	0.00620	100%	0.56

Table 1: Memory Access Overlap Effect

Analysis of cache misses per instruction (MPI) sensitivity and stall factor allows us to predict SPECjAppServer2004 behavior on future platforms and provides guidance to architects on what factors to optimize. Our simulation-based method has allowed us some insight into the impact of the memory subsystem. However, the methodology needs to be improved and more simulations done to refine the stall factor since code read, data read, and data write MPI overlap with each other.

Table 1 shows the impact of MPI overlap. For this configuration we have the MPI breakdown with the associated stall-factors (from section 4). If we

take a simplistic approach, we can try taking the weighted sum of the stall-factors and see if it matches the overall stall-factor. We see from Table 1 that the actual stall factor is lower than the weighted stall factor by a difference of 4%. This is because of the fact that the different types of cache miss overlap with each other. When cache accesses overlap, the stall factor decreases since the effective stall time goes down. In the example shown in Figure 8, the first case with back to back accesses will have 100% stall factor. The full latency of memory access is exposed to CPI. In the second case with parallel access, the stall time is cut roughly in half, and the stall factor is closer to 50%.

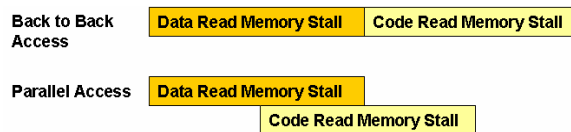


Figure 8: Memory Access Overlap Effect

The interaction of different memory access types is complex. In order to predict stall factors for future platforms, we propose to run additional simulations with different MPI mixes and curve-fit the stall factor to the simulated result.

6. Conclusions and Future Work

In this paper, we have presented trace-simulation based methodology to understand the impact of the memory sub-system on application performance. We have used our technique to characterize SPECjAppServer2004 and noted the effects of memory latency, cache size, cache sharing, code misses, data-read misses, and data-write misses on application performance. Our results indicate that our method needs to be improved to handle memory-level parallelism between different types of memory requests. For future work, we propose to first improve the methodology in that regard and then apply our method to a wide suite of applications so as to provide the research community with a rich data-set related to memory sensitivity.

References

[1] www.spec.org
 [2] L. Spracklen, Y. Chou, S.G. Abraham, "Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications", 11th International Symposium on High-Performance Computer Architecture, 2005.

[3] L.A. Barroso, K. Gharachorloo, E. Bugnion, "Memory System Characterization of Commercial Workloads", 25th International Symposium on Computer Architecture, 1998
 [4] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt and D. Newell, "Exploring the Cache Design Space for Large-Scale CMPs," 1st Workshop on Design, Architecture and Simulation of CMP (dasCMP), 2005.
 [5] E. Nurvitadhi, N. Chalanant, S. Lu, "Characterization of L3 cache behavior of SPECjAppServer2002 and TPC-C", International Conference on Supercomputing, 2005.
 [6] C. Kim, D. Burger, S. W. Keckler, "Nonuniform Cache Architectures for Wire-Delay Dominated On-Chip Caches," IEEE Micro 23(6): 99-107 (2003)
 [7] C. Liu, A. Sivasubramaniam, and M. Kandemir, "Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs," 10th IEEE Symposium on High-Performance Computer Architecture, 2004.
 [8] K. Olukotun, B. A. Nayfeh, et. al., "The case for a single-chip multiprocessor," Proceedings of the 7th International Conference on Architectural support for Programming Languages and Operating Systems, 1996.
 [9] Y. Chou, B. Fahs, S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism", 31st International Symposium on Computer Architecture, 2004
 [10] B. Maron, T. Chen, D. Vianney, B. Olszewski, S. Kunkel, A. Mericas, "Workload characterization for the design of future servers", IEEE International Symposium on Workload Characterization, 2005
 [11] Ravi Iyer, "Performance Implications of Chipset Caches in Web Servers", IEEE International Symposium on Performance Analysis of Systems and Software, 2003
 [12] Z. Zhang, Z. Zhu and X. Zhang, "Design and Optimization of Large Size and Low Overhead Off-Chip Caches", IEEE Transactions on Computer, vol. 53, no. 7, July, 2004
 [13] Ravi Iyer, M. Bhat, L. Zhao, R. Illikkal, S. Makineni, M. Jones, K. Shiv and D. Newell, "Exploring Small-Scale and Large-Scale CMP Architectures for Commercial Java Servers," IEEE International Symposium on Workload Characterization (IISWC 2006), 2006.
 [14] K. Shiv, Ravi Iyer, M. Bhat, L. Zhao, R. Illikkal, S. Makineni, M. Jones and D. Newell, "Addressing Cache/Memory Overheads in Enterprise Java Servers," IEEE International Symposium on Workload Characterization (IISWC 2007), Oct 2007.

Scalability study of a Java Application Server on two multi-core systems

Yohei Ueda

Hideaki Komatsu

Toshio Nakatani

IBM Toyko Research Laboratory
{yohei,komatsu,nakatani}@jp.ibm.com

Abstract

To improve scalability of Java application servers, multiple JVM instances are often used on a single machine in production environments. In this paper, we studied scalability of a Java application server, SAP EP-ESS benchmark, on the two multi-core systems, Sun's single-socket 8-core Niagara system and Intel's dual-socket quad-core Clovertown system, with three configurations of 1, 2, and 4 JVMs. On the Niagara system, 2 JVMs achieved the highest throughput. In contrast, on the Clovertown system, 4 JVMs achieved the highest throughput. We investigated the micro-architectural performance data, and we concluded that this is due to the smaller cache of Niagara.

1. Introduction

To meet the increasing demands on throughput and overcome power constraints, multi-core processors are the most promising approach for the design of future server systems. IBM shipped the first dual-core processors in 2001 [1]. Lately Intel and AMD shipped 4-core processors [2][3], while Sun Microsystems already shipped 8-core processors [4][5]. Several manufacturers have plans to advance this trend further by increasing the scale of chip multi-processing (CMP) and simultaneous multi-threading (SMT). When we shift from single-core processors to multi-core processors, we encounter various scalability problems.

There is a design choice to adopt either simple cores or complex cores [6] for the design of the multi-core processors. A simple core typically has a lower clock rate and in-order pipelines with smaller cache memories. For example, Sun's UltraSPARC T1 (Niagara) and UltraSPARC T2 (Niagara 2) are such multi-core processors with simple cores. A complex core typically has a higher clock rate and out-of-order pipelines with larger cache memories. For example, IBM POWER5 and POWER6, Intel Xeon 5300 Series (Clovertown) and 5400 Series (Harpertown), and AMD Opteron 8300 Series (Barcelona) are such multi-core processors with complex cores.

To archive good scalability of Java application servers on a single multiprocessor system, multiple JVMs are widely used in production environments. We can avoid typical bottlenecks such as lock contention with multiple JVMs. Some studies have been made for scalability analysis of Java workloads on multi-core systems [7][8], but their main focus has been on the scalability of a single JVM. In contrast, our focus is on studying the scalability of multiple JVMs on multi-core systems. For our experiments, we chose a Java Web application server, SAP EP-ESS benchmark, as a representative commercial workload and ran it on Sun's Niagara and Intel's Clovertown systems to understand its scalability on both systems.

With a single JVM, we observed heavy lock contention on both Niagara and Clovertown, and thus we used multiple JVMs to alleviate this lock contention. The results show that 2 JVMs achieved the best throughput on Niagara, whereas 4 JVMs achieved the best throughput on Clovertown. The performance degradation on Niagara was due to increased cache pressure. The smaller cache memories of Niagara cannot contain the memory footprint of 4 JVMs.

To achieve better scalability on simple cores, a Java application must be scalable on a single JVM. In the case of the EP-ESS benchmark, the bottleneck lies on high lock contention in accessing a hash table in the application. After we reduced lock contention using Java concurrent library, the throughput scaled almost linearly even on a single JVM on Niagara.

The rest of this paper is organized as follows. In Section 2, we describe the benchmark we used to

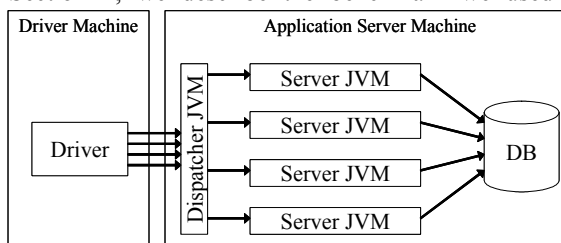


Figure 1: 2-tier configuration of the EP-ESS Benchmark

measure performance on the two multi-core systems. In Section 3, the experimental environments are described. We present the performance results and analysis in Section 4. We discuss how we improved scalability of the application server in Section 5. Finally, we conclude this paper in Section 6.

2. Benchmark Overview

The SAP Employee Self-Service Portal (EP-ESS) benchmark [9] is the only Java benchmark for SAP’s application platform (SAP NetWeaver) [10]. It provides Web interfaces to human resource information including work time recording and personal information updating. Other SAP benchmarks are written in SAP’s proprietary language ABAP (Advanced Business Application Programming).

In the benchmark runs, the Web traffic to the application server tier is generated by a driver program running on an external machine. It simulates the Web transactions of multiple client users and records their response times. The benchmark score is based on how many client users can be served while meeting a QoS requirement in the average response time that should be within 2 seconds.

Each user submits the following Web transactions: (1) log on, (2) choose work time, (3) choose leave request, (4) choose leave request overview, (5) choose personal information, (6) choose address, (7) choose bank information, (8) choose benefits and payment, and (9) log off. The seven transactions from steps 2 to 8 are submitted in a loop, and the loop is repeated. The user think time is 10 seconds, so each loop takes at least 70 seconds. A benchmark run consists of a ramp up phase, a warm up phase, a steady state phase, and a ramp down phase. The response times are recorded only during the steady state phase.

Figure 1 shows the software configuration of the application server machine for the SAP-EP ESS benchmark, where multiple instances of the server JVM handle these requests forwarded by the dispatcher JVM. The back-end database is collocated with the server JVMs on the same machine.

3. System Setup

3.1. Hardware

To compare the performance differences between the two multi-core systems, we used Sun UltraSPARC T1 (Niagara) as the simple core processor, and Intel Xeon E5345 (Clovertown) as the complex core processor. Table 1 shows the specification of the Niagara and Clovertown processors in more detail.

The Niagara system is a Sun Fire T2000 with a single 8-core 1.2GHz UltraSPARC T1 processor. Each core has four hardware threads, which share a 16 KB L1 instruction cache and an 8 KB L1 data cache. All

Table 1: Processor Specifications

	Niagara	Clovertown
CPU	UltraSPARC T1	Xeon E5345
Clock	1.2 GHz	2.33 GHz
Socket	1	2
Core	8 cores / socket	4 cores / socket
SMT	4 HW threads / core	1 HW thread / core
L1 cache	Inst: 16 KB / core Data: 8 KB / core	Inst: 32 KB / core Data: 32 KB / core
L2 cache	Unified: 3 MB/socket	Unified: 8 MB / socket
ITLB	64 entries / core	128 entries / core
DTLB	64 entries / core	L0: 16 entries / core L1: 256 entries / core

Table 2: JVM configurations

	Niagara	Clovertown
JVM	Sun HotSpot JVM 1.4.2	IBM J9 JVM 1.4.2
GC Policy	ParNewGC	gencon
Page Size	4 MB	4 KB
Java Heap Size	8GB total	8GB total

eight cores share a 3 MB L2 unified cache. The Niagara system has 32 GB of DDR2-533 SDRAM for main memory, 10,000 RPM SAS hard disks, and Gigabit Ethernet ports.

The Clovertown system is an IBM BladeCenter HS21 XM with two quad-core 2.33 GHz Xeon E5345 processors. Each core has only a single hardware thread, which has a 32 KB L1 instruction and a 32 KB L1 data cache. Two cores share a 4 MB L2 unified cache. A single Clovertown processor has 2 L2 caches, so it has 8 MB L2 cache. A single socket supports 4 hardware threads, and our dual-socket machine totally supports 8 hardware threads. The Clovertown system has 16 GB of DDR2-667 SDRAM for main memory, a 10,000 RPM SAS hard disk, and Gigabit Ethernet ports.

For both Niagara and Clovertown, we used the same Intel machine as the driver of the benchmark, and the application server machines and the driver machine are connected via a Gigabit Ethernet network.

3.2. Software

We used SAP NetWeaver 7.0 (2004s) for the application server and IBM DB2 Universal Database V8.2 for the database. We used a two-tier configuration. That is, both the application server and the database were located on the same machine. According to the documentation provided by SAP, database activities in the steady state of benchmark runs are almost negligible, allowing us to use this two-tier configuration.

SAP NetWeaver supports multiple instances of application server JVMs, which run with the dispatcher JVM on the same machine. The dispatcher JVM receives Web requests from clients and then forwards them to one of the application server JVMs. Even if only a single instance of the application server JVM is used, a dispatcher JVM must be used.

In the Niagara system, we used Solaris 10 and Sun HotSpot Java VM 1.4.2 (64 bit). This JVM was not the latest version, but it was required for SAP NetWeaver 2004s. Table 2 shows the JVM configurations we used. We used an 8 GB Java heap, 25% of which was assigned for the young space and the rest was assigned for the old space for the generational garbage collection. We specified the large pages. That is, 4 MB memory pages were used for the Java heap. When multiple JVM instances were used, we used the same total size of Java heaps among the JVMs as the case for a single JVM. That is, we used 4 GB Java heaps for each of the 2 JVMs, while 2 GB Java heaps for each of the 4 JVMs. For performance monitoring tools, we used vmstat for OS-level system usages, busstat for memory traffic, nicstat for network load, and cpustat for hardware performance counters for the Niagara system [11].

In the Clovertown system, we used Red Hat Enterprise Linux 5 with Linux kernel 2.6.18 and the IBM J9 Java VM 1.4.2 (64 bit). Sun does not provide a 64-bit HotSpot JVM of this version for Linux, so we used the J9 JVM instead. The specified options for Java heap and garbage collection were the same as those used for the Niagara system except for the large page support. This version of the J9 Java VM does not support large pages in Linux, so default 4 KB memory pages were used for the Java heap. For performance monitoring tools, we used vmstat for OS-level system usages, nmon [12] for the usage of various resources, and oprofile for the hardware performance counters for the Clovertown system.

4. Performance Results

First we measured scalability in the throughput while increasing the number of cores. On the Niagara system, all four hardware threads were used for each core. Then we measured scalability in the throughput while increasing the number of hardware threads per core on the Niagara system. Thread scalability was not measured on the Clovertown system because it does not have multiple hardware threads per core. Finally we measured performance improvement on Niagara, when we eliminated the hottest lock by using the Java concurrent library [13]. Throughout these measurements, we collected system usage and micro-

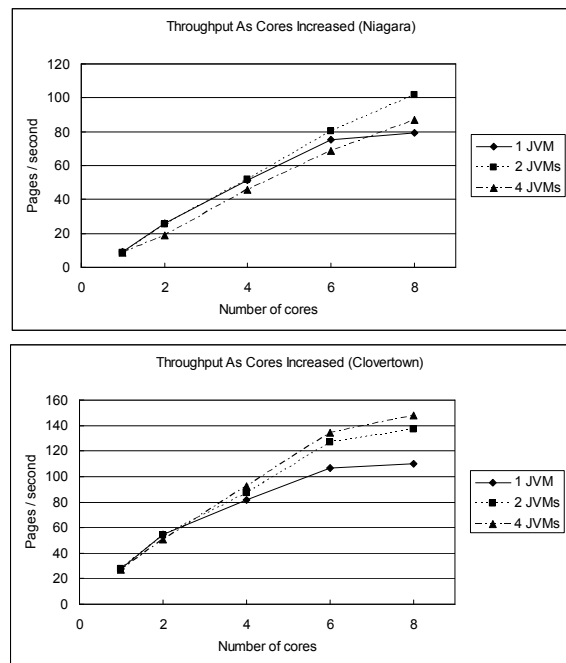


Figure 2: Core Scalability

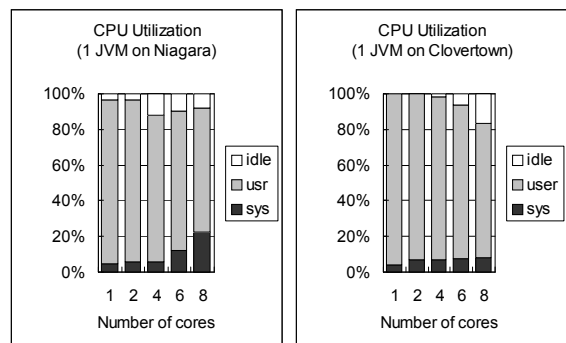


Figure 3: CPU Utilization

architectural performance data such as CPI and cache misses.

4.1. Core Scalability

Figure 2 shows the throughput of the EP-ESS benchmark on the Niagara and Clovertown systems, both with 1, 2, 4, 6, and 8 cores. On the Niagara system, all four hardware threads are used in each core. On the Clovertown system, 1 socket was used for the measurements with 1, 2, and 4 cores, and 2 sockets were used with 6 and 8 cores. We ran the benchmark with 1, 2, and 4 application server JVMs.

In the single JVM configuration, the throughput started to saturate from 6 cores on Niagara and 4 cores on Clovertown. Figure 3 shows the CPU utilization of both systems. On the Niagara system, we observed relatively high percentage of both system and idle CPU

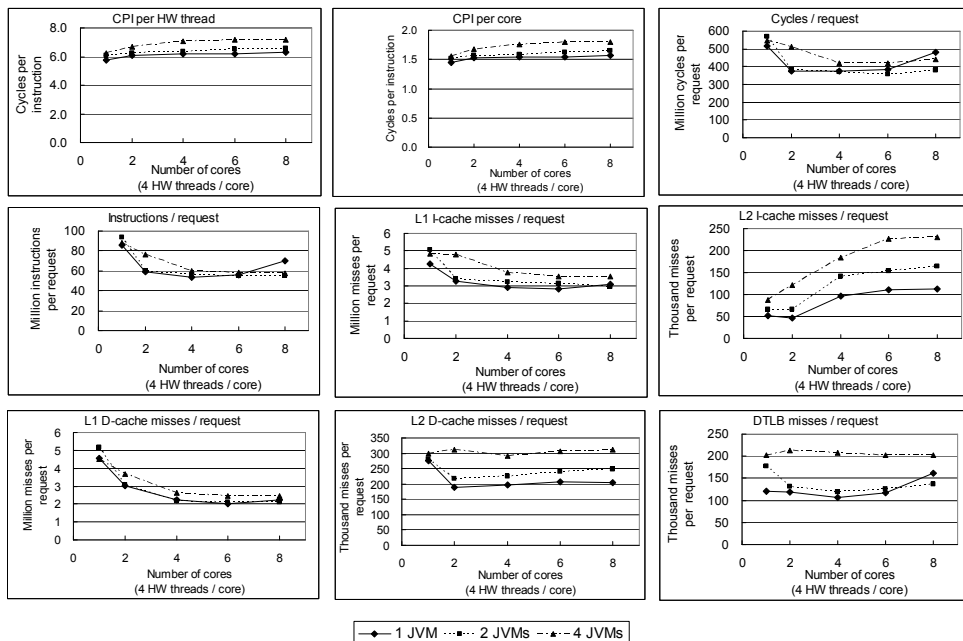


Figure 4: Micro-architectural performance on Niagara

times on 4, 6, and 8 cores. On the Clovertown system, we observed a relatively high percentage of idle CPU time on 6 and 8 cores. We believe this is why the single JVM showed limited scalability.

We analyzed the Java thread stack dumps and execution profiles, and concluded that accessing a hash table caused heavy lock contention. This is an instance of `java.util.Hashtable` and every access to a `Hashtable` object requires synchronization. We also analyzed the Clovertown results and found similar contention on a hash table.

We measured 1, 2, and 4 JVM configurations on both systems, as described in Figure 2. On Niagara, 2 JVMs achieved almost linear scaling. Although 4 JVMs also scaled well, 2 JVMs had the best throughput for Niagara.

To find out the cause of performance degradation using 4 JVMs on Niagara, we analyzed the micro-architectural performance data from Figure 4. We looked into both per-instruction metrics, such as cycles per instruction (CPI), and per-request metrics, such as million cycles per request. The per-instruction metrics will be improved when the application has more lock contention. This is because spending more time in a spin lock will improve the code and data locality without doing any useful work. For this reason, we use per-request metrics for measuring cache and TLB misses.

In the single JVM case, the cycles per request became worse when the number of cores increased, and the instructions per request also became worse.

The cycles per request of 2 JVMs were better than those of 4 JVMs, while the instructions per request of 2 and 4 JVMs were very close. This degradation with 4 JVMs was caused by increased cache pressure, as we can see in the graphs of L2 I- and D-cache misses and DTLB misses in Figure 4. Niagara has smaller cache memories and TLB, where a larger memory footprint of more Java processes caused significant performance degradation. The L2 I-cache misses with 4 JVMs were twice as frequent as those of 1 JVM. The L2 D-cache misses with 4 JVMs were 50% more frequent than those of 1 JVM. Even though we use 4 MB large pages for the Java heap, the DTLB misses with 4 JVMs were twice as frequent as those of 2 JVMs. The increased cache pressure was caused by the increased memory footprint of 4 JVMs, where every JVM executes the same set of Java methods independently without sharing any code and data.

In contrast to the results on Niagara, 4 JVMs had the best throughput for Clovertown.¹ Figure 5 shows the micro-architectural performance data on Clovertown. Both cycles and instructions per request were improved when multiple JVMs were used. This is mainly due to reduction in lock contention in accessing the hash table. Cache thrashing did not occur on Clovertown. However, the throughput was saturated

¹ Additional experimentation on Clovertown confirmed that the throughput with 8 JVMs was worse than that with 4 JVMs.

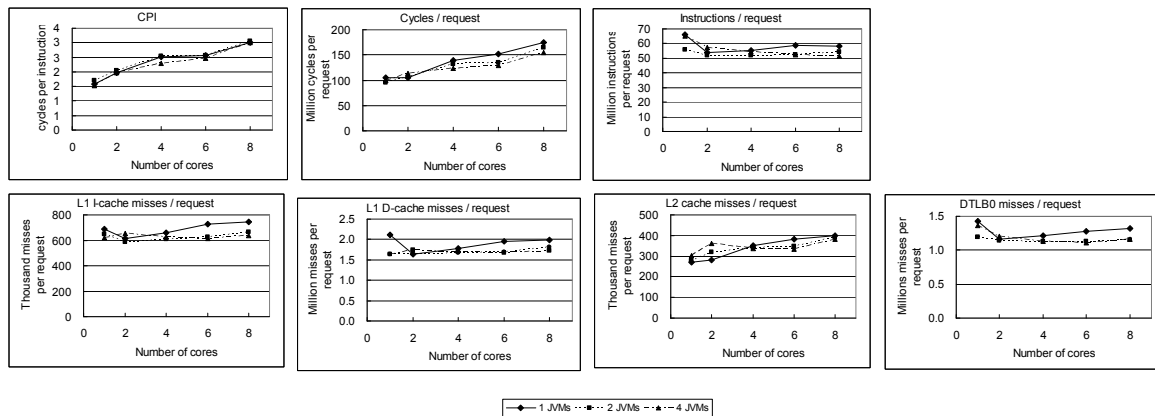


Figure 5: Micro-architectural performance on Clovertown

around 6 cores, and using 8 cores did not improve the throughput very much.

4.2. Thread Scalability

Figure 6 shows scalability in the throughput performance when we increased the number of hardware threads per core while using all 8 cores on the Niagara system. For the 1-JVM configuration, the throughput was saturated at 3 hardware threads per core, and 4 hardware threads degraded the performance. This was due to the lock contention problem described earlier. With 3 hardware threads, the peak performance was 2.2 times better than the single thread performance.

We also measured scalability in the throughput performance using 2 JVMs, which was the best configuration on Niagara. In this case, 4 hardware threads performed better than 3 hardware threads per core. This is because lock contention was alleviated with 2 JVMs. The peak performance was 3.0 times better than single thread performance. In other word, it performed at 74% of the theoretical peak performance with 4 hardware threads.

5. Scalability Improvement

There are two approaches in improving scalability with simple cores. One approach is to improve single-JVM scalability by reducing lock contention in the application. The other approach is to improve multiple-JVM scalability by assigning processor affinity.

5.1. Single-JVM Scalability

In general, reducing lock contention can improve the throughput of the application. We identified a highly contended lock in accessing a hash table in the SAP EP-ESS benchmark. We used a bytecode rewriting tool called Javassist [14], to replace the java.util.Hashtable object with java.concurrent.ConcurrentHashMap from the Java concurrent library.

Figure 7 shows the original result with 1 JVM and the result using the concurrent library with 1 JVM.

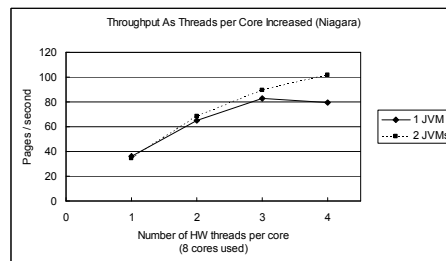


Figure 6: Throughput as threads per core increased

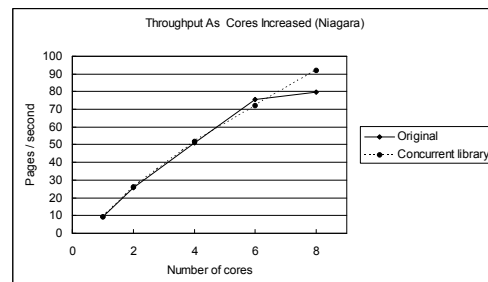


Figure 7: Comparison between results without and with concurrent library

Before rewriting the source code, the throughput was saturated on 6 cores. After rewriting, it scaled well on up to 8 cores. As a result, we achieved 15% throughput improvement by using the concurrent library [13].

5.2. Multiple-JVM Scalability

In general, assigning processor affinity (that is, assigning each process to a set of the hardware threads) can improve memory locality and thus scalability of the application. For our experiment, we used the `prset` command available for Solaris to bind each JVM to a set of dedicated hardware threads.

Figure 8 shows the utilization of hardware threads when we used 4 JVMs, each of which is bound to 1 and 3/4 cores. For example, the hardware threads 1, 2, and 3 of the first core, and 4, 5, 6, and 7 of the second

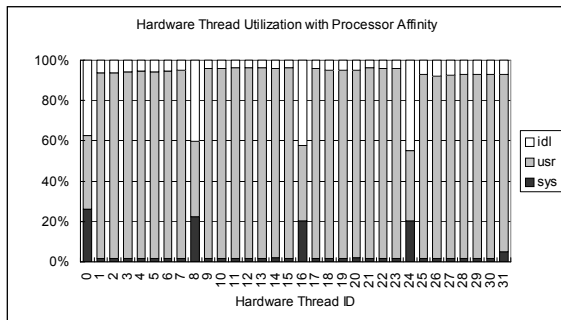


Figure 8: Hardware Thread Utilization with processor affinity (4 JVMs)

core were used for the first application server JVM. The other processes, such as the dispatcher JVM, DB processes, and OS processes, ran on the remaining hardware threads on 4 different cores. That is, they ran on the hardware threads 0, 8, 16, and 24. This binding improved the throughput of 4 JVMs by 7% over that of 4 JVMs with no affinity. This configuration archived the best throughput among all the configurations we tested with 4 JVMs. However, we still observed a large idle CPU time on the hardware threads 0, 8, 16, and 24. Reducing the number of these hardware threads from 4 to 3 did not improve the throughput. In fact, this best throughput with 4 JVMs was still 8% lower than that of 2JVMs. Assigning affinity did not improve the throughput of 2 JVMs.

We also used processor affinity on Clovertown using taskset command available for Linux to bind JVMs to the cores. However, assigning affinity did not improve the throughput of 4 JVMs.

5.3. Potential Area for Improving Multiple-JVM Scalability

To reduce the memory footprint of the multiple-JVM configuration, we need to increase data sharing among JVMs as well as reducing memory usage of each JVM. A JVM needs various metadata such as class information outside the Java heap. The latest version of IBM J9 JVM can share class files among JVMs. This feature may help reduce L2 D-cache misses. To reduce I-cache misses, we need to share binary code compiled by the Just-In-Time compiler. Sharing JIT-compiled code among JVMs remains an interesting challenge.

6. Conclusions

This paper describes our performance analysis of the SAP EP-ESS benchmark on the two multi-core systems: the Sun Niagara and Intel Clovertown systems. For a single JVM, lock contention was observed on more than 4 cores in both systems. For multiple JVMs, 2 JVMs achieved the highest throughput on Niagara, while 4 JVMs achieved the

highest throughput on Clovertown. This is because Niagara comes with smaller caches and TLB, whereas Clovertown comes with larger caches and TLB.

To achieve better scalability on Niagara, the application must be more scalable with a single JVM. We confirmed this by reducing lock contention by using a Java concurrent library. As a result, we obtained a 15% performance gain on Niagara, and the throughput scaled almost linearly.

7. References

- [1] S. F. H. L. J. Tendler, S. Dodson and B. Sinharoy. POWER4 System Microarchitecture. IBM Technical White Paper, Oct. 2001.
- [2] Intel Corporation. Quad-core for servers. <http://www.intel.com/quadcoreserver/>.
- [3] AMD Inc. AMD Multi-core Processors. <http://multicore.amd.com/>.
- [4] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded SPARC Processor. IEEE Micro, 25(2): 1-29, March/April 2005.
- [5] U. Nawathe, M.Hassan, L. Warriner, K. Yen, B. Upputuri, D.Greenhill, A.Kumar, H. Park. An 8-core, 64-thread, 64-bit, power efficient SPARC SoC. 2007 IEEE International Solid-State Circuits Conference (ISSCC), February 2007.
- [6] K. Olukotun, L. Hammond, J. Laudon. Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency. Morgan & Claypool Publishers, 2007.
- [7] D. Kaseridis and L. K. John. CMP/CMT Scaling of SPECjbb2005 on UltraSPARC T1. Tenth Workshop on Computer Architecture Evaluation using Commercial Workloads, February 2007.
- [8] J. H. Tseng, H. Yu, S. Nagar, N. Dubey, H. Franke, P. Pattnaik, H. Inoue and T. Nakatani. Performance Studies of Commercial Workloads on a Multi-core System. 2007 IEEE International Symposium on Workload Characterization (IISWC), September 2007.
- [9] SAP AG, SAP EP-ESS benchmark. <http://www.sap.com/solutions/benchmark/ep-ess.epx>.
- [10] SAP AG, SAP NetWeaver. <http://www.sap.com/platform/netweaver/>.
- [11] R. McDougall, J. Mauro, and B. Gregg. Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris. Prentice Hall PTR, 2006.
- [12] IBM. nmon performance: A free tool to analyze AIX and Linux performance. http://www.ibm.com/developerworks/aix/library/au-analyze_aix/
- [13] D. Lea, Overview of package util.concurrent Release 1.3.4. <http://g.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html>
- [14] S. Chiba. Javassist --- A Reflection-based Programming Wizard for Java. In Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java October, 1998

Memory Characterization of Emerging Recognition-Mining-Synthesis Workloads for Multi-Core Processors

Yu Chen¹

Wenlong Li²

Junmin Lin¹

Zhizhong Tang^{1*}

Aamer Jaleel³

¹Department of Computer Science & Technology, ²Intel China Research Center, ³Intel Corporation, VSSAD
Tsinghua University, Beijing, China
chenyu00@mails.tsinghua.edu.cn
wenlong.li@intel.com
aamer.jaleel@intel.com

Abstract

World data is doubling every three years and is now measured in exabytes - a billion billion bytes. Recognition, mining, and synthesis (RMS) are three fundamental classes of processing capabilities to make meaningful use of this enormous sea of information. This paper characterizes the memory behavior of some RMS applications for multi-core processors by examining their working set sizes, data sharing behavior, and spatial data locality. Our study reveals that these RMS workloads are memory intensive, have large working-set sizes, exhibit a significant amount of sharing, and show strong streaming access characteristics. We find that for these workloads a) large DRAM caches can be useful to address their large working-set sizes; b) shared last level cache (LLC) provides better cache performance than private cache in terms of number of last-level cache misses; c) hardware prefetching provides significant performance benefit as it captures the large amount of spatial locality in these workloads.

1. Introduction

As the number of cores on chip keeps increasing, incorporating multiple cores on the same die presents many memory hierarchy design choices to the processor architects, making what already was a difficult design decision even more challenging. A detailed understanding of the memory behavior of the emerging parallel applications is crucial for architects when making key memory hierarchy design choices. Inspired by an analysis of the SPLASH-2 benchmark suite [20] and last level cache performance study of bioinformatics workloads [8], this paper investigates the memory system requirements of the critical emerging Recognition, Mining, and Synthesis (RMS) application domains [3][5] on multi-core processors, and provides architectural insights and

recommendations to help define cache hierarchy of future chip-multiprocessor (CMP) running these workloads.

2. Characteristics and methodology

We study the RMS workloads by examining the following three characteristics to understand the interaction between workloads and cache hierarchy on future CMPs:

- (i) **Working-Set Size:** the working-set size of an application implies its temporal data locality. By evaluating the cache performance versus cache size, we can understand how large the cache should be designed to hold the total working set.
- (ii) **Sharing Behavior:** one key issue in designing the memory subsystem will be the policy for the on-die last-level cache. The most important characteristics that drive this design would be the amount and type of sharing exhibited by the multi-threaded workloads.
- (iii) **Streaming Behavior:** besides temporal locality, spatial locality is also one important property when designing the memory subsystem. By evaluating the streaming degree, we can correlate it with the effectiveness of hardware stride prefetcher and cache line size.

In this paper, we adopt the instrumentation driven simulation (IDS) methodology, which has advantage of fast, simple, storage saving, and supporting full-run of applications, to conduct memory system studies [7]. For our workload characterization studies, we use CMPsim [7], an instrumentation driven simulator for CMPs.

We model a two-level cache hierarchy. For the purpose of this study, we assume a perfect L1 instruction cache. The L1 data cache is 8-way, and 32KB. The L2 cache (LLC) is 16-way, 4/8/16/32/64MB, and either private or shared across all cores. All caches in the simulated hierarchy use 64B line size, non-inclusive, MSI invalidate-based protocol, write-back policy, and LRU replacement policy.

The host machine is a 16-way Intel® Xeon® shared memory system. All workloads are compiled using Intel's

* Supported partially by the National Natural Science Foundation of China under Grant No. 60573100, and 60773149

C/C++ compilers with optimization flags -O3 on a SuSe 9.3 32-bit system and Windows 2003 32-bit system.

3. RMS workloads overview

Today, our society is generating massive amounts of multimedia data (such as audio, video, image and text etc) as the world continues to go digital. Due to required heavy computations, however, we don't have the capability to enjoy the rich resources. There is an urgent need for scalable, adaptable and programmable computing architectures that have the strong capability to recognize, mine and synthesize(RMS) the large multimedia data [3][2][4]. In this section, we provide a brief introduction to the RMS workloads.

3.1 Recognition workloads

Player Detection: player detection is to find multiple players within the playfield, track their moving trajectories, and identify their labels in the soccer video [13]. In the system, players are first detected by boosted cascade Haar features. Then based on unsupervised prior learned player appearance models, each player is automatically categorized into three classes: team A, team B, and the referee. A player's moving trajectory is generated by a forward-backward tracking algorithm in which nearest neighbor data association is utilized [18].

Ball Detection: this application is to locate the ball and track its moving trajectory during the play in soccer video [21]. It can be used to find the pattern of the ball trajectory, and replay the interesting scenes, such as a goal. Due to the difficulty of identifying the unique ball with much false noise, shape and color change led by fast motion, it tracks multiple ball candidates and identifies the most possible one through their trajectories discrimination and verification in a three-level processing scheme [18].

Face Analysis: face detection and tracking are important technologies and pre-requirement for many person analysis relevant applications, such as cast indexing. The Boosting learning based face detection algorithm by Viola and Jones is the most successful one, which uses Haar wavelet features to

learn a cascade detector. In the detection phase, the learned detector slides by a window over the image to detect whether the window contain a face or not. Face tracking [12] is an extension of face detection technology, which can detect person's continuous faces from a video sequence.

3.2 Mining workloads

FIMI: Frequent Itemsets Mining (FIM) is a very commonly used and well-studied data-mining problem. It tries to discover groups of items or values that co-occur frequently in a transactional data set. Many FIMI (FIM Implementation) algorithms have been proposed in literature, including FP-growth and Apriori-based algorithms. The FIMI workload [14] is by far the fastest FP-growth implementation based on the FP-Zhu package, which includes three stages: first-scan, FP-tree construction, and mining.

PageRank: PageRank was developed by Lawrence and Sergey [11]. It is a link analysis algorithm that assigns a numerical weighting to each element of a hyperlinked set of documents, such as the World Wide Web, with the purpose of "measuring" its relative importance within the set. PageRank is based on the linking structure of the whole web. The basic approach of PageRank is that a document is considered the more important the more other documents link to it.

3.3 Synthesis workloads

Face Animation: animation of human faces has been a useful application in Computer Graphics and games. To simulate dynamic faces, each face has a set of control parameters for facial muscles, eyeballs, eyelids and mouth. For face rendering, triangle meshes are used. Visual appearance can be further improved by using textures from photos and exploiting the simulation of skin structure and reflection properties [9][19].

ODE: ODE is based on open-source Open Dynamics Engine (ODE) physics engine [10]. Its two main components are a rigid body dynamics simulation engine and a collision detection engine. It simulates destructions of the single castle which is composed of oriented bounding boxes hit by the fly-

Table 1: Basic workload characteristics

Workloads	Parameters	Size of Data Input	Instruction	% Memory	% Memory Read	L1 Misses / 1000 Inst
Player Detection	1m MPEG-2 video	23MB, 720x576 resolution	158.4B	51.2%	39.6%	17.29
Ball Detection	1m MPEG-2 video	23MB, 720x576 resolution	108.3B	49.6%	41.5%	7.52
Face Analysis	26s MPEG-1 Video	4.4MB, 352x288 resolution	152.9B	53.2%	44.3%	32.11
FIMI	990k transactions and mini-support=800	30MB, real dataset Kosarak from http://fimi.cs.helsinki.fi/data/	59.3B	49.7%	36.2%	3.52
PageRank	7M nodes and 194M arcs	48MB, real indochina2004 data from http://law.dsi.unimi.it/index.php?option=com_include&Itemid=65	137.8B	46.5%	38.4%	20.48
Face Animation	1 frame, 80K particles 372K tetrahedral	6.7MB face model	42.7B	56.8%	40.5%	5.86
ODE	24 balls, 1 tower of 100x100 boxes	<1K	9.9B	56.8%	42.8%	18.93
Ray Tracer	860K vertices, 673K triangles, 3 lights	beetle dataset, with 44MB model, and 77MB KD-Tree	61.1B	41.2%	31.1%	1.23

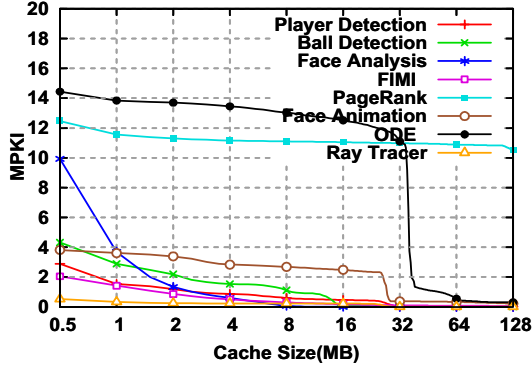


Figure 1: Cache performance of single-threaded RMS workloads as a function of cache size

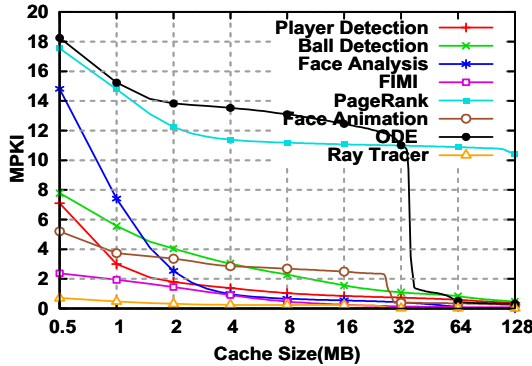


Figure 2: Cache performance of 16-threaded RMS workloads as a function of cache size

ing balls. At each frame, Collision detection engine detects collision among moving objects. Then physics solver is called to find contact forces to respect non-penetration constraints.

Ray Tracer: ray tracing is a global illumination based rendering method in 3D computer graphics [4]. It works by tracing a path from an imaginary eye through each pixel in a virtual screen, accumulating the contribution of each light in the scene to that pixel. As each ray is cast from the eye, it is tested for intersection versus objects within the scene. Ray tracing facilitates advanced optical effects, such as accurate simulations of reflection and refraction and is broadly used to generate realistic graphics.

3.4 Basic workload characteristics

Table 1 shows the chosen parameters and dataset sizes for the workloads, as well as the characteristics of the workloads run to completion in single-threaded mode using CMP\$im. We choose application parameters and datasets such that they represent realistic and time consuming executions. The instruction profile indicates that the workloads consist of roughly 41-57% memory instructions. We also observe that memory read instructions constitute 71-83% of total memory instructions. The large share of memory instructions, especially memory read instructions, is to be expected as these workloads work through large amounts of data in an attempt

to discover meaningful patterns or relationships, and it also solidifies the fact that these RMS workloads are data intensive by nature. From Table 1, we can see except for Ray Tracer, which have relatively low L1 cache misses, all other applications suffer from high L1 cache misses, which are caused by the inherent nature of streaming data access, and thus insufficient data reuse is exhibited.

4. Working set Size

We start with the working set analysis of these parallel RMS workloads by evaluating their cache performance with different cache sizes. We use a stack distance approach to simulate the performance of multiple cache sizes in one simulation run [15].

Figure 1 presents the data cache sensitivity study for the single-threaded RMS workloads. The figure shows cache sizes in megabytes (MB) on a logarithmic x-axis and misses per one thousand instructions (MPKI) on the y-axis.

The last x-coordinate data of the curves in the figure provide an indication of the total memory footprint of each workload. From the figures, three RMS workloads have total memory footprint greater than 128MB, while seven workloads have total memory footprint greater than or equal to 16MB (16MB for Ball Detection, 26MB for Ray Tracer, 32MB for FIMI and Player Detection, more than 128MB for ODE, Face Animation, and PageRank).

The sudden drops in the curves of some workloads indicate the working set sizes. For example, ODE has a better cache performance with a 40MB cache, since the total number of miss drops down significantly with a 40MB cache. Unlike SPEC workloads (SPEC CPU2000 occurs at a cache size of 1-2MB, while SPEC CPU2006 occurs at a cache size of 16-32MB [6]), the large memory working set sizes of these RMS workloads will continue to put pressure on processor and DRAM architects to reduce the ever growing "memory gap".

Since these RMS workloads operate on a very large working set, the future memory systems are expected to provide high bandwidth and low latency to achieve high performance. The large DRAM cache is a promising technique to achieve this goal. The DRAM cache can be enabled either through 3D stacking or by the use of a multiple-chip package, and its benefits are two-fold: (a) it reduces the bandwidth requirement on external DRAM and hence allows a potential reduction in the pin count out of the package and (b) the latency to and from the DRAM cache is only a fraction of that of external memory. As a result, the DRAM cache is suitable for latency-bound workloads as well as bandwidth-bound workloads. Of course, the benefits of the DRAM cache are largely a function of the miss rate that it provides. In this paper, we evaluate the DRAM cache miss characteristics (from as low as 32M to as high as 128MB) of the multi-threaded RMS workloads. Among these eight workloads in a single-threaded scenario, ODE, Face Animation, and PageRank

will certainly benefit from the use of a DRAM cache as we can see the cache miss per 1000 instructions (MPKI) drops from 13.8 at 1MB cache to 0.6 at 64MB cache for ODE applications, 3.6 MPKI at 1MB cache to 0.37 MPKI at 32MB cache for Face Animation, and 11.6 MPKI at 1MB cache to 10.5 MPKI at 128MB cache. We observe that reduction in MPKI translates into reduced bandwidth demands beyond the last level cache (LLC). Based on the data for ODE and Face Animation workloads, there are 23-fold and 10-fold decreases when we use a large DRAM cache, which reinforces the fact that these data-intensive workloads tend to benefit from a large DRAM cache.

Figure 2 shows the cache performance of 16-threaded RMS workloads. According to Figure 1 and Figure 2, we project that there is a direct correlation between the ratio of global/local data and the performance of LLC from the application/algorithm viewpoint. For PageRank, Face Animation, ODE, and Ray Tracer, as the threads operate on a global working set, and each thread contributes a relatively small amount of thread-local data copies, the overall data footprint does not change significantly with the increase in thread number, and the overall cache performance experiences only a marginal change. However, for FIMI and Face Analysis, besides sharing a global data structure, each thread contributes its own local data copies to the workload data footprint, hence increasing the overall data footprint of the workload. Ball Detection and Player Detection applications, whose working-set sizes grow significantly, have each thread operating on a pretty large local copy of data set, about 12MB per thread for Ball Detection, and 25MB per thread for Player Detection. However, there is a relatively small global working set for them. Therefore the total working set increases near-linearly with the number of threads, and the cache performance becomes poor at the same cache configuration when scaling the number of cores.

Face Animation, Ray Tracer, and PageRank don't show obvious changes in cache performance when scaled to 16 threads. Based on our understanding of these workloads, we believe that the cache performance of these workloads will not scale on a large number of cores, even on 128 cores. So for ODE, Face Animation, Ray Tracer, and PageRank, a DRAM cache is required to hold the large working set to provide good memory subsystem performance. While for FIMI and Face Analysis workloads, there is some increase in cache misses when scaling core numbers from 1 to 16, due to an increased working set. On 16 cores, the working set for FIMI and Face Analysis is about 45MB and 64MB, respectively. Based on these observations and the knowledge of these workloads, we project that the working set of these two workloads will further increase as core numbers increase, and their working set will exceed 128MB on 128 cores. Thus, a large DRAM cache may provide good memory subsystem performance. As for Player Detection and Ball Detection, their working sets scale near-linearly with the number of cores as each thread maintains a large local working set while

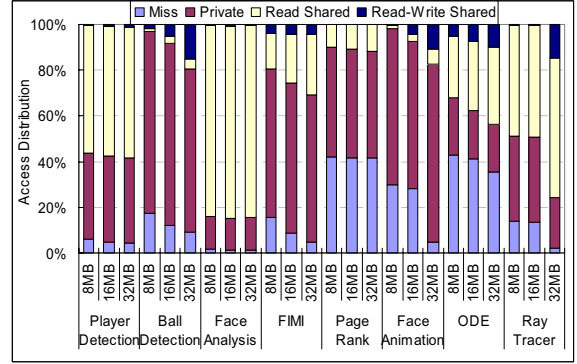


Figure 3: LLC access distribution of 16-threaded RMS workloads with varying cache size

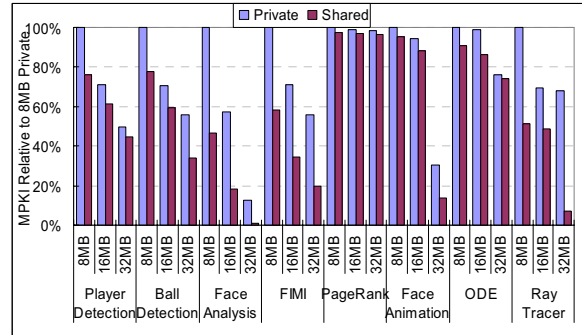


Figure 4: Performance comparison of 16-threaded RMS workloads between shared cache and private cache

the size of global data is much smaller. On 16 cores, the working set of Player Detection and Ball Detection is larger than 128MB; therefore, they are certain to be good candidates for large DRAM caches.

5. Data sharing

The second characteristic of interest is the data sharing behavior, which affects the policies of on-die last-level cache (LLC). Figure 3 shows the LLC access distribution of 16-threaded RMS workloads. We term an access as *read-write shared* if it is a write access which writes data for other threads to read, or it is a read access which reads the data just written by some other thread. We term an access as *read shared* if it reads data which is just previously read by some other thread. We term an access as *private* if it accesses the cache line that is not shared among threads.

Based on the sharing behavior, we can classify these workloads into two categories: (1) the global data structure is partition-able among threads, and the updates to it are thread-private; and (2) a large portion of the global data structure accesses are read-only shared among threads. Ball Detection, FIMI, PageRank, and Face Animation are in the first category. Less than 25% of LLC accesses are to the shared data. For Ball Detection, some threads are responsible for video decoding, while other threads perform feature ex-

traction on the decoded frames, so we observe the read-write data sharing. For FIMI and PageRank, they share a global read-only tree structure and vector, respectively. For Face Animation, each thread operates on a grid of the simulated frame, and they share the adjacent boundary. In contrast, Player Detection, Face Analysis, ODE, and Ray Tracer are in the second category. For Ray Tracer, all threads share a kd-tree structure. For Player Detection and Face Analysis, they share a global training model structure among threads. In conjunction with the working set study, although Player Detection has very large thread-private data structure, most accesses of this workload are to the shared data. Additionally, we observe that the amount of sharing varies with the cache size. Increasing the cache size increases the percent of shared data. This is because conflict and capacity misses in smaller caches can result in the eviction of potential shared data. Increasing the cache capacity reduces the chance of conflict misses and provides opportunity for shared data to exist in the cache. This behavior is prevalent in all RMS workloads.

Having demonstrated that these RMS applications exhibit large amounts of data-sharing, we now analyze the type of data-sharing. From the figure, we can see most shared accesses are in read-read sharing pattern. Based on the type of data sharing exhibited, we conclude that even though the RMS applications expose little read-write shared data, the fact that these workloads exhibit extensive read sharing emphasizes the need for shared caches to avoid unnecessary duplication of data. Also the significant read-read sharing activity implies that the coherence protocol could be further optimized to avoid the unnecessary snoops for power saving.

Based on the working set analysis and data sharing characterization, we observe these RMS workloads have large working set size and exhibit a significant amount of data sharing. We show that the working set size scales with the thread count, and the amount of data sharing exposed is increased when other factors such as cache misses are removed from the increasing cache capacity. To accommodate the large working set and utilize the inherent sharing behavior of these RMS workloads, a large shared-cache is more attractive.

Figure 4 compares the normalized cache performance between shared and private caches with size of 8/16/32 MB for 16-threaded RMS workloads. The misses of private cache configuration with 8MB total size serves as the baseline, and all other configurations are normalized to this one. When using private caches, the total cache is partitioned equally amongst all the cores of the CMP, i.e., for 16 cores, the private cache sizes are 512KB/1MB/8 MB.

As expected, increasing the size of caches for both private and shared configurations helps decrease the overall cache misses. Except PageRank, most workloads show significant improvement in cache performance with the increasing cache size. This behavior is consistent with previous working set analysis. Figure 4 also illustrates that the shared cache configuration always outperforms the private cache with the same size. For example, with a 8MB cache, the reductions of

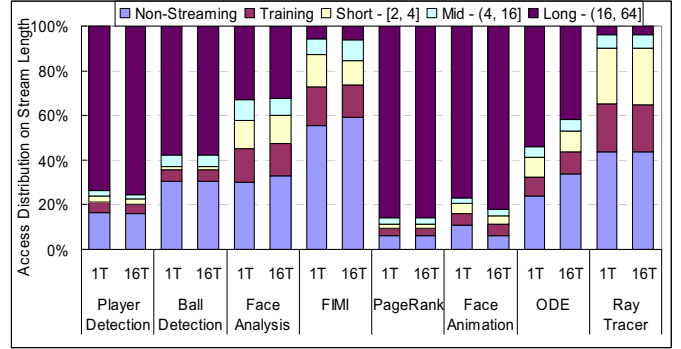


Figure 5: Access distribution on stream length

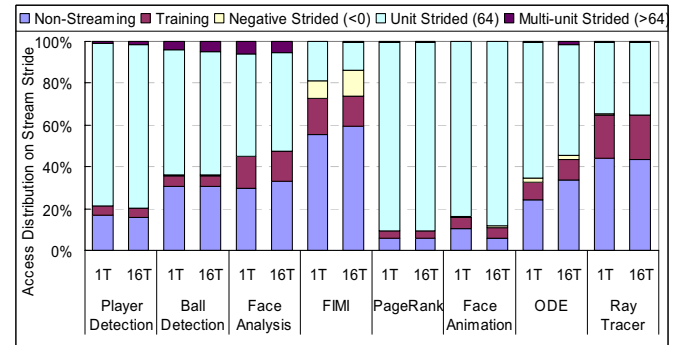


Figure 6: Access distribution on stream stride

misses per kilo instructions (MPKI) for Player Detection, Ball Detection, Face Analysis, FIMI, and Ray Tracer are 24%, 22%, 53%, 42%, 49%, respectively. The improvement of Player Detection, Face Analysis and Ray Tracer can be explained by the large amount of data sharing behavior explored in the previous section.

6. Streaming behavior

The third workload characteristic of interest is spatial locality, which is measured by the streaming behavior. In this work, we define *Stream* as a sequence of memory accesses with a constant non-zero difference between any two adjacent addresses in the sequence [16][17]. The total number of elements in a stream is defined as the *length*. The constant distance in memory address is defined as the *stride*. In the simulator, we implement similar mechanism as that in [17] to detect streams with unit and non-unit strides. In the literature of bridging the growing gap between processor and memory performance, stream is one important metric to represent memory access characteristics, and many memory subsystem optimizations have been proposed to target such kind of memory accesses pattern, such as hardware prefetching [1], streaming buffer [17], and profile-driven optimizations [16], etc.

In our implementation, we use eight stream detection engines to collect the streaming degree on the missed L1 data cache requests of these RMS workloads. That means our

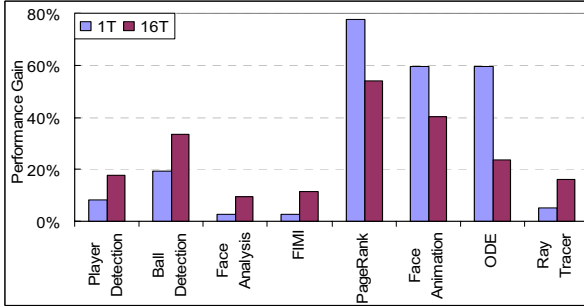


Figure 7: Performance gain of hardware prefetching

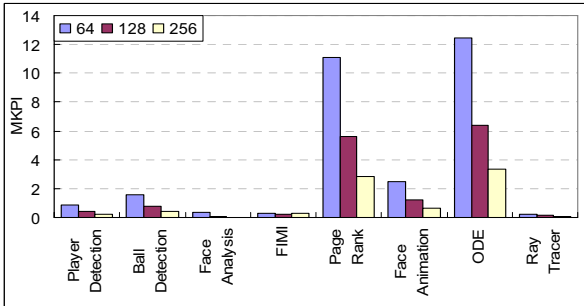


Figure 8 Performance of 16MB LLC with different cache line sizes for 16-threaded RMS workloads

stream detection engine lies in between L1 and L2 data cache, and it detects streams on the L2 cache requests (the data locality has been filtered by the L1 data cache, and the minimum stride value is one cache line size). As there is no PC information for the missed L1 data cache request, we implement the address based stream detector. To simulate similar hardware prefetching when reaching page boundary in most commercial processors, we stop streams at page boundary, and re-train them in the next memory page. For multithreaded programs, we maintain unique stream detection table to each threads, where the dedicated stream detection table is responsible for stream detection of the associated thread in the instrumented application.

Figure 5 and Figure 6 show access distribution on stream length and stream stride. The *non-streaming* access means it doesn't belong to any streams. The *training* access represents the stream recognition part. The *streaming* access means it hits a steady stream. In our implementation, unit stride stream requires one element for training, while non-unit stride stream needs two elements to recognize a stream. As we don't detect streams across page boundary, the maximum stream length is 64 (the page size is 4KB in our experiment).

All workloads fall into three categories according to their streaming degree exposed in Figure 5: (1) most of accesses (quantitatively >70%) are to long streams (streams with length larger than 16); (2) accesses consist of long-streaming (30~60%) as well as non-streaming (20~40%); (3) few accesses are to long streams and most are non-streaming (>40%). From Figure 5, three workloads (Player Detection, PageRank, and Face Animation) are in the first category, and

three (Ball Detection, Face Analysis and ODE) are in the second category. These two classes of workloads expose significant streaming degree. More than 60% accesses are in streams. Only two workloads (FIMI and Ray Tracer) are in the third category, who involve tree and kd-tree traversal, respectively. Moreover, The streaming behavior of these parallel RMS workloads remain flat with increasing thread count as the coarse-grained data level parallelism is extracted to parallelize these RMS workloads. Overall, these RMS workloads have highly predictable access pattern and are good candidates for prefetchers.

Most streams of these RMS applications have unit stride pattern as showed in Figure 6. The dominant unit-stride access pattern indicates these workloads have good spatial locality, thus large cache line size could provide better cache performance by reducing the compulsory misses. In addition, it also indicates that keeping a simple logic for unit stride stream detection on future large scale CMPs would run well for these RMS workloads.

To know the potential impact of streaming behavior on memory subsystem design, we evaluate the performance impact of hardware prefetching and large cache line size in Figure 7 and Figure 8, respectively. Figure 7 presents the performance benefit of hardware prefetching measured on the 16-way SMP system. It is showed that the performance of all applications is considerably improved. Figure 8 presents cache performance of 16-threaded RMS workloads with varying cache line sizes on 16MB cache. All these workloads except FIMI achieve better cache performance when the cache line size is scaled. Particularly, we observe that Ball Detection, PageRank, Face Animation, and ODE almost get linear miss reductions (around 1/3 to 1/4) from 64B to 256B.

Based on the streaming analysis and hardware prefetching as well as large cache line size study, we conclude that these RMS workloads expose good spatial locality and strong streaming behavior.

7. Conclusion

The RMS workloads studied in this work are data and memory intensive, have large working sets, and require large last level cache to accommodate the capacity misses. In addition, these workloads exhibit a tremendous amount of data sharing and as many as 85% of the accesses to the last-level cache are to the shared data. The amount of data-sharing exposed by these workloads is a function of the total cache size available; the larger the data cache the better the sharing behavior. Thus, rather than portioning the last-level cache into multiple private caches, we show that a shared last-level cache is important for improving the performance of these RMS workloads on future high performance platforms. Additionally, these workloads expose significant streaming degree, and they are good target for stride prefetcher and large line size.

References

- [1] GA Abandah, and ES Davidson, "Configuration Independent Analysis for Characterizing Shared-Memory Applications", *Proceedings of the 12th. International Parallel Processing Symposium (IPPS)*, Orlando, Florida, 1998.
- [2] Y. Chen, E. Li, et al, "Media Mining – Emerging Tera-scale Computing Applications", *Intel Technology Journal*, 2007.
- [3] P. Dubey, "Recognition, Mining and Synthesis Moves Computers to the Era of Tera", *Intel Technology Journal*, February 2005.
- [4] J. Hurley, "Ray Tracing Goes Mainstream", *Intel Technology Journal*, Volume 09, Issue 02, May 19, 2005.
- [5] Intel Inc., "Tera-scale Computing Usage Models", http://www.intel.com/research/platform/terascale/ts_usage.htm
- [6] A. Jaleel, "Memory Characterization of Workloads Using Instrumentation-Driven Simulation -- A Pin-based Memory Characterization of the SPEC CPU2000 and SPEC CPU2006 Benchmark Suites", Web Copy: <http://www.glue.umd.edu/~ajaleel/workload/>
- [7] A. Jaleel, R. Cohn, et al. "CMP\$im: Using PIN to Characterize Memory Behavior of Emerging Workloads on CMPs", Technical Report - UMD-SCA-2006-01
- [8] 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", *Proceedings of 12th International Symposium on High Performance Computer Architecture (HPCA)*, Austin TX, February 2006.
- [9] K. Kähler, J. Haber, H. Yamauchi, and HP. Seidel, "Head shop: Generating Animated Head Models with Anatomical Structure", *Proceedings ACM SIGGRAPH Symposium on Computer Animation (SCA)*, 2002, 21-22 July 2002, pp. 55-64.
- [10] Open Dynamic Engine, <http://www.ode.org/>
- [11] P. Lawrence, B. Sergey, "The PageRank Citation Ranking: Bringing Order to the Web", *Stanford Digital Library Technologies Project*, 1998
- [12] Y. Li Y., H. Ai, C. Huang, and S. LAO, "Robust Head Tracking with Particles Based on Multiple Cues Fusion", *HCI/ECCV 2006*, LNCS 3979, pp.29-39, 2006.
- [13] J Liu, X. Tong, W. Li, T. Wang, Y. Zhang, H. Wang, B. Yang, L. Sun, and S. Yang, "Automatic Player Detection, Labeling and Tracking in Broadcast Soccer Video", *British Machine Vision Conference*, Sep, 2007.
- [14] L. Liu, E. Li, Y. Zhang, Z. Tang, "Optimization of Frequent Itemset Mining on Multiple-Core Processor", *33rd International Conference on Very Large Data Bases (VLDB)*, 2007
- [15] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger, "Evaluation Techniques in Storage Hierarchies", *IBM Journal of Research and Development*, 9, 1970.
- [16] T Mohan, BR Supinski, S.A. McKee, F. Mueller, A. Yoo, and M. Schulz. "Identifying and Exploiting Spatial Regularity in Data Memory References", *ACM Supercomputing Conference*, 2003.
- [17] S. Palacharla, R. Kessler, "Evaluating Stream Buffers as A Secondary Cache Replacement", *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, 1994, 24-33
- [18] X. Tong, T. Wang, W. Li, Y. Zhang, "A Three-Level Scheme for Real-time Ball Tracking", *International Workshop on Multimedia Content Analysis and Mining*, China June 2007.
- [19] K. Waters, "A Muscle Model for Animating Three-Dimensional Facial Expression", *IEEE Computer Graphics*, 21(4)
- [20] SC Woo, M. Ohara, E. Torrie, JP Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations", *Proceedings of 22nd Annual International Symposium on Computer Architecture*, 1995, 24-36
- [21] X.Yu, C. Xu, H.W. Leong, Q. Tian, Q. Tang, and K.W. Wan, "Trajectory-based Ball Detection and Tracking with Applications to Semantic Analysis of Broadcast Soccer Video", *Proceedings of ACM Multimedia*, 2-8 November, 2003, pp.11-20

Characterizing Memory Behavior of XML Data Querying on CMP

Hong Liu Rubao Li Qiang Gao Bihui Duan Taoying Liu

Institute of Computing Technology of Chinese Academy of Sciences, Beijing, China

Graduate School of the Chinese Academy of Sciences, Beijing, P.R. China

liuh@ict.ac.cn, {lirubao, gaoqiang, duanbihui, liutaoying}@software.ict.ac.cn

Abstract

The wide spread use of XML storage and web services increases the demand for querying and processing XML data. XQuery is rapidly emerging as the standard for querying XML data. Since XQuery applications usually have large data sets and appear to have little data reuse, they are expected to be memory-intensive and have poor cache performance. Furthermore, the heavy memory traffic may lead to the memory wall problem on multi-core platforms. However, few studies have been done on this topic. In this paper, we measure the memory performance of XQuery workloads on two representative systems with multi-core processors, Intel's Xeon and Sun's T1. Our experiments show that a moderately-sized cache can achieve high hit rate but a large shared last-level cache provides little further help. Furthermore, on the Intel's Xeon server, we measure the effect of hardware prefetching. We find that prefetching does improve throughput but its effect is limited by the memory bandwidth.

1. Introduction

Data management is entering a new era. On the one hand, from the viewpoint of applications, XQuery [1], the standard XML query language, is more and more popular in various data management scenarios. Although the execution of XQuery shares many common concepts with SQL, its memory access pattern is based on path navigation in the Document Object Model (DOM) [2] tree, which is totally different with the iterator-model tuple processing in traditional Database Management Systems (DBMS) [3]. On the other hand, from the viewpoint of the underlying hardware, the integration of multiple cores in a single die with shared last-level cache (LLC) provides new opportunities for multi-threaded applications to utilize constructive data sharing [4] in LLC to hide memory access latencies.

Early studies [5] [6] on characterizing DBMS's memory access behaviors indicate that optimizing L2 cache data placement to hide off-chip access latencies is the key to improve overall performance; But recent research [7] shows that when running concurrent database workloads on Chip Multiprocessors (CMP), the performance bottleneck is the on-chip L2 hit latency since these workloads exhibit significant sharing between cores. However, it is unclear whether the XQuery workload also follows these conclusions considering its unique memory access pattern and unknown working set size.

In our previous paper [8], we examined architectural characteristics of single-threaded XQuery execution on several different CPUs including AMD Sempron and Intel P4. In this paper, we try to answer the following questions by examining concurrent XQuery workloads on multi-core systems:

- What are the memory reference characteristics of XQuery execution?
- How well does the cache hierarchy perform when running concurrent XQuery executions?
- What is the effect of hardware prefetching if there is heavy traffic between on-chip cache and main memory? Does it succeed in decreasing L2 cache misses, or does it just unnecessarily consume bandwidth?

This paper makes the following three contributions:

- We present memory references characteristics of XQuery workloads and compare them with characteristics of SPECint [9] 2006. Compared to the results of SPECint, the XQuery workloads have less data memory references per instruction and higher L1 cache hit rates.
- We measure the memory behavior of concurrent XQuery workloads on two servers based on Intel's Xeon and Sun's T1 respectively. Our results show that both L1 and L2 cache miss rates do not change significantly when the concurrency rises. The large shared

L2 cache does not yield significant benefit on reducing the L2 cache misses.

- We evaluate the impact of Xeon’s hardware prefetching technology on the memory performance of XQuery. Our results show that the combination of the L2 streaming prefetching and the Data Prefetching Logic prefetching has better effect than the Data Prefetching Logic prefetching alone. We conclude that prefetching does improve throughput but its effect is limited by the memory bandwidth.

The rest of this paper is organized as follows. In the following section we provide a background overview of the XQuery and CMP cache hierarchy. In Section 3, we discuss the details of the XQuery workloads used in this paper. Section 4 describes our methodology. Section 5 presents our results and analysis. Section 6 concludes the paper.

2. Background and related works

2.1 XQuery

XQuery is an XML query language based on the XPath [10] data model. In brief, the execution of an XQuery query consists of two stages. First, the target XML file is parsed to a DOM tree or a tree-like object that stays in main memory. Second, corresponding query algorithms are used to fetch necessary nodes in the tree by path navigation and to execute involved calculations on fetched nodes. We summarize the memory characteristics of XQuery applications as follows:

- Memory consuming: Compared to the original XML file, the parsed DOM tree resident in memory is extremely large. The dominant factor for XQuery processing is not the disk accesses like in DBMS but the all-in-memory executions.
- Non-sequential memory access: The path navigation in the DOM tree consists of a series of memory accesses. However, the pattern of such memory access is not sequential as processing successive tuples in a DBMS. Under this situation, it is unclear what kind of locality exists.

2.2 Multi-core cache hierarchy

Compared to traditional symmetric multiprocessors, the multi-core system is able to reduce off-chip communication costs between cores. However, the

tightly integrated design often exerts higher pressure on communication to the off-chip memory. Thus the memory system can be the bottleneck. In general, the cache hierarchy is the most significant solution to this problem.

To efficiently organize the cache hierarchy, one key issue is whether or not to use a shared last-level cache (LLC). Each approach has its pros and cons. The private LLC architecture can reduce access latency and bus contention, while the shared LLC architecture may increase the cache hit rate for some workloads. Since the tradeoff exists, there is a large solution space between the two extreme approaches.

In our test, we selected two servers which have distinct cache designs. One server is Sun UltraSPARC T1000 which has an UltraSPARC T1 CPU with 8 cores, and the other is a Dell 1900 which includes dual Intel Xeon 5310 CPUs (each CPU has 4 cores). In the T1 based system, a high-speed, low-latency crossbar connects and synchronizes all eight on-chip cores and shared L2 cache memory banks. The Intel Xeon’s cache hierarchy is organized in a hybrid model. Two cores share an L2 cache and the Front-Side Bus connects all eight cores in two CPUs. It is unclear which cache hierarchy suits the XQuery applications better.

2.3 Related work

Few prior works specifically studied XML (XQuery) processing from the viewpoint of computer architecture. The paper [11] reported the architectural characteristics of XML processing on an Intel Xeon platform. In [12], the authors in Intel measured the performance of XML data parsing and introduced an accelerating approach. We believe that the information presented in this paper will be useful in understanding XQuery workloads and optimizing XQuery processors on multi-core systems.

3. XQuery workloads design

3.1 Dataset

Our dataset is based on the Digital Bibliography & Library Project (DBLP) [13] xml file. This file includes bibliographic information on the major journals and proceedings of various conferences about computer science. The file size is growing and was about 409MB in October, 2007.

Figure 1 shows the file’s structure. Under the root node, there is a collection of child nodes, each of which represents a category of articles, books or papers,

etc. The leaf nodes contain elements such as years, authors, and other metadata information.

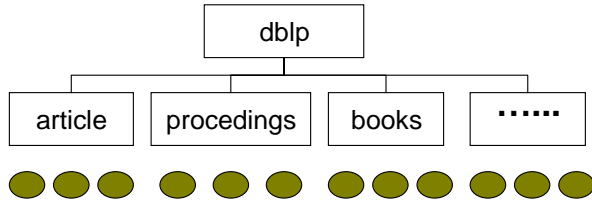


Figure 1: DBLP dataset's structure

To avoid the impact of page swapping in our experiments, we define two subsets of data whose sizes are 39MB and 73MB separately. This is achieved by only selecting top conference papers. The test results on these two datasets are similar. However, on a server with 4GB main memory, we can only run four concurrent XQuery processing threads on the 73MB dataset without any page swapping. Therefore we choose the 39MB subset in this paper. The size is big enough to reflect some architectural metrics while is small enough to let at least eight concurrent XQuery processing threads be resident in the 4GB main memory.

3.2 Queries

We built an XQuery-based web server for querying the DBLP data set and put it online for about half a year. Referring to the trace of our website, we choose eighteen queries which can be classified into three categories:

- 1) Query for exact matching: These queries need exact string matching with specified path expressions (Q1 - Q4).
- 2) Query for aggregation: These queries have aggregation operations in their bodies. (Q5 - Q12).
- 3) Query for sorting. These queries require their result must be ordered. (Q13 - Q18).

Table1. Selected queries of the workload

Q1	Lists all articles written by a specified author within a certain time period
Q5	Returns the number of the articles in a specified conference's proceedings within a certain time period
Q13	Lists all articles that contain a specified key word, sorted by date

In table 1, we list three typical queries, each of which represents a category.

4. Methodology

This section describes the software tools and hardware platforms used to measure the memory performance of XQuery workloads.

4.1 Software

Our experiment program is the Nux [14] XQuery processor. It uses Xerces [15] to build DOM trees and Saxon [16] to execute the query. The Java VM used in our experiments is the jre1.6.0.

We use the VTune [17] performance analyzer and Solaris's [18] cputrack to monitor the architectural behavior on the X86 architecture and SPARC processors respectively. We calculate the following metrics:

- L1 Data cache miss per instruction
- L2 Data cache miss per instruction
- Cycles per instruction (CPI)
- Memory bandwidth

Table 2 lists these performance metrics and the way to calculate them.

4.2 Hardware

Our experiments are mainly performed on two servers in our evaluation. One is a DELL 1900 server that consists of two Intel Xeon E5310 processors and 4096MB DDR2 667 RAM. Its operating system is the RedHat Enterprise Linux server 5; the kernel version is 2.6.18. The other is a Sun Fire T1000 Server with 8 cores and 8192MB DDR2 800 RAM. Its operating system is Solaris 10. Table 3 gives the detailed information of the CPUs on these servers.

We also used a server with dual Opteron 248 processors. The Opteron 248 has 64KB L1 data cache and 1MB L2 cache. The operating system on this server is also the RedHat Enterprise Linux server 5.

Table 2. Performance metrics

	Intel Xeon E5310	Ultra-SPARC T1
L1 Dcache Miss per Instruction	L1 Data Lines Allocated / Instructions Retired	DC_miss/Instr_cnt
L2 Miss per Instruction Ratio	L2 Lines Allocated/ Instructions Retired	L2_dmiss_ld/Instr_cnt
CPI	CPU_CYCLES/ Instructions Retired	Cycle_cnt/Instr_cnt
Bus utilization	BUS_TRANS_ANY * 2 /CPU_CLK_UNHALTED	Do not know

Table 3. Detail information of CPUs

Characteristic	Intel Xeon E5310	Sun Ultra-SPARC T1
L1 cache organization	Split instruction/data caches	Split instruction/data caches
L1 cache size per core	32KB instruction cache, 32KB data cache	16KB instruction cache, 8KB data cache
L1 cache associative	2-way associative	4-way set associative
L1 block size	32 bytes	32 bytes(instruction cache), 16 bytes(data cache)
L2 cache organization	Unified(instruction and data)	Unified(instruction and data)
L2 cache size	4MB × 2 (two cores share one L2 cache)	3MB (shared by cores)
L2 cache associative	16-way set associative	12-way set associative
L2 block size	64 bytes	64 bytes

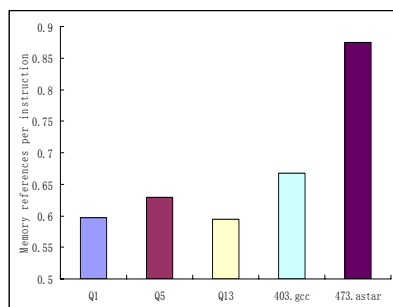


Figure 2. Data memory references per instruction

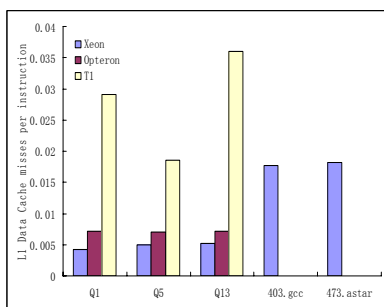


Figure 3. L1 cache misses comparison

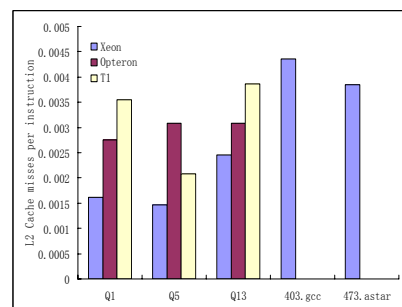


Figure 4. L2 cache misses comparison

5 Results and analysis

5.1 Memory references characteristics

First, because the XQuery program has a very large data set, we expect that it is memory-intensive. We choose the data memory reference per instruction as a metric to measure whether it is memory-intensive. A higher value indicates the higher demand on the memory subsystem (in general, including the cache subsystem).

We measure the data memory reference per instruction of three typical queries, Q1, Q5 and Q13 separately. We also measure the value of the other 15 queries and the results are similar. Then we compare the results with the results of two SPECint 2006

programs: 403.gcc and 473.astar, whose results are in a middle position among all SPECint programs.

Figure 2 shows that the data memory reference per instruction of XQuery is even less than the value of two SPECint 2006 programs. It means that the XQuery program is not such a memory-intensive program as we expect. Another important question is how well the cache hierarchy performs. We compare the cache miss per instruction of XQuery programs with two programs of SPECint 2006. Although we also measure other data such as instruction cache misses and TLB misses, these metrics do not show significant difference between XQuery and SPECint.

In order to show the cache performance under different cache configurations, we perform XQuery workloads on systems with T1, Xeon or Opteron. Our results show that the XQuery workloads' L1 cache

miss per instruction values vary wildly and the L2 cache miss per instruction values are almost the same. Compared to the SPECint programs running on Xeon, the L1 cache miss rate is much lower.

We find that the XQuery execution has the following features:

- Frequent data reuse in a medium working set. The XQuery algorithm is a typical divide-and-conquer algorithm and performs a considerable amount of computation on sub-trees. In DBLP, most sub-trees include less than 1000 nodes and consequently their sizes are less than 100KB.
- Little data reuse across the medium working sets. Once the operations on a sub-tree are finished, the sub-tree's data is seldom reused.

These features make the XQuery's working set fit in a moderately-size cache. Therefore the Xeon's 32KB L1 cache can achieve high hit rate but its 4M L2 cache is less effective. Moreover, T1's 8KB L1 cache is somewhat small in our experiments.

5.2 Fundamental performance metrics of concurrent querying

In this section, we measure the fundamental performance metrics of concurrent query executions. Our test program starts multiple threads and each thread executes the 18 queries in different order by random permutation. These threads share a single parsed DOM tree and the DOM tree parsing process is not included in our experiments.

Because T1 has cut off some performance counters, we do not know the way to calculate the bus utilization

on it. However, using Solaris's [18] busstat, we can estimate that the T1 based system is roughly transferring about 4 Gigabytes per second on bus when there are 32 threads. The T1's specification claims that the four on-chip memory controllers can support over 20 Gigabytes per second memory bandwidth, so we believe that T1's memory system is far from saturation.

The Figures 5, 6 and 7 show that as the thread number increases the Xeon based system's performance rolls off. At the same time, the bus utilization is above normal (40%) and CPI also increases rapidly. The T1 based system has nearly linear speedup.

5.3 Cache hierarchy's performance of concurrent querying

The results of fundamental performance metrics indicate that the Xeon based system may encounter a typical trouble caused by the Memory Wall. Then, what is the role of the memory hierarchy? We present the measurement of L1 data cache misses and L2 cache misses below. Figure 8 and Figure 9 show the L1 and L2 cache misses per instruction of Xeon and T1. Figure 10 shows the local miss rate of L2 cache that indicates the effect of L2 cache. The results show that on Xeon, the L2 cache itself has poor hit rate. In addition, not only on Xeon, whose cache is shared by two threads, but also on T1, whose cache is shared by up to 32 threads, the shared L2 cache does not yield significant benefit on the L2 cache's hit rate

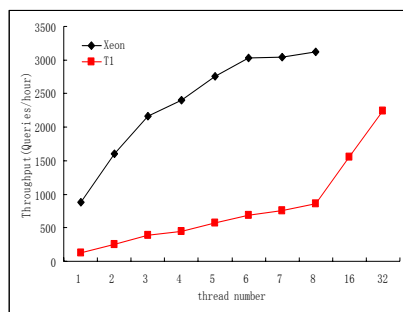


Figure 5. Throughput on T1 and Xeon

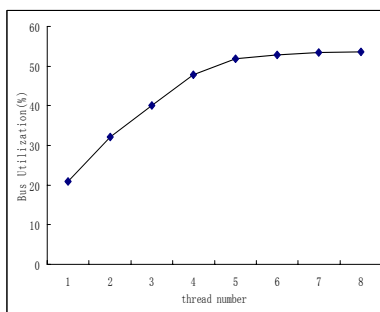


Figure 6. Bus utilization of Xeon

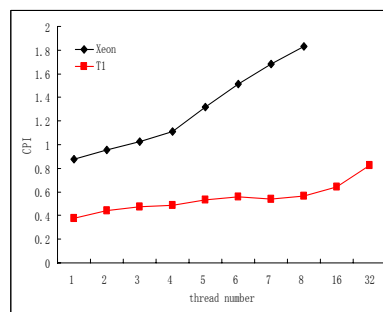


Figure 7. CPI of T1 and Xeon

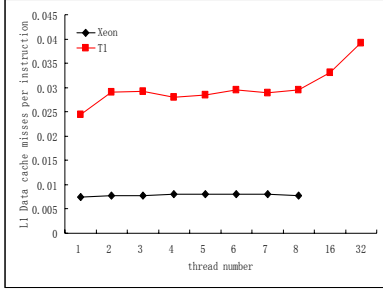


Figure 8. L1 data cache misses per instruction of T1 and Xeon

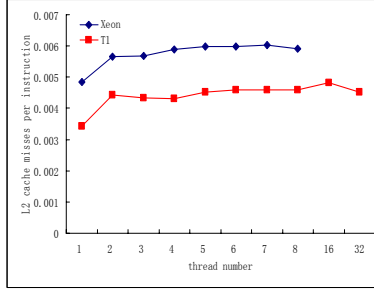


Figure 9. L2 cache misses per instruction of T1 and Xeon

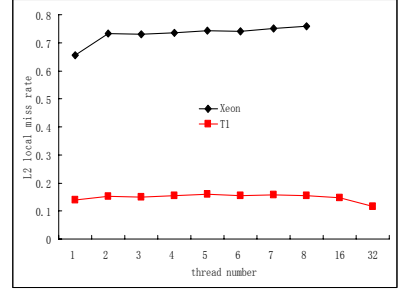


Figure 10. L2 local cache miss rate

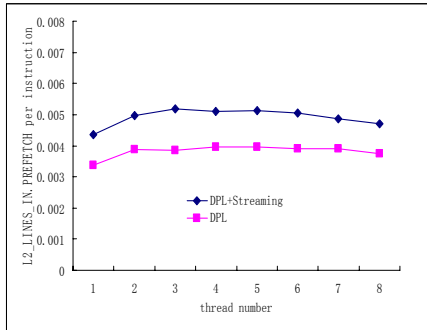
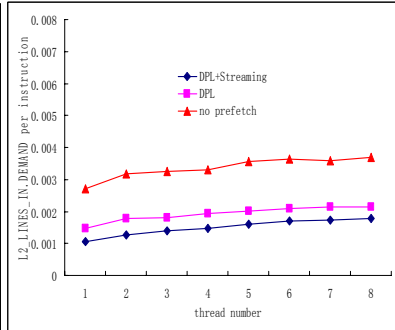
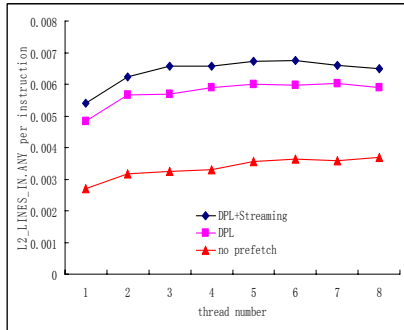


Figure 11. Comparison L2 misses

5.4 Effect of hardware prefetching

Hardware prefetching is an effective approach to reducing the cache miss rate. The Intel Core architecture provides two prefetch mechanisms, namely the Data Prefetch Logic (DPL) and L2 Streaming Prefetch. The DPL prefetch mechanism fetches streams of instructions and data from memory to the L2 cache when a stride in the memory access pattern is detected. L2 Streaming Prefetcher fetches the adjacent 64-byte cache line for each memory access. By default, the DPL prefetch mechanism is enabled and L2 Streaming Prefetcher is disabled. We evaluate the following three configurations:

- Disabling both prefetching
- Enabling DPL (default)
- Enabling both prefetching

Figure 12 shows that both prefetching mechanisms together can increase the system's throughput by 50% when the bus utilization is below a threshold (referring to Figure 6, about 40%). However, as the thread

number increases, its advantage diminishes. As such, for eight threads, the configuration that disables all prefetching has the best throughput.

The reasons can be found in Figure 11. L2_LINES_IN event is used to count the number of cache lines allocated in the L2 cache. The left picture of Figure 11 presents all L2_LINES_IN events number per instruction; the middle picture shows the demand L2_LINES_IN events number per instruction and the right one demonstrates the prefetched L2_LINES_IN events number per instruction. We find that when both DPL prefetching and L2 streaming prefetching are enabled, the demand request's number decreases. However, the number of cache lines allocated in the L2 cache is twice as much as it is on the configuration that disables all prefetching. Moreover, when the bus is busy, the prefetching will reduce the throughput. In addition, Figure 12 indicates that the combination of both types of prefetching produces a better result than the DPL prefetching alone.

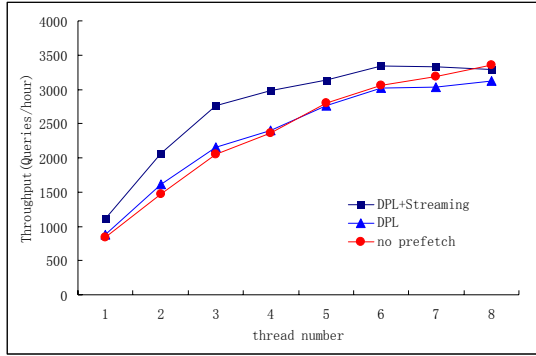


Figure 12. Comparison of throughput

5. Conclusions and future work

This paper presents a detailed characterization of the memory behavior of XQuery workloads on different multi-core platforms. Out of our expectation, a moderately-size cache can achieve high hit rate but a large shared L2 cache can not yield considerable benefit. Furthermore, both L1 and L2 cache miss rate only vary slightly when the concurrency rises. Our experiments also show that although L2 data prefetching does improve throughput, its effect is limited by memory bandwidth on highly-concurrent workloads. Our future work is to develop shared-cache-optimized XQuery processors. We think a staged query engine design [19] could be an attractive solution for exploiting a shared last level cache.

Acknowledgement

We sincerely thank chairs of this workshop, Russell Clapp and Adrian Moga, for their invaluable comments on this paper. We corrected many mistakes in this paper with their help. This work is supported in part by the National Natural Science Foundation of China (Grant No. 60573102, 60403023, 90412010, 90612019), the 973 Program (Grant No. 2005CB321800). The authors are very grateful for

these supports. We would like to thank Wang Hao for his suggestions, ideas and enthusiastic support.

References

- [1] XQuery: <http://www.w3.org/TR/xquery/>
- [2] DOM: <http://www.w3.org/DOM/>
- [3] G. Graefe. "Query Evaluation Techniques for Large Databases." *ACM Computing Surveys* 25(2), 1993.
- [4] L. Spracklen and S. G. Abraham. "Chip Multithreading: Opportunities and Challenges." In *Proc. HPCA*, 2005.
- [5] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. "Contrasting Characteristics and Cache Performance of Technical and Multi-user Commercial Workloads." In *Proc. ASPLOS*, 1994.
- [6] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. "DBMSs on a Modern Processor: Where Does Time Go?" In *Proc. VLDB*, 1999.
- [7] M. HardAvellas, I. Pandis, R. Johnson, N. G. Mancheril, A. Ailamaki, and B. Falsafi. "Database Servers on Chip Multiprocessor: Limitations and Opportunities." In *Proc. CIDR*, 2007.
- [8] R. Lee, B. Duan, and T. Liu. "Architectural Characterization of XQuery Workloads on Modern Processors." In *Proc. ExpDB*, 2007.
- [9] SPECint 2006: <http://www.spec.org/>
- [10] XPath: <http://www.w3.org/TR/xpath>
- [11] P. Apparao, R. Iyer, R. Morin, N. Nayak, and M. Bhat. "Architectural Characterization of an XML-centric Commercial Server Workload." In *Proc. ICPP*, 2004.
- [12] L. Zhao and L. Bhuyan. "Performance Evaluation and Acceleration for XML Data Parsing." In *Proc. CAECW*, 2006.
- [13] DBLP: <http://www.informatik.uni-trier.de/~ley/db/>
- [14] Nux: <http://dsd.lbl.gov/nux/>
- [15] Xerces: <http://xerces.apache.org/>
- [16] Saxon: <http://saxon.sourceforge.net>
- [17] Vtune: <http://www.intel.com/cd/software/products/asmona/eng/vtune/>
- [18] Solaris: <http://www.sun.com/software/solaris/>
- [19] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. "QPipe: A Simultaneously Pipelined Relational Query Engine." In *Proc. SIGMOD*, 2005

Memory Characterization of SPEC CPU2006 Benchmark Suite

Junmin Lin¹ Yu Chen¹ Wenlong Li² Zhizhong Tang^{1*} Aamer Jaleel³
¹Department of Computer Science & Technology, ²Intel China Research Center, ³Intel Corporation, VSSAD
Tsinghua University, Beijing, China Beijing, China Hudson, MA
linjunmin@gmail.com wenlong.li@intel.com aamer.jaleel@intel.com

Abstract

SPEC CPU2006 benchmark suite has been released to measure the performance across various computer systems. Understanding the memory requirements of workloads from the SPEC CPU2006 suite is essential in understanding the performance results and evaluating the efficiency of the memory system. This paper conducts a detailed memory characterization of the CPU2006 suite via full run simulation. We study the working set size and streaming behavior, which show the temporal and spatial locality respectively. In general, SPEC CPU2006 comprises of many memory-intensive workloads that require more than 4MB of cache size for better cache performance. Cache size sensitivity study indicates many workloads have poor temporal locality, and large DRAM cache may serve as a good candidate. Meanwhile, a large portion of workloads show large amount of streaming behavior, indicating good spatial locality. Large cache lines or prefetching can utilize this streaming behavior to improve the memory performance.

1. Introduction

The SPEC CPU benchmarks are widely used for performance measurement across various computer systems in both industry and academia. The latest released SPEC CPU2006 [1][7] suite is much more compute-intensive than the prior suite, and continues to exert pressure on the memory system. Understanding the memory requirements of these workloads is essential in evaluating the performance of a computer's memory system. Gove studied the memory footprint of CPU2006 suite [6] and compared CPU2006 to CPU2000. Sair et al. studied the memory behavior of SPEC CPU2000 suite [14]. Earlier versions of SPEC were evaluated in [3][5].

This paper studies the memory behavior of the full-run applications of SPEC CPU2006 suite with binary instru-

mentation-driven simulation, in terms of working set size and streaming behavior. Our study makes the following contributions:

- A detailed cache sensitivity study (by varying the cache size from 64KB to 128MB) reveals that most of them require more than 4MB of cache size to fit, and in particular, 11 workloads have working set size exceeding 128MB. Such large working set sizes indicates the need for large caches. Since large SRAM based cache organizations can be expensive to build, alternative cache organizations using DRAM (e.g. embedded DRAM, off-die DRAM caches, or 3D die-stacking) can be useful to reduce the latency and bandwidth to main memory.
- A streaming behavior study reveals that 21 of the 35 integer workloads and 14 of 21 floating point workloads show strong streaming behavior, and among them, more than 50% memory accesses form streams. This indicates that these workloads have good spatial locality and highly predictable access patterns. Therefore, both large cache lines and hardware stride prefetchers should provide better cache performance for these workloads.

2. Characteristics and Methodology

2.1. Characteristics

We characterize the memory behavior of SPEC CPU2006 benchmarks by focusing on the following two characteristics:

- (i) **Working Set Size:** The working set [4] size of a workload indicates its temporal locality. Whether an important working set fit in the cache or not can have a significant impact on the cache performance. By evaluating the cache performance versus cache size, we can understand how large the cache should be designed to hold the total working set.
- (ii) **Streaming Behavior:** Streaming access pattern indicates the spatial locality of a workload. This is an important property when designing the memory subsystem. For example, for a workload with significant

* Supported in part by the National Natural Science Foundation of China under Grant No. 60573100,60773149

streaming behavior, hardware prefetcher can be expected to work well. For such case, larger cache line can also provide better cache performance by reducing compulsory misses. In addition, we found that 32 stream detection engines can capture most streams from these workloads.

2.2. Experimental Methodology

In this paper, we use instrumentation driven simulation (IDS) [10][15] to conduct memory system studies, for its advantages of fast, robust, simple and supporting full run of applications. Specifically, we use CMPsim [8], an instrumentation driven memory system simulator based on Pin[10]. All workloads from the SPEC CPU2006 suite are run to completion using their reference input sets. We also collect CPI numbers using performance counters on an Intel® Xeon® 3.33 GHz system with 32KB L1 data cache and 1MB L2 unified cache. All workloads are compiled with -O3 optimization level on a Red Hat 32-bit Linux system using Intel’s C/C++/Fortran compilers.

For each workload, we conduct a cache sensitivity

analysis using the stack distance [11] approach to simulate multiple cache sizes in one run. We model a 2048-way 128MB data cache. Thus the simulated cache sizes are direct mapped 64KB, 2-way 128KB, 4-way 256KB, and so on. For our caches, we use 64B line size and LRU replacement policy.

For streaming behavior, we implement the same mechanism as that in [13] to detect streams with unit and non-unit strides. We use 32 stream detection engines to collect the streaming characteristics on the missed L1 data cache requests. That means our stream detection engine lies in between L1 and L2, and it detects streams on the L2 cache requests (the data locality has been filtered by the L1 cache, and the minimum stride value is one cache line size, i.e., 64 bytes). As there is no PC information for the missed L1 data cache request, we implement the address based stream detector. To simulate hardware prefetching in most commercial processors, when reaching page boundary, we stop streams at page boundary, and re-train them in the next memory page.

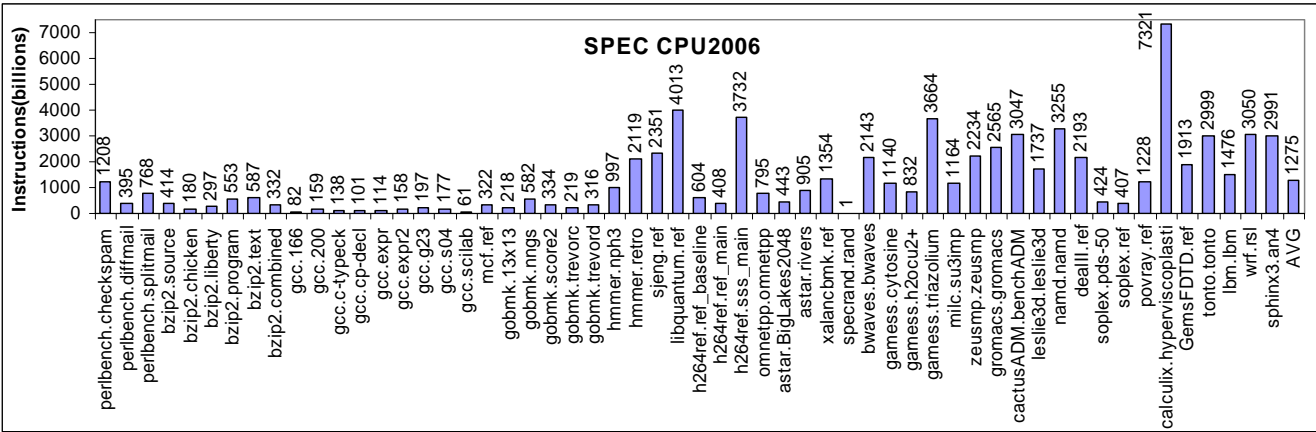


Figure 1: Instruction Count of SPEC CPU2006 Workloads.

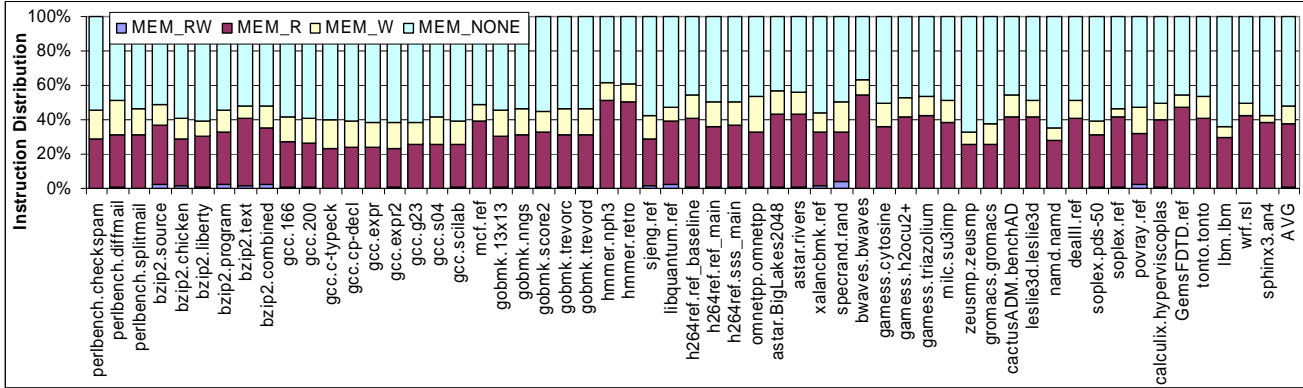


Figure 2: Instruction Profile of SPEC CPU2006 Workloads. The figure shows the instruction count and instruction mix of the applications. MEM_NONE: instructions that do not reference memory (ALU operations only); MEM_R: instructions that have one or more source operands in memory; MEM_W: instructions whose destination operand is in memory; MEM_RW: instructions whose source and destination operands are in memory.

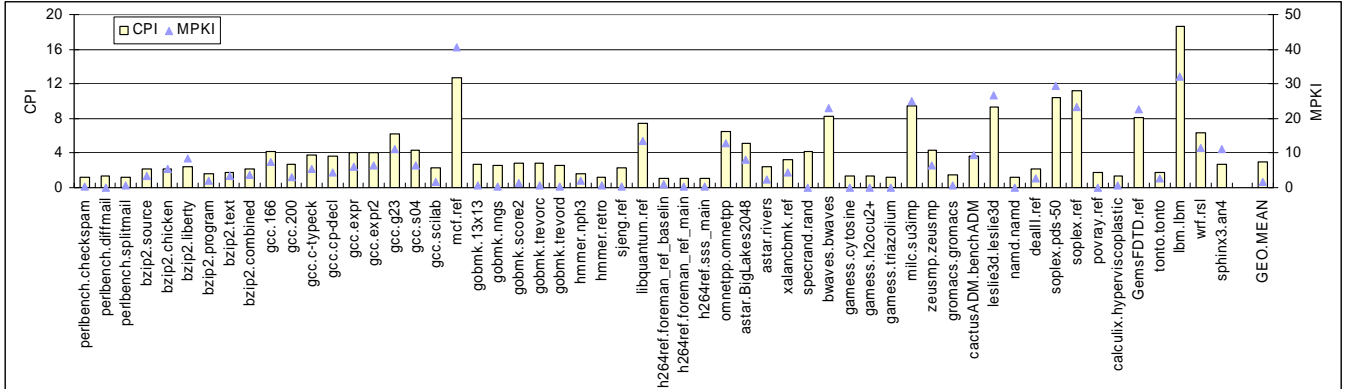


Figure 3: Performance of SPEC CPU2006 Workloads. The figure shows the CPI values gathered using performance counters on a 3.33GHz Xeon system.

3. Overview of SPEC CPU2006

The SPEC CPU2006 suite consists of 12 integer programs and 18 floating point programs. With all reference input sets, the number of corresponding workloads is 35 and 21, respectively. Compared to earlier versions, it is more compute-intensive and time consuming. Most of the workloads are also memory intensive, continuing to stress the memory hierarchy. In this section, we present basic characteristics of the SPEC CPU2006 benchmarks, including application instruction profile and the overall performance.

3.1. Application Instruction Profile

The average instruction count for the SPEC CPU2006 benchmark suite is 1276 billion. Figure 1 shows the total instruction count for each workload in the suite. For applications with large run lengths, full-run simulation is more valuable than only simulating small regions of applications, as it supports varying phase behavior and helps choosing representative regions for detailed simulation. However, it presents a challenge to simulation speeds. Since IDS methodology allows for quick full-run exploratory studies of applications, it is adopted in the following study.

To understand the contribution of instructions that reference memory, as Figure 2 shows, we classify the instructions into four categories: MEM none, MEM read, MEM write, MEM read and write. On average, roughly half of the instructions reference memory. Of the instructions that reference memory, very few (< 1%) both read from and write to memory while 20% of total memory instructions write to memory.

3.2. Processor Performance

As shown in Figure 3, the average CPI of the SPEC CPU2006 workloads is 2.94. By the CPI values, the primary memory bound workloads (with CPIs greater than 5)

are identified as follows: *gcc* (*g23* input), *mcf*, *libquantum*, *omnetpp*, *astar* (*BigLakes2048* input), *bwaves*, *milc*, *leslie3d*, *soplex*, *GemsFDTD*, *lbm* and *wrf*. We also present the misses per one thousand instructions (MPKI) collected on the target machine in Figure 3. We observe that applications with high CPI values have high last level cache misses. For example, The CPI is 18.6 for *lbm*, 12.7 for *mcf*, while the MPKI reaches 32.1 and 40.5 respectively. We further examine the relationship between the CPI and MPKI values, and find that the correlation coefficient is as high as 0.918. This implies that the memory system requirements of the workloads may have significant impact on the overall performance. To understand the memory system requirements of these workloads, the next section presents workload sensitivity to different cache sizes.

4. Working Set

We perform working set analysis of the workloads by evaluating their cache performance with different cache sizes. The cache misses per thousand instructions versus cache size curve can provide us with insights into how much temporal locality each program has, as well as how effective caches will be at reducing bandwidth usage to the next level of the memory hierarchy. Typically, applications have multiple working sets because different data structures in the application have different temporal locality.

4.1. Working Set Size

First, we show that SPEC CPU2006 workloads have very different working set size, varying from 256KB to more than 128MB. Figure 4 presents the cache sensitivity studies with cache size in megabytes (MB) on a logarithmic x-axis and misses per one thousand (MPKI) instructions on the y-axis. Note that the y-axis varies significantly between different workloads.

We observe that 11 of the 56 SPEC CPU2006 workloads have total memory footprint greater than 128MB.

With the stack distance approach, we can not only obtain the cache performance for multiple cache sizes but also know the total memory footprint of each workload. For example, if the workload has a total data footprint that is smaller than the simulated large cache, the amount of valid data in the cache (assuming the cache is invalid at simulation start) represents the total memory footprint of the workload. This information is represented on the cache sensitivity graphs for workloads where the last x-axis co-ordinate ends before the simulated large cache size (128MB in our experiment). For example, the last x-coordinate data cache point for *sphinx3* occurs at 8MB, implying that it has a total data memory footprint size of approximately 8MB. However, for workloads such as *bwaves*, *milc* where misses occur even in a 128MB cache, the footprint size of these workloads is greater than or equal to 128MB.

The data cache sensitivity study also reveals that different workloads may have totally different cache friendly behavior—consistent friendly or sometimes unfriendly. As shown in Figure 4, most of the workloads exhibit cache friendly behavior—incremental increases in cache size yields incremental improvements in cache performance. Such behavior is observed when the cache miss rate shows a smooth decreasing behavior with increasing cache sizes (e.g. *bzip2*). However, there are some workloads where incrementally increasing the cache size provides no improvements in cache performance until a particular cache size is reached. At this cache size suddenly significant improvement in cache performance is observed, e.g., *libquantum* and *sphinx3*. This behavior implies that the respective workloads have a working set that is of the same cache size as where the drops occur. Some workloads like *gcc* illustrate multiple drops in the cache miss rate function implying that they have multiple working set sizes.

Since the SPEC CPU2006 workloads operate on a very large working set, the future memory systems are expected to provide high bandwidth and low latency to achieve high performance. The large DRAM cache is a promising technique to achieve this goal. The DRAM cache can be enabled either through 3D stacking or by the use of a multiple-chip package, and its benefits are two-fold: (a) it reduces the bandwidth requirement on external memory and (b) the latency to and from the DRAM cache is only a fraction of that of external memory. Of course, the benefits of the DRAM cache are largely a function of the miss rate that it provides. When increasing the cache size from 1MB to 128 MB, the average cache miss per 1000 instructions (MPKI) drops significantly. Among these workloads, the

MPKI drops from 10.1 to 0.19 for *gcc.g23*, 46.3 to 12.5 for *mcf*, 13.6 to nearly zero for *libquantum*, 10.9 to nearly zero for *omnetpp*, 8.35 to 0.01 *astar* (BigLakes2048 input), 17.1 to 12.8 for *GemsFDTD*. Reduction in MPKI can be translated into reduced bandwidth demands beyond the last level cache (LLC). Based on the data for *mcf*, there are nearly 4-fold decreases when we use a large DRAM cache.

4.2. Sensitivity to Different Input Sets

Different input sets may have various impacts on the working sets of an application, as shown in Figure 4. Input sets that have the highest MPKI are considered more memory intensive than others. For example, for the *astar* benchmark, the *BigLakes2048* input is more data memory intensive of the two. Since some input sets may have significant impact on the working sets of an application, we should choose the representative input sets carefully for memory behavior study.

5. Streaming Behavior

In this section, we characterize the spatial locality of CPU2006 workloads by examining their streaming behavior.

5.1. Basic Characterization

In this work, we define *Stream* as a sequence of memory accesses with a constant non-zero difference between any two adjacent addresses in the sequence [12][13], namely *strided accesses*. Stream is one important metric to represent memory access characteristics, and many memory subsystem optimizations have been proposed to target such kind of memory accesses pattern, such as hardware prefetching [2], streaming buffer [13], and profile-driven optimizations [12], etc.

Figure 5 (a) and (b) show the access distribution on stream length and stream stride. The *non-streaming* means the memory access doesn't belong to any streams. The *training* represents the memory access that recognizes a stream. The rest represents the memory access that hits a steady stream. In our implementation, unit stride stream requires one element for training, while non-unit stride stream needs two elements for recognizing a stream. As we don't detect streams across page boundary, the maximum stream length is 64 (the size of page divided by stream stride, i.e., 4KB / 64B).

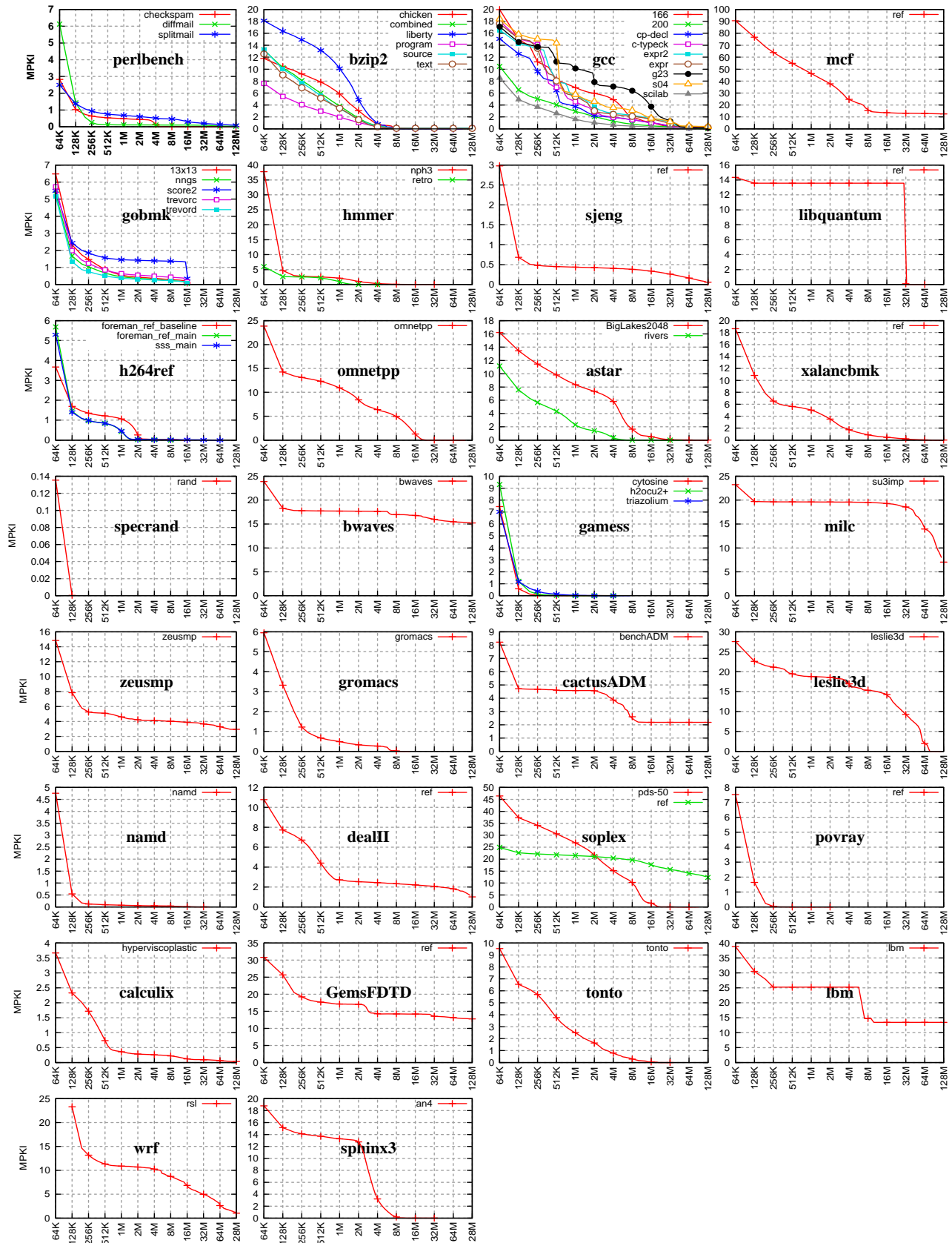


Figure 4: Cache Performance of SPEC CPU2006 Workloads as a Function of Cache Size

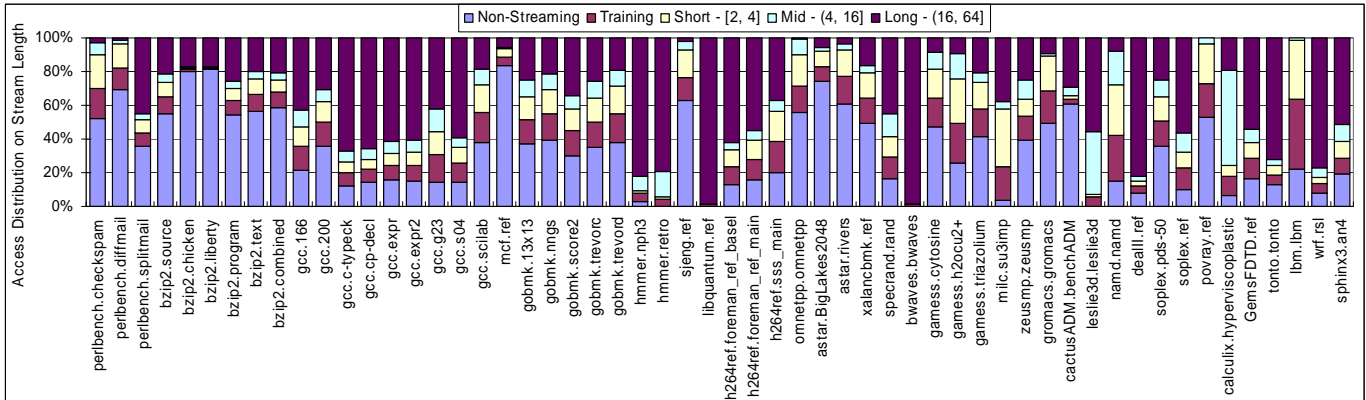
All workloads fall into three categories according to their streaming degree exposed in Figure 5 (a): (1) strong streaming – more than 60% memory accesses are to long streams (streams with length larger than 16), (2) moderate streaming – more than 50% memory accesses are to streams, but the accesses to long streams are moderate, and (3) more than 50% memory accesses are non-streaming. The first two categories may benefit from hardware prefetching, while the third category may not. From Figure 5 (a), seven integer workloads (*gcc.c-typeck*, *gcc.cp-decl*, *gcc.expr*, *gcc.expr2*, *hmmmer.nph3*, *hmmmer.retro* and *libquantum.ref*) and four floating-point workloads (*bwaves.bwaves*, *deall.ref*, *tonto.tonto* and *wrf.rsl*) are in the first category. 14 integer workloads and 10 floating point workloads are in the second category. These two classes of workloads expose significant streaming behavior, with more than 50% memory accesses in streams. Finally, as many as 21 workloads are in the third category. In summary, the SPEC CPU2006 programs exhibit highly predictive memory access patterns, and they are good candidates for prefetchers that can detect these patterns accurately.

Most streams of the SPEC CPU2006 workloads have unit stride value as shown in Figure 5 (b). The dominant unit-stride streaming memory access pattern indicates that

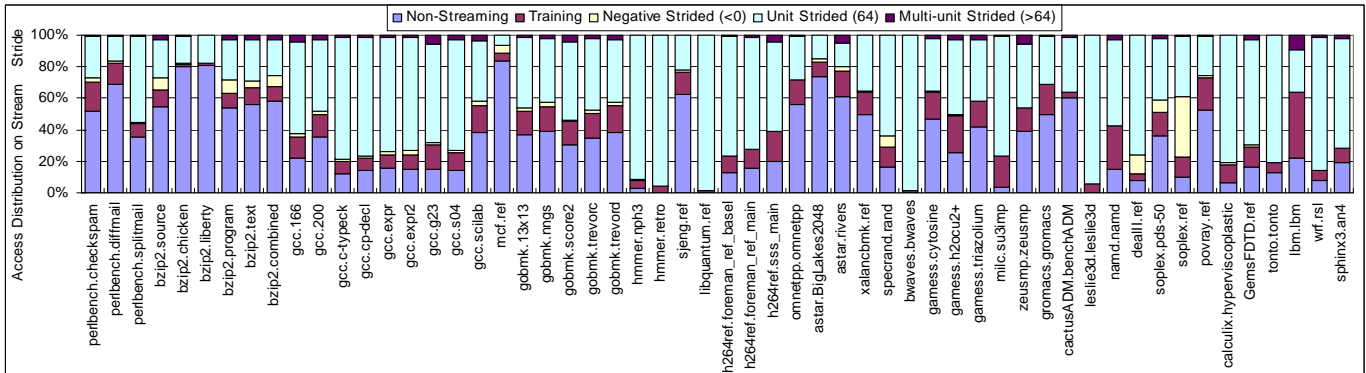
these workloads have good spatial locality, thus larger cache line could provide better cache performance by reducing the compulsory misses. To know the potential impact of streaming behavior on memory subsystem design, we evaluate the performance of large cache line size and hardware prefetching in the following subsections.

5.2. Varying Cache Line Size

Figure 6 presents cache performance of workloads with varying cache line size on 4 MB simulated cache. All workloads except *perlbench.checkspam* and *astar.rivers* achieve better cache performance when the cache line size is scaled. We can see that for workloads exposing good streaming behavior with dominant unit-strides and having large working sets, the cache performance is greatly improved with large cache line size. For example, in the strong streaming category specified in Section 5.1, eight workloads (*gcc.c-typeck*, *gcc.cp-decl*, *gcc.expr*, *gcc.expr2*, *libquantum.ref*, *bwaves.bwaves*, *deall.ref*, and *wrf.rsl*) have working set size larger than 8MB. Their cache misses are all reduced with 128B and 256B cache line sizes. On the other hand, there are a few workloads that have large working sets but gain little (even degrade) cache performance improvement from large cache line sizes, such as



(a) Access Distribution on Stream Length



(b) Access Distribution on Stream Stride

Figure 5: Streaming Behavior of SPEC CPU2006 Workloads. *stream length*: the number of elements in a stream. *stream stride*: the constant distance in memory address between any two adjacent elements of a stream.

sjeng.ref, *astar.rivers*, *astar.BigLakes2048*, and *omnetpp.omnetpp*. We can see from Figure 5 (b) that these workloads have high portion of non-streaming access and insignificant long-streaming accesses, thus increasing cache line size may increase conflict misses. The penalties in these workloads outweigh the benefits from the limited spatial locality and worsen the cache performance. In general, a cache with scaled line sizes could utilize the large portion of unit-stride streaming access pattern in SPEC CPU2006 to provide better cache performance.

5.3. Prefetching

Figure 7 presents the performance benefit from hardware prefetching measured on a real Xeon system. We can see that for workloads which expose good streaming behavior and have large working sets, hardware prefetching can improve the performance significantly. For example, eight workloads in the strong streaming category specified in Section 5.1, receive performance gains ranging from 34% to 179%. On the other side, there are a few workloads that have large working set size but hardly gain any performance improvement from hardware prefetching, in-

cluding *sjeng.ref*, *omnetpp.omnetpp*, *astar.BigLakes2048*, and *astar.river*. Due to the poor streaming behaviors of these workloads, the on chip hardware prefetchers could catch little predictable access pattern on them. In general, we observe an average of 28.8% performance improvement on SPEC CPU2006 workloads, indicating that the high portion of streaming access pattern exhibited from these workloads can be well utilized by hardware prefetcher to improve the memory performance.

5.4. Varying Stream Detection Engine Number

To understand how the identified streaming behavior is sensitive to the number of stream detection engines on SPEC2006 benchmarks, we also present data for 8, 16 and 1024 stream detection engines respectively. As shown in Figure 8, most streaming characteristics captured by 32 stream detection engines can also be recognized with 8 and 16 engines, except for *catusADM.benchADM* and *lbm.lbm*. These two programs have too many concurrent short streams, requiring a large stream table. In general, 32 stream detection engines are good enough to capture most

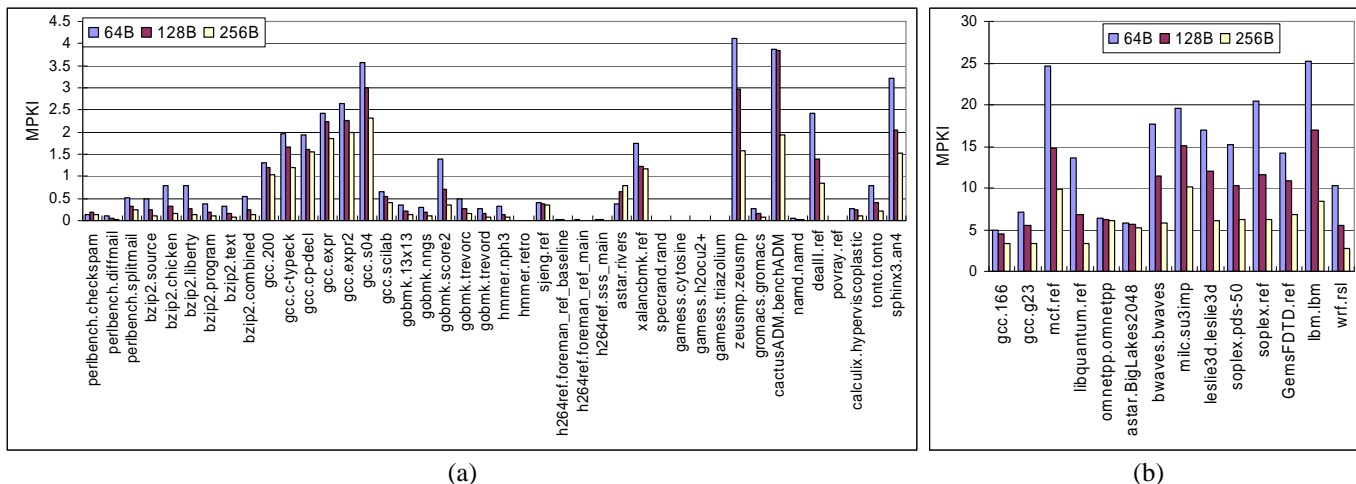


Figure 6: Performance of 4MB L2 with Different Cache Line Sizes for SPEC CPU2006 Workloads. For better readability, we draw two graphs with different scales.

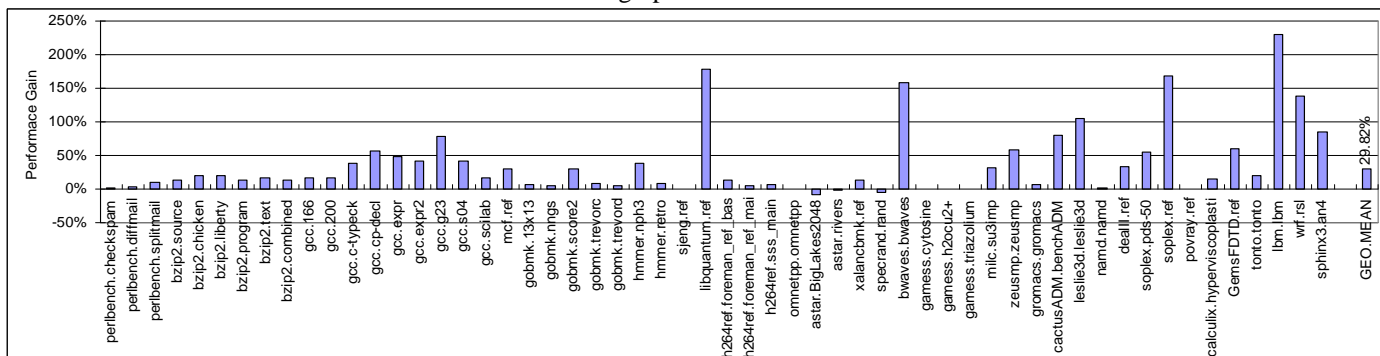


Figure 7: Performance Gain from Hardware Prefetching. The performance is measured in terms of CPI on a 3.33GHz Xeon system.

of the streaming behavior for SPEC CPU2006 workloads at a modest hardware cost.

6. Conclusion

This paper presents the memory system requirements of SPEC CPU2006 benchmarks suite. We show that most of the SPEC CPU2006 workloads are memory intensive and have large working sets. Large DRAM cache can be a good candidate to provide better cache performance. In addition, a large portion of workloads exhibit significant streaming behavior, indicating good spatial locality and highly predictable access pattern. We further examine the effect of large cache line and hardware prefetching, and demonstrate that they can utilize the spatial locality to improve the performance of memory subsystem.

7. References

[1] SPEC CPU2006: <http://www.spec.org/cpu2006/Docs/readme1st.html>

[2] GA Abandah, and ES Davidson, "Configuration Independent Analysis for Characterizing Shared-Memory Applications," in Proceedings of the 12th. International Parallel Processing Symposium (IPPS), Orlando, Florida, 1998.

[3] M. Charney and T. Puzak, "Prefetching and Memory System Behavior of the SPEC95 Benchmark Suite", IBM J. Research and Development, vol. 41, no. 3, pp. 265-286, May 1997.

[4] Peter J. Denning, "The working set model for program behavior". Communications of the ACM, 5/1968, Volume 11, pp. 323-333.

[5] Jeffrey Gee, Mark Hill, Alan J Smith, "Cache Performance of the SPEC Benchmark Suite", University of California at Berkeley, Technical Report: CSD-91-648, 1991.

[6] Darryl Gove, CPU2006 Working Set Size, ACM SIGARCH Computer Architecture News, v.35 n.1, March 2007.

[7] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions." In ACM SIGARCH newsletter, Computer Architecture News, Volume 34, No. 4, September 2006.

[8] A. Jaleel, R. S. Cohn, C. K. Luk, B. L. Jacob. "CMP\$im: Using PIN to Characterize Memory Behavior of Emerging Workloads on CMPs", Technical Report - UMD-SCA-2006-01

[9] A. Jaleel, M. Mattina, et al, "Last-Level Cache (LLC) Performance of Data-Mining Workloads on a CMP-A Case Study of Parallel Bioinformatics Workloads", in Proceedings 12th International Symposium on High Performance Computer Architecture (HPCA), Austin TX, February 2006.

[10] 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." In Proceedings of Programming Language Design and Implementation (PLDI), Chicago, Illinois, 2005.

[11] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. "Evaluation techniques in storage hierarchies." IBM Journal of Research and Development, 9, 1970.

[12] T Mohan, BR Supinski, et al. "Identifying and Exploiting Spatial Regularity in Data Memory References", in ACM Supercomputing Conference, 2003.

[13] S. Palacharla, R. Kessler, "Evaluating Stream Buffers as A Secondary Cache Replacement", in Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA), 1994, 24-33

[14] S. Sair and M. Charney. "Memory Behavior of the SPEC CPU2000 Benchmark Suite." IBM Thomas J. Watson Research Center Technical Report RC-21852, October 2000.

[15] Srivastava and A. Eustace. "ATOM: A System for Building Customized Program Analysis Tools", Programming Language Design and Implementation (PLDI), 1994, pp. 196-205.

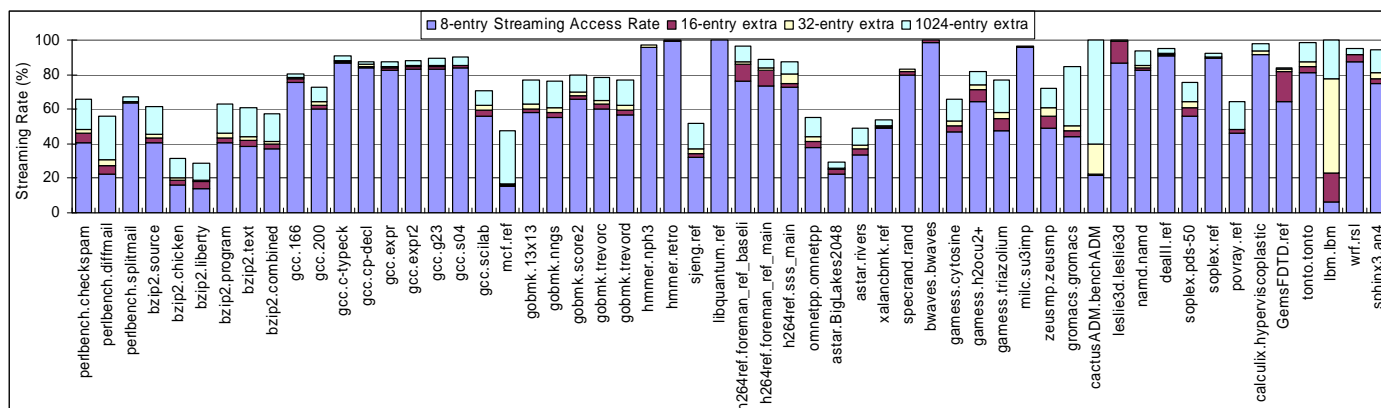


Figure 8: Streaming Coverage with Varying Number of Detection Engines. This figure shows the ratio of the accesses that are in streams with 8, 16, 32 and 1024 detection engines.