Performance Studies of Commercial Workloads on a Multi-core System

Jessica H. Tseng, Hao Yu, Shailabh Nagar, Niteesh Dubey, Hubertus Franke, Pratap Pattnaik
IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598-0218, US
{*jhtseng,yuh,shailabh,niteesh,frankeh,pratap*}*@us.ibm.com*

Hiroshi Inoue, Toshio Nakatani
IBM Toyko Research Lab., 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa 242-8502, Japan
{*inouehrs,nakatani*}*@jp.ibm.com*

Abstract

The multi-threaded nature of many commercial applications makes them seemingly a good fit with the increasing number of available multi-core architectures. This paper presents our performance studies of a collection of commercial workloads on a multi-core system that is designed for total throughput. The selected workloads include full operational applications such as SAP-SD and IBM Trade, and popular synthetic benchmarks such as SPECjbb2005, SPEC SDET, Dbench, and Tbench. To evaluate the performance scalability and the thread-placement sensitivity, we monitor the application throughput, processor performance, and the memory subsystem of 8, 16, 24, and 32 hardware threads with (a) increasing number of cores and (b) increasing number of threads per core. We observe that these workloads scale close to linearly (with efficiencies ranging from 86% to 99%) with increasing number of cores. For scaling with hardware-threads per core, the efficiencies are between 50% and 70%. Furthermore, among other observations, our data show that the ability of hiding long latency memory operations (i.e. L2 misses) in a multi-core system enables the performance scaling.

1 Introduction

To overcome the power constraint, the growing gap between memory and the processing speed, and the limited instruction-level parallelism, multi-core systems are inevitable and will play a major role in the future markets. Following the lead of IBMs Power 4 [14], and Sun Microsystems Niagara processor [15], Intel announced its multi-core to many-core roadmap [13] for general purpose computing marketplace. The current trends show that the number of cores on a chip is doubled with each silicon generation. However, in a recent study from Berkeley [9], the authors raise concerns of diminishing returns when systems grow beyond 8 processor cores. Knowing how the workloads perform in existing multicore systems is critical to identify potential scaling opportunities and challenges.

Despite the tremendous increase of interests in multi-core processors, we found few relevent in-depth performance studies of realistic workloads on such platforms. The purpose of our study is to gain better understanding of the performance impact of running typical commercial/system workloads on a real platform. We evaluate both fully operational applications (SAP and IBM Trade) and synthetic benchmarks (SPECjbb2005, SPEC SDET, Dbench, and Tbench) on a commercial available multi-core system, SUN FireT2000. We chose this set of applications/benchmarks because of their representations in typical enterprise workloads.

SAP targets practices in a diverse set of industry sectors and it is widely used in midsize and large organizations to handle tasks such as production planning, human resources, material management, sales, and distributions. IBM Trade is a stock trading application supporting multiple clients submitting weighted random requests such as logon, logoff, quote, buy, and sell. SPECjbb2005 emulates a 3-tier system that supports most common type of server-side Java application. SPEC SDET is used for performance studies of operating systems and I/O sub-systems. Dbench and Tbench are open-source benchmarks used in the Linux development community for performance evaluation of file servers.

The rest of the paper is organized as follows: In Section 2, we describe the workloads used in this study. The experiment environments and setups are explained in Section 3. We present the performance data and its analysis in Section 4. Finally, we conclude the paper and discuss the future work in Section 5.

2 Overview of Workloads

In this section, we describe the applications and benchmarks used in our study. While configuring the workloads, we are interested in the scaling of throughputs while enabling more cores and/or hardware threads. For the purpose, we have setup the workloads to be CPU bound, and briefly describe the corresponding setups here.

2.1 SAP-SD

mySAP ERP [6] is an integrated enterprise resource planning software available from the SAP AG. SAP typically targets best practices in a diverse set of industry sectors and it is widely used in midsize and large organizations. SAP provides various software modules to handle these best practices. Examples of these modules are Production Planning, Human Resources, Material Management and Sales & Distribution. The modules are written in SAP's prorietary language called ABAP (Advanced Business Application Programming) and are executed on the SAP NetWeaver platform. Like many other middlewares, mySAP can be setup as a 2-tier or 3-tier configuration, where the application serving tier can be comprised of multiple, potentially distributed, application servers. For our study, we have chosen the SAP-SD Sales and Distribution Benchmark [7], which assesses how many distinct users can be supported by a given setup.

The benchmark specifies that each user is represented by his own master data, such as material, vendor, or customer master data to avoid data-locking situations. Furthermore, each user submits a set of transactions comprised of the following: (i) create an order with five line items, (ii) create a delivery for this order, (ii) display the customer order, (iii) change the delivery and post goods issue, (iv) list 40 orders for one sold-to party, and (v) create an invoice. The benchmark runs through a ramp up phase, a measurement phase and ramp down phase. The application server is mainly comprised of two sets of processes, (i) dialog processes, which handle the interaction with the user and the data retrieval from the database and (ii) update processes, which handle modifications to the database. The benchmark measures the average response time experienced by the individual users while with the server.

For our evaluation we have chosen a 3-tier setup with a single instance of the application server and DB2 UDB V9.1 running on a 4-CPU POWER5 system. The metric for this benchmark is how many users can be suported while still satisfying the customary limit of 2.0 second average response time. We configured the application server to provide 5 dialog processes and 1 update process for each core (4 hwthreads) utilized. This has proven to effectively hide the database access latency of individual processes without reducing utilization. No extraordinary effort was spent to finetune the system, however, we ensured that SAP did not encounter any internal swapping.

2.2 Trade

Trade [4, 5, 3] is the short name of Trade Performance Benchmark Sample for WebSphere Application Server (WAS). WAS is a family of IBM software products, which provides development and deployment environment for modern multi-tier web applications [4]. Trade is designed and developed to cover the significantly expanding programming model and performance technologies associated with WAS. It is a stock trading application and allows a registered user to buy and sell stock, to check his portfolio, and so on. All Trade versions are WebSphere-version dependent. Here we used Trade v6.1 together with WAS v6.1 packages. For the Java VirtualMachine, we use Sun HotSpot 64-bit Server VM 1.5.0_09_b01.

We use a 3-tier deployment of Trade. Only the middle tier, WebSphere Application Server, runs on the multi-core system. For the database tier, we run DB2 UDB v9.1. On the client side, we run a light-weight workload simulator for web applications. In our deployment, transactions are initiated by the workload simulator, which mimics the operations performed by a collection of on-line traders. Each transaction is initiated by the driver through an http request. The output of the transaction is a web page that is sent back to the driver. Some requests are purely lookups on accounts, portfolios, and stock prices. Other requests actually perform buy or sell operations on stocks, which usually perform multiple database operations and result in relative longer latency.

The simulation workload is driven by a script written in a scripting language defined for the workload simulator engine. The script defines a distribution of the Trade requests sent to the application server. Specifically, to represent realistic workload, the distribution defines relative large numbers of data lookup operations (e.g. portfolio check, quotes check), and less data update operations (e.g. buy or sell stocks).

For the purpose of stressing the application server to its peak capacity, we put Trade's relative small database on a memory based file system to reduce the latency of database accesses. In addition, putting the database in memory has saved the need of fine-tuning data configurations on the database server. For the same purpose, we define 0 thinking delay between contiguous requests of a given set of active clients. WAS v6.1 provides effective caching service for applications to reuse short-lived or dynamic contents (servlets, JSPs, etc.), which are in the form of Java objects known as *dynamic cache*. In this study, we experiment with the dynamic caching option on.

2.3 SPECjbb2005

SPECjbb2005 (SPEC Java Business Benchmark) [2] is an industry-standard benchmark developed by the Standard Performance Evaluation Corporation (SPEC) and is widely used to evaluate performance of server hardware and Java Virtual Machines (JVMs). The benchmark simulates a 3-tier server system for a wholesale supplier. On a single tier, it is implemented as a self-contained Java application and does not require external database or application server software. In the benchmark, an arbitrary number of threads can run in parallel and each thread processes an independent dataset called warehouse. The metric of the benchmark is aggregated throughput of database transactions for all threads. In our experiments, we specify the same number of simulated warehouses (execution threads) as enabled hardware threads.

We have observed that 32-bit JVM gives about 30% better performance than 64-bit JVM, we use the 32-bit version. The JVM version we use is Sun HotSpot 32-bit Server VM

1.5.0_11. We have run with two configurations: using one JVM and using four JVMs. The one JVM configuration aims to see the scalability of the JVM with increasing number of hardware threads and the four JVMs configuration aims to achieve the best performance on the Niagara system since four JVMs configuration can achieve better result due to larger total java heap size. Here, we report results for the one JVM configuration.

2.4 SPEC SDET

Previous applications represent typical commercial workloads that exercise the entire software and hardware stacks of today's commercial software deployment. From these applications, we observed that most of their CPU usages are in user-space and little in operating system kernel space. To gain better perspective of the scalability of operating system on multi-core, multi-thread architectures, we have selected a few additional server workloads.

SDET [12] is a benchmark from SPEC SDM91 suite, which has been used for performance studies of operating systems [8] and I/O sub-systems [11]. It simulates a server used for development tasks: multiple users performing development related work: file operations, editing, compiling, test run, etc. Each user is simulated by a shell script, and has a distinct home directory and issues shell commands that are independent of other uses. Each of the shell scripts for simulating distinct users is composed of 21 pre-defined script files. The benchmark launches a number of such scripts and reports a throughput value that is defined as the number of scripts executed per hour.

Because SDET is a relatively old benchmark, targeted for much slower systems, the work defined for each of its simulated user (script) is very short when running on today's server systems. For this matter, we have each simulated user use the pre-defined script pieces 5 times to increase the work of each user.

The original results of SDET indicate that the benchmark is I/O bound while running on the Niagara system. To saturate the system CPU resource, we moved the working directory to a memory mapped file system (*tmpfs* in Solaris) to remove the I/O bottleneck. This way, we observed scalable performance, i.e. the benchmark reported throughput (number of scripts per hour) scales with the number of cores used. As expected, for a fixed system configuration, the throughput will peak around certain number of concurrent users and will slightly drop when simulating too many users. Therefore, we choose to run the benchmark with weak scaling, i.e. keep the average work of each hardware thread constant. For instance, on our Niagara system, with 1 core and 4 threads per core, we specify 32 concurrent users (scripts); with 4 core and 4 threads per core, we specify 128 concurrent users.

2.5 Dbench & Tbench

Dbench and Tbench [17] are open-source benchmarks used in the Linux development community. They are synthetic benchmarks together emulating the industry standard benchmark NetBench [1], which is used to measure the performance of file servers serving a large number of clients running on distinct physical nodes. Among the 2 programs, the dbench program produces workloads for file access, while the tbench program produces TCP load on client and server sides.

The workload of a single client of the dbench/tbench programs is defined in an input file, which is essentially a log of a real NetBench run. The input file associated to Dbench v3.04 has about 450 thousand command lines, which includes 40 thousand write operations and 120 thousand read operations. The total I/O amounts defined are 1 GigaByte for writes and 1.5 GigaByte for reads. When multiple clients are running, the benchmark introduces significant amount of I/O load. The user of the benchmark can specify a large number of concurrent clients, which are run as processes. In the case of tbench, the number of clients corresponds to the number of socket connections between the client and server nodes.

Both dbench and tbench are I/O constrainted. For instance, on our 1GHz Niagara-1 system, dbench utilizing the local disk reports up to 20 MB/s throughput and CPU is about 99% idle. For tbench when running the clients on a remote system connected via 1GB, we were only able to obtain 20% CPU utilization. Hence we measured the system by running (a) dbench by locating the file system onto a memory backed filesystem and (b) tbench over loopback, which reflects the networks stack without going over the network.

3 System Setup

Sun's Niagara [15] is currently the only commercially available general purpose processor with a large number of threads and cores. Our Niagara system has eight cores and thirty-two hardware threads, runs at a clock rate of 1.0GHz. Four threads share a single execution core and all eight cores share a single floating point unit. Within each execution core, thread selection is switched between the available threads every cycle with priority given to the least recently executed thread. Instructions are fetched, decoded, and executed in program order.

Each core contains a 16KB 4-way associative L1 Instruction Cache with 32 byte line size and a 8KB 4-way associative L1 Data Cache with 16 byte line size. A unified 3MB 12-way associative L2 cache with 64 byte line size is shared among all eight processor cores. Coherency is managed by the L2 cache and directories are maintained for all L1 caches. Each core has a 16-entried Instruction-TLB (ITLB) and a 32-entried Data-TLB (DTLB). The Niagara server that we used to evaluate the benchmark has a 6GB DDR2 off-chip memory with 23GB/s memory bandwidth.

The system runs Solaris 10. We used performance monitoring tools available on the system; specifically, *vmstat* for collecting OS level system usages, *busstat* for memory traffic, *nicstat* for network load, *cpustat* for hardware performance counters, and *dtrace* for software analysis [16].

As mentioned previously, in this study, we concentrate on

Table 1. Workloads

Apps	High-level Characteristics	Experiment loads
SAP-SD	Enterprise resource planning [7]	3-tier, Gb Ethernet, 5 dialogs and 1 update per core
Trade	Stock trading, using WebSphere Application Server [4]	3-tier, Gb Ethernet, 200 clients
SPECjbb2005	Synthetic Java bussiness workloads [2]	1 warehouse per HW thread
SPEC SDET	Software development environment [12]	memory backed filesystem, 64 concurrent users
Dbench	File access loads of file servers [17]	memory backed filesystem, 64 clients
Tbench	Network loads of file servers [17]	loopback, 64 clients

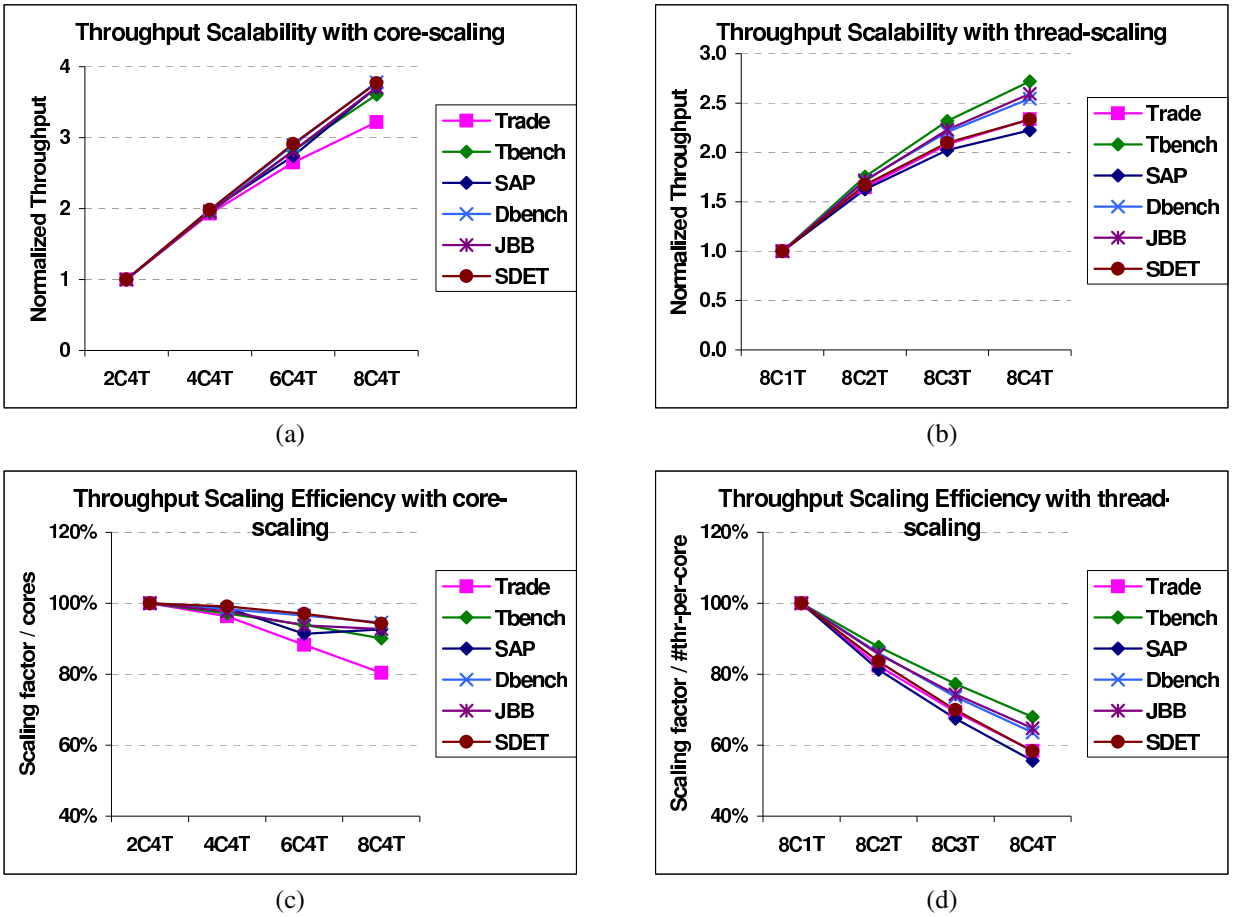


Figure 1. Application Performance Scalability

scaling the throughput of selected workloads. Table 1 highlights the selected workloads with specific configurations we used here.

4 Performance Results

Our evaluation first consists of establishing the scalability of the application's throughput while (a) the number of cores of the server (Niagara) increases and (b) while the number of hardware threads per core increases. Once the application has reached steady state, we collected the system utilization and low-level performance information such as CPI and miss rates for TLB, L1 and L2 caches. Then we report the sensitivities of these parameters to the scaling of cores or HW-threads per core. For the results presented in this paper, we use the following label: *xCyT* defines that *x* cores are enabled and each of those cores has *y* hardware threads enabled. For brevity, in the graphs, *SAP* represents SAP-SD; *JBB* represents SPECjbb2005; *Trade* represents IBM Trade v6.1.

4.1 Throughput Scalability

To facilitate cross-comparison between core-scaling and thread-scaling experiment results, the total number of HW-threads is fixed at 8, 16, 24, and 32. With respect to core scalability, we enable 2, 4, 6, or 8 cores, each with 4 HW-threads enabled (denoted as 2C4T, 4C4T, 6C4T, 8C4T). The Figure 1(a) and Figure 1(c) indicates that SAP-SD, SPEC SDET, and Dbench have very good scalabilities (a.k.a. efficiency of scaling) of 93%, 100%, and 96% respectively. The scalability of SPECjbb2005, Trade and Tbench are reasonable with 86%, 82%, and 85% respectively.

For examining the scalability of the number of HW-threads per core we report results for 8C1T, 8C2T, 8C3T, and 8C4T configurations (referred to as *thread-scaling*). Figure 1(b) shows the normalized throughput as additional HW-threads are enabled. While increasing the number of HW-threads from 1 to 4, all applications show roughly the same improvement, with an average factor of 2.5 at 4. Figure 1(d) shows that although the efficiency degrades while the number of HW-threads increases, the efficiency is still above 50%. The rate of the efficiency degradation suggests that there is room for throughput improvement with a higher number of HW-threads per core.

4.2 CPU Usage Distribution

Next, we observe the CPU time breakdowns (into user, system, and idle times) across scaling the number of cores and HW-threads per core. Because the distributions of CPU usages for core-scaling and thread-scaling are much alike, we show the core-scaling data in Figure 2.

All applications exhibit a very stable time distribution across the various scenarios. SPECjbb2005 and SAP-SD exhibit approximately 3% system time, Trade 17%. For server workloads, SPEC SDET exhibits 28% system time, Tbench 75%, and Dbench 60%. SAP-SD, SPEC SDET, SPECjbb2005, Dbench, and Tbench all essentially exhibit no idle time ($< 1\%$). Trade holds idle time below 5% for up to 6 core, but then increases to 10.96% idle time for the 8C4T configuration, which partially explains the drop in application performance.

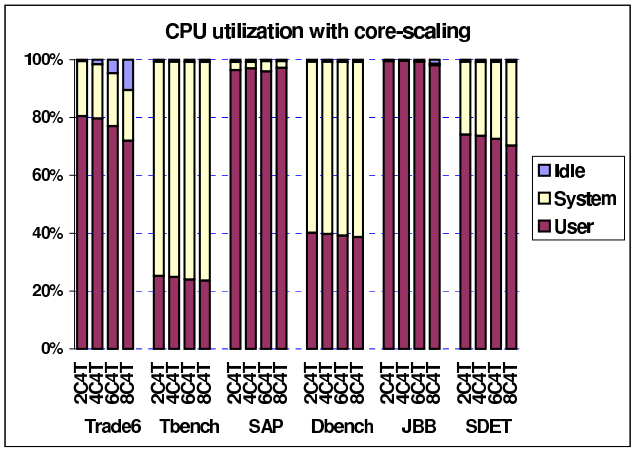


Figure 2. System CPU Usage Breakdowns

To understand the increased idle time of Trade for larger number of cores, we examined the frequencies of hot system calls. The hot system calls fall into two categories (i) communication and file accesses (i.e. read/write/send/recv), and (ii) synchronization primitives (i.e. mutex and condition variables). We observed that the frequency of communication and file primitives scaled linearly with the throughput, while the frequency of synchronization primitives increased non-linear. The system primitives are called by the JVM when locks are contended. We concluded that the middleware increasingly hits serialization bottlenecks and ultimately results in increased idle time.

4.3 Cycles Per Instruction (CPI)

We now turn to the CPI. The CPI (cycles per instructions) was computed by the number of non-idle cycles (both kernel and user modes) divided by the number of retired instructions. CPI tells us how much time is spent per instruction and is a relevant indication on how efficient the execution of an application is. Contributors to high CPI numbers are memory stalls and common resource sharing. In each core, HW-threads share L1 caches and TLBs as well as the entire execution pipeline. Hence, for one Niagara core, the ideal CPI-per-core when running 4 threads is 1; the ideal CPI-per-thread is 4.

The CPI data for the core-scaling scenarios are shown in Figure 3(a) and (c). The data shows that the CPI-per-thread values are different for the various workloads, ranging from 5.5 to 7.5. Nevertheless, for each individual workload, the CPI numbers remain essentially constant while the number of cores increases. This indicates that the performance scales with the number of cores. The results also suggest that the increased average memory latency (due to higher L2 cache resource con-

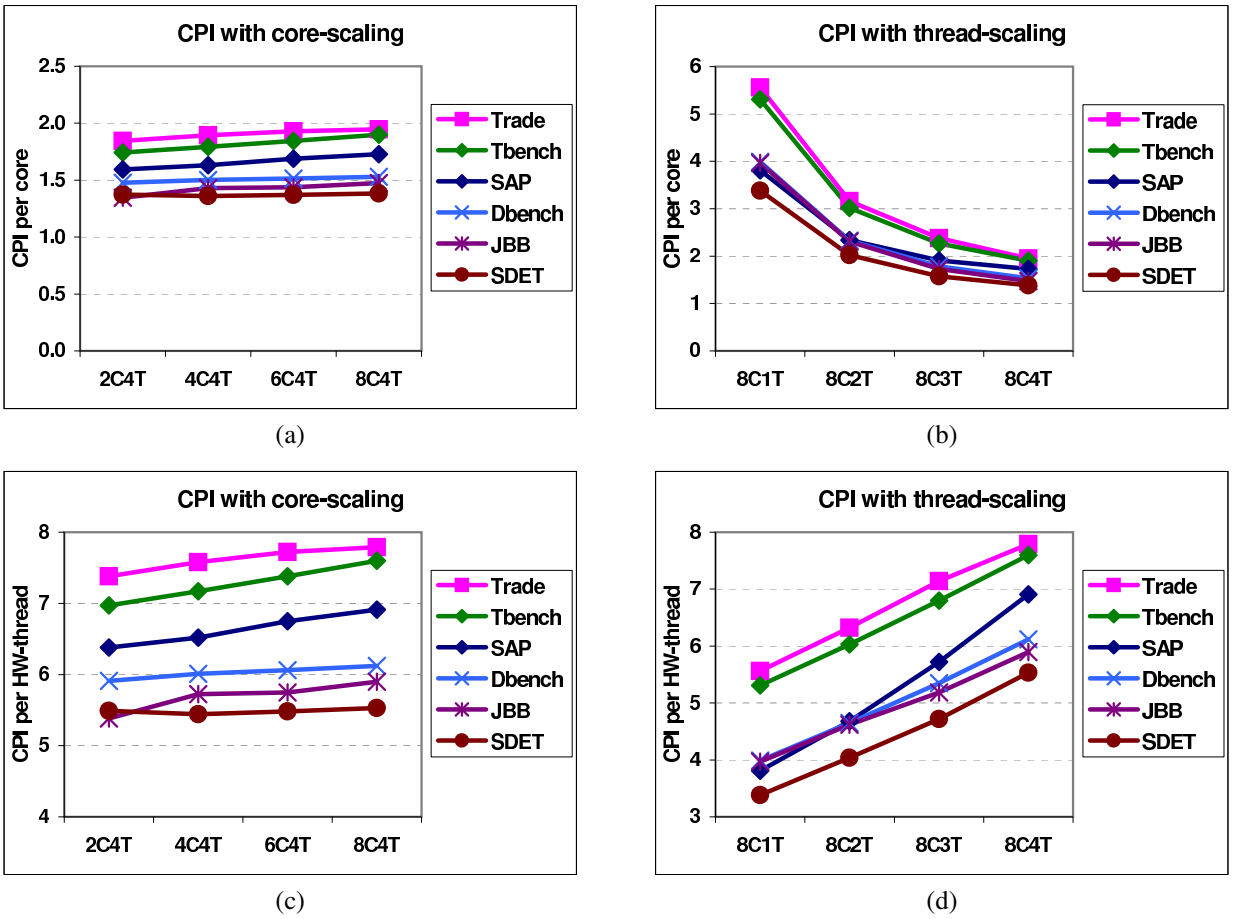


Figure 3. CPI per Core and per HW-Thread

tention) is well hidden when the number of enabled core (total HW-thread) increased.

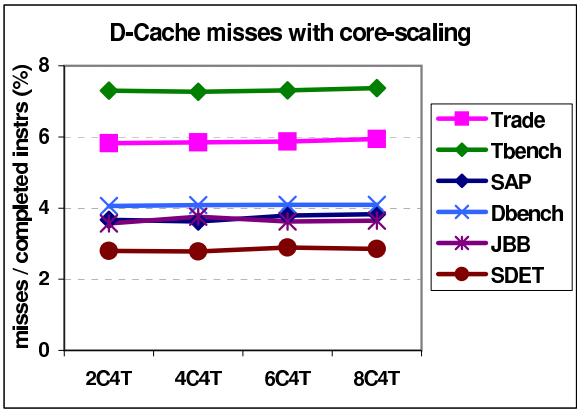
Figure 3(b) and (d) show the CPI scaling for thread-scaling scenarios. One observation is that the CPI-per-core scales well but not in linear. It improves roughly by a factor of 2.5 instead of 4. This is due to the contentions of intra-core resources such as TLBs, L1 caches, and execution pipelines. Furthermore, CPI-per-thread shows an increase of 40% to 80% and that also indicate that there is resource contention within a core. Although the resource contentions exist when we increase the number of HW-thread per core, our data show that the multi-threading configurations still give significant overall throughput improvements.

4.4 Memory Subsystem Performance

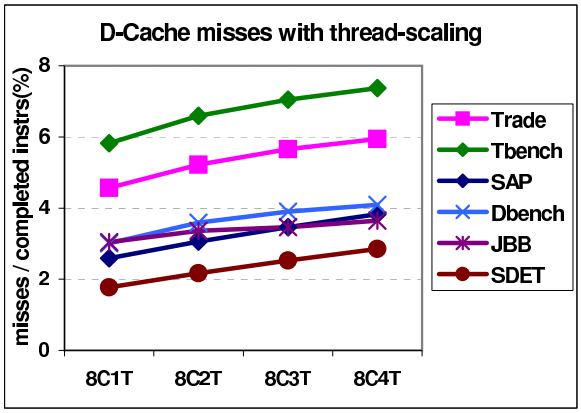
To investigate the performance of the memory subsystem, we observe miss rates for Data TLB, L1 and L2 caches. The results are shown in Figure 4. The miss rate is the percentages of misses of the total number of instructions (both memory and non-memory). From the plots for miss rates of L1 D-Cache and L1 I-Cache, Figure 4 (a), (b), (c), and (d), we note that the miss rate is significant. The relative high miss rate is due to the small sizes of the L1 caches of the experimental platform while compared to commodity systems such as Intel woodcrest that having 32 KB I-Cache and 32 KB D-Cache [10]. In addition, There are multiple hardware threads sharing the small caches.

In addition, Figure 4 (a) and (b) show that the miss rates of L1 D-Cache and L1 I-Cache are essentially flat as the number of enabled cores increased in the system. The L1 D-Cache miss rate is expected to increase if different cores share the same data and the data is frequently being invalidated by others. Therefore, our results suggest that there is limited data sharing across cores and/or the data accesses are mostly reads (instead of writes) in the commercial workloads that we evaluated.

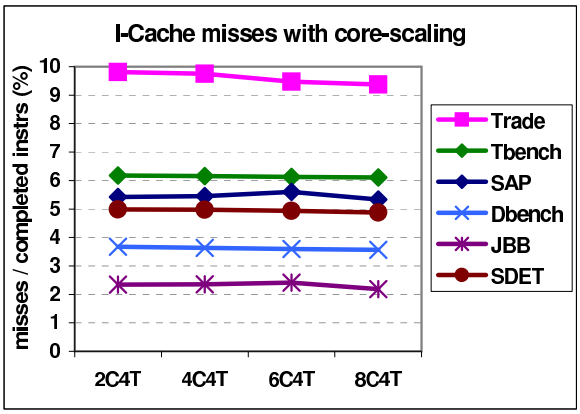
Figure 4 (b) and (d) show the miss rates of L1 D-Cache and L1 I-Cache for the thread-scaling experiments. The average L1 D-Cache miss rate increases from 3.5% for 8C1T to 4.6% for 8C4T configuration and the data across different workload illustrates a similar sub-linearly increasing trend as the number of enabled HW-thread increased per core. However, not all the workloads follow a similar trend for L1 I-Cache miss rate results. For benchmark programs such as Tbench, Dbench, and SPECjbb2005, the L1 I-Cache miss rates appear to be either slightly improved or stay consistent as the number of enabled HW-thread increases. This is due to the fact that there is instruction sharings or has little capacity conflict among the HW-



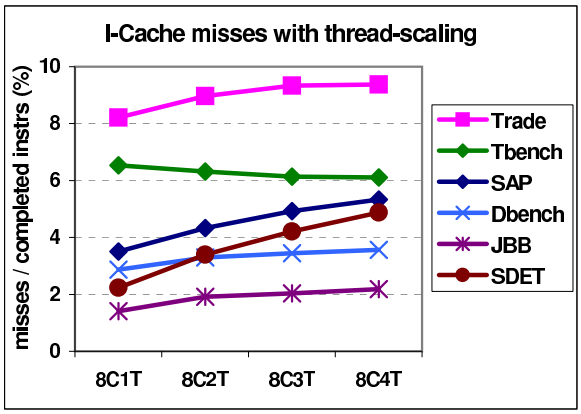
(a)



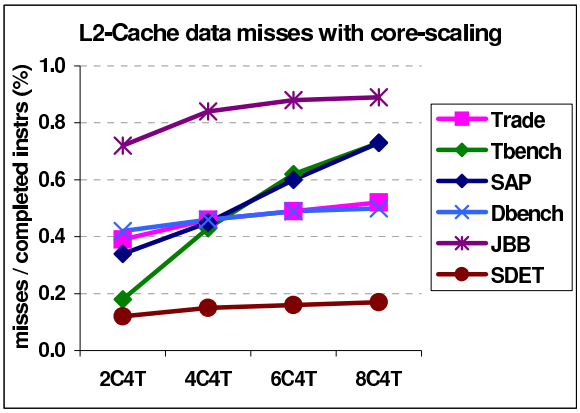
(b)



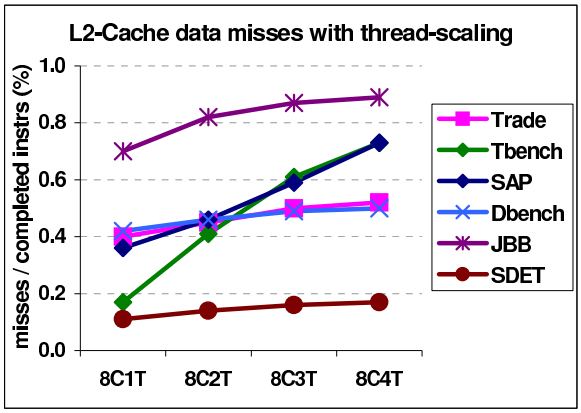
(c)



(d)

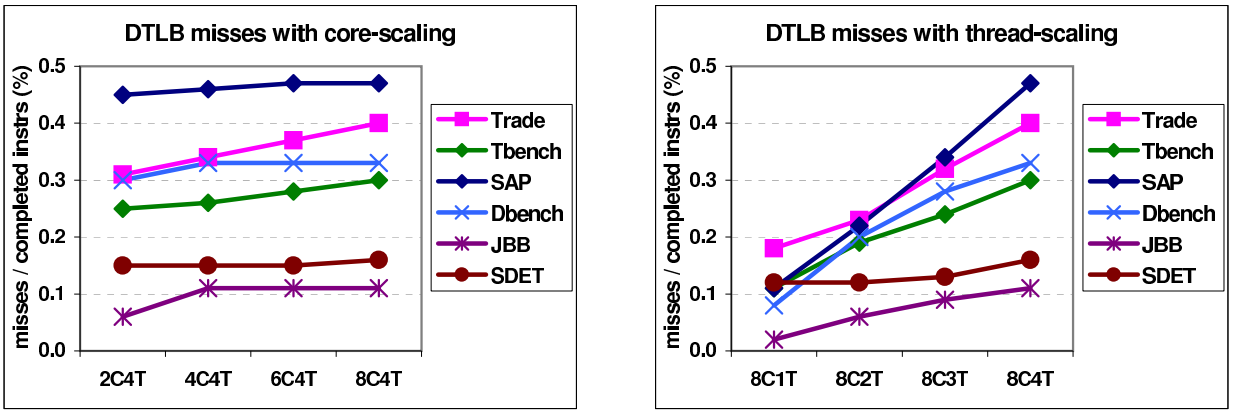


(e)



(f)

Figure 4. Cache Miss Rates



(a) (b)

Figure 5. Data TLB Miss Rate

threads running within the same core in these benchmark programs. On the other hand, the more realistic workloads such as Trade, SAP-SD, and SPEC SDET shows a slight increase in the L1 I-Cache miss rates as we scale up the number of threads per core. These observations hint that the later workloads have less instruction sharing and/or have more L1 I-Cache resource conflict between the HW-threads.

The L2-cache load miss rates for core-scaling and thread-scaling are presented in Figure 4 (e) and (f). The miss rate is dividing the number of L2-Cache misses due to load requests by the number of total instructions. Figure 1 (a) and Figure 4 (e) show that despite the increasing L2-Cache miss rates, the application throughput scales linearly with the number of enabled cores in the system. This suggests that the L2-Cache contention is not the bottleneck in the overall application performance scaling and the multi-core architecture is capable in hiding the average increased memory latency. Additionally, we notice that the L2-Cache miss rates for core-scaling, Figure 4 (e), and thread-scaling, Figure 4 (f), are almost identical. This indicates that the L2-Cache performance is independent of the placement of HW-threads across cores and is strongly related to the number of total HW-threads in the system.

As we expected, Figure 5 (a) and (b) show the DTLB miss rate increases as we increase the number of enabled HW-threads per core but remains fairly constant as we increase the number of enabled cores in the system. Figure 6 shows that the average off-chip memory traffic is much less than the maximum memory bandwidth of 23GB provided by the Niagara system. Given that the memory bandwidth usage is correlated to the number of core and the L2-Cache miss rate, as the number of core increases in the future design, close monitoring of memory subsystem performance is crucial to ensure the memory bandwidth not be a limitation.

4.5 Interference of workload collocation

Incoming requests are typically distributed to available thread without regard to their locality and threads are typically

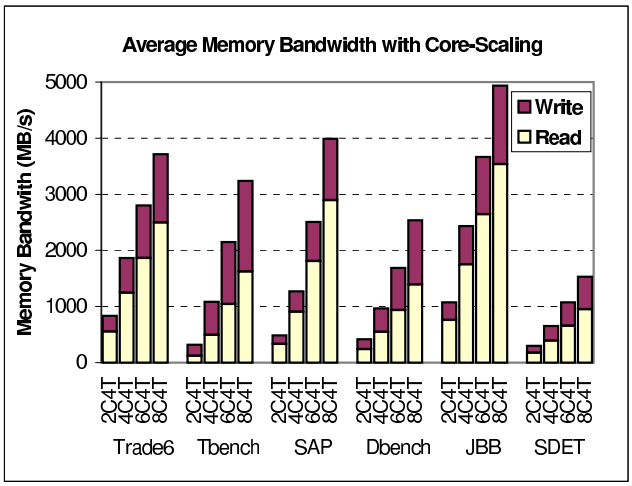


Figure 6. Average Memory Bandwidth

not bind. We speculate that if similar work is to be collocated, we can obtain better throughput due to better locality. To study this effect, we modified the transaction mix of Trade6 to represent different ratio of quote and purchase transactions. We ran the system on a four core and four threaded (total 16 hardware threads) configuration to ensure the idle does not factor into the total performance. Figure 7 shows that the average service time per transaction with respect to different mix. The percentage of x-axis represents the fraction of quote transactions; the rest of transactions are purchase transactions.

The ideal service time shown in the figure is computed as below $T_{quote} \times p + T_{purchase} \times (1 - p)$, where T_{quote} and $T_{purchase}$ are the measured serviece time for quote only and purchase only transactions, and p is the fraction of quote operations (i.e. X axis of the Figure 7).

The figure indicates that we have performance interference above 20% due to collocation between mixes of 50% to 90% with a maximum of 28% degradation at 70% of quotes. This suggests that if the workload mix is known, the performance

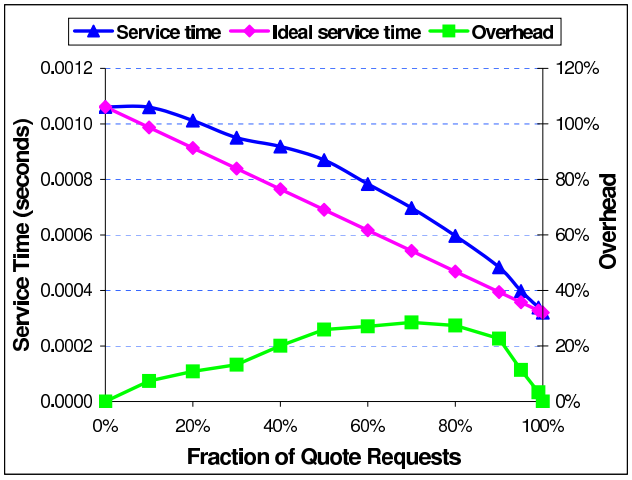


Figure 7. Trade with Quote and Purchase Requests

can be improved by proper collocating similar work. By assigning the same tasks to a set of dedicated cores, code locality can be improved and interference can be reduced. One possible implementation is for middleware to inspect incoming requests and appropriately schedule the work onto hardware threads/cores in order to create stronger locality.

5 Summary and Future Work

This paper presents our performance studies of a collection of commercial workloads on a multi-core system that is designed for total throughput. Our results confirm that multi-core architectures are suitable for the commercial applications that we evaluated. We focus on the scalability of performance at both application and system level. The data presented in the paper show that these workloads scale close to linearly with increasing number of cores. The scaling efficiency for increasing the number of hardware-threads per core are between 50% and 70%. The low-level performance data indicate that although there is resource contention among the threads, multi-core architectures are capable of hiding the associated latencies. Furthermore, our early results on Trade suggest that optimizations such as task/resource-collocations in the middleware layer will give further performance improvement.

6 Acknowledgement

We thank following people from IBM for their helpful discussions, comments, and resource-related helps: Damon Bull, Kattamuri Ekanadham, Nigel Hinds.

References

[1] IBM System p5 - NetBench Disclosure Report. IBM White Papers, at http://www-03.ibm.com/systems/p/hardware/whitepapers/netbench.pdf.

[2] SPECjbb2005 (Java Server Benchmark). http://www.spec.org/jbb2005/.

[3] WebSphere Application Server Performance Website. URL: http://www-306.ibm.com/software/webservers/appserv/was/performance.html.

[4] WebSphere Application Server V6 Scalability and Performance Handbook. *IBM Redbook number: SG246392*, at URL: http://www.redbooks.ibm.com, May 2005.

[5] Using WebSphere Extended Deployment V6.0 To Build an On Demand Production Environment. *IBM Redbook number: SG247153*, at URL: http://www.redbooks.ibm.com, June 2006.

[6] S. AG. mySAP Business Suite. http://www.sap.com/solutions/business-suite/index.epx, March 2007.

[7] S. AG. SAP Standard Application Benchmarks. http://www.sap.com/solutions/benchmark/index.epx, March 2007.

[8] J. Appavoo, M. A. Auslander, D. D. Silva, D. Edelsohn, O. Krieger, M. Ostrowski, B. S. Rosenburg, R. W. Wisniewski, and J. Xenidis. Providing a Linux API on the Scalable K42 Kernel. In *USENIX Annual Technical Conference, FREENIX Track*, pages 323–336, 2003.

[9] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. U.C. Berkeley technical report, UCB/EECS-2006-183, Dec. 2006.

[10] S. Bird, A. Phansalkar, L. K. John, A. Mericas, and R. Indukuru. Characterization of Performance of SPEC CPU Benchmarks on Intel's Core Microarchitecture Based Processor. In *2007 SPEC Benchmark Workshop*, Austin, TX, Jan. 2007.

[11] P. M. Chen and D. A. Patterson. A New Approach to I/O Performance Evaluation - Self-scaling I/O Benchmarks, Predicated I/O Performance. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1993.

[12] S. L. Gaede. Perspectives on The SPEC SDET Benchmark. http://www.spec.org/sdm91/sdet/SDETPerspectives.pdf, Jan 1999.

[13] J. B. J. Held and S. Koehl. From a Few Cores to Many: A Tera-scale Computing Research Overview. Intel Technical White Paper, Sep. 2006.

[14] S. F. H. L. J. Tendler, S. Dodson and B. Sinharoy. POWER4 System Microarchitecture. IBM Technical White Paper, Oct. 2001.

[15] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded SPARC Processor. IEEE Micro, 25(2): 21-29, March/April 2005.

[16] R. McDougall, J. Mauro, and B. Gregg. *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*. Prentice Hall PTR, 2006.

[17] A. Tridgell. The dbench Benchmark. Available: http://samba.org/ftp/tridge/dbench/.