The Impact of Performance Asymmetry in Emerging Multicore Architectures

Saisanthosh Balakrishnan[†] Ravi Rajwar[‡] Mike Upton[§] Konrad Lai[‡]

[†]*Computer Sciences Department*
University of Wisconsin-Madison
sai@cs.wisc.edu

[‡]*Microarchitecture Research Lab* [§]*Digital Enterprise Group*
Intel Corporation
{ravi.rajwar, mike.upton, konrad.lai}@intel.com

Abstract

Performance asymmetry in multicore architectures arises when individual cores have different performance. Building such multicore processors is desirable because many simple cores together provide high parallel performance while a few complex cores ensure high serial performance. However, application developers typically assume computational cores provide equal performance, and performance asymmetry breaks this assumption.

This paper is concerned with the behavior of commercial applications running on performance asymmetric systems. We present the first study investigating the impact of performance asymmetry on a wide range of commercial applications using a hardware prototype. We quantify the impact of asymmetry on an application's performance variance when run multiple times, and the impact on the application's scalability.

Performance asymmetry adversely affects behavior of many workloads. We study ways to eliminate these effects. In addition to asymmetry-aware operating system kernels, the application often itself needs to be aware of performance asymmetry for stable and scalable performance.

1. Introduction

A multicore processor provides increased total computational capability on a single chip without requiring a complex microarchitecture. As a result, simple multicore processors have better performance per watt and area characteristics than complex single-core processors. The diminishing returns on serial performance with increasingly complex cores make multicore organizations particularly attractive.

A performance-asymmetric multicore organization, where individual cores have different compute capabilities, is attractive because a few high-performance complex cores can provide good serial performance, and many low-performance simple cores can provide high parallel performance. Simple cores provide efficient use of transistors for computation in addition to meeting power and thermal budgets. Complex cores, while inefficient, provide computational power for single threads that require it. Researchers have proposed numerous such asymmetric processor organizations for power and performance efficiency, and have investigated the behavior of multi-programmed single threaded applications on them [5, 8, 9, 11, 17].

Performance asymmetry in multicore systems breaks a long-standing assumption made by multi-threaded application developers. These developers typically assume all computational cores provide equal performance when they write their parallel algorithms and applications. However, no study has investigated the impact, if any, of computational asymmetry on the behavior of these multi-threaded applications. For example, does computational asymmetry result in unpredictable performance characteristics of a commercial server, which must meet certain performance guarantees? Does the asymmetry expose an application's scalability problem, that otherwise would not have manifested? Ensuring that applications run as expected on a new architecture is crucial for the architecture's adoption. Answering questions regarding application behavior predictability and scalability are therefore important for understanding the implications of asymmetric architectures on software that runs on them. This paper tries to answer these questions.

We present the first study investigating the impact of performance asymmetry on the behavior of numerous multithreaded applications. We use a hardware prototype of a multiprocessor system to perform our study. We approximate performance asymmetry by varying the individual processor frequencies in a multiprocessor. Varying frequency is an effective way to create the property of performance asymmetry on real hardware.

We focus on two questions:

1. Does performance asymmetry in a multiprocessor system have a negative impact on an application's performance characteristics? Can we predict performance of an application on an asymmetric system? Does the application scale as expected or does it experience scalability bottlenecks that were otherwise not present on a symmetric system?

2. For applications that do suffer due to performance asymmetry, what methods can help alleviate the problem? Is exposing the asymmetry to the operating system sufficient or, do the applications also need to consider asymmetry at an algorithmic level?

We establish a baseline performance behavior by studying the applications on a performance-symmetric system and ensuring their performance is predictable on such systems, and then vary individual frequencies (at ¼ fraction increments) to study the impact of performance asymmetry. We determine whether the application provides predictable performance by running the same application multiple times on the same asymmetric setup. We also determine whether the application performance scales predictably in proportion to the total compute power in the system (even if the cores are performance asymmetric).

We then investigate ways to eliminate anomalous behaviors if any. Does re-structuring the operating systems scheduler help? Do we also need to modify the application?

Our benchmarks include commercial managed runtime servers (SPECjbb and SPECjAppServer), database servers (TPC-H), web servers (Zeus and Apache), scientific applications with closely coupled synchronization (SPEC OMP), a media application (H.264), and an application development tool (PMAKE).

Using the above workloads, the paper quantitatively makes the following four key points:

1. Performance asymmetry in systems adversely affects the predictability of a number of commercial workloads, and makes them less scalable. This effect increases with increasing concurrency.
2. An asymmetry-aware operating system helps eliminate unpredictability in some applications. In others, the application also needs to be asymmetry-aware.
3. An asymmetric multiprocessor gives higher performance than a multiprocessor in which all cores are slow because the fast core is effective for serial portions of the threaded program.
4. Mechanisms for exchanging asymmetry information between hardware and software, and application design methods tolerant of asymmetry need investigation.

Section 2 discusses the experimental methodology and Section 3 presents workload description and analysis. Section 4 summarizes the results, Section 5 presents related work, and Section 6 concludes.

2. Experimental methodology

Our experimental platform comprises a 4-way 2.8 GHz Intel® Xeon™ multiprocessor (Shasta series). Our benchmark under test runs on this platform. We disable Hyper-Threading in all processors by using the BIOS. The system has 2-MB of unified Level-3 cache. We use Windows Server 2003 and Linux operating systems.

Intel Xeon processors allow software to change the active duty cycle of processors for thermal management [6]. Duty cycle is the time-period during which the clock signal drives the processor chip. A *stop clock* mechanism disables the processor clock, during which time everything on the processor stops. This does not affect any modules outside the processor, e.g., coherence network or memory (DRAM) is unaffected, and only the processor appears to slow down. We vary duty cycle in multiple steps: 12.5%, 25%, 37.5%, 50%, 63.5%, 75%, and 87.5%.

We developed a device driver to control asymmetry in Windows Server 2003. The driver, running in privileged mode, controls the duty cycle by changing the clock modulation register. Linux kernel 2.4 requires a new module to read and write the clock modulation register. Linux kernel 2.6 provides a thermal monitoring infrastructure and does not require the user to write to the clock modulation register directly. A process-affinity API selects specific processors.

Using the above duty cycle modulation to emulate performance asymmetry is effective for this paper's study because the unpredictability of performance and limitations in scalability arise due to differences in the computation power of the individual cores and not due to communication latencies. The results therefore should hold when the communication network and latencies change.

3. Results and analysis

All applications were set up according to the rules provided by the respective organizations (SPEC and TPC). An industrial performance evaluation group ensured strict compliance. Many of these setups were multi-tier, and only the application, whose behavior we were studying, ran on the system described in Section 2.

We focus on two key predictability metrics:

1. Is the application's performance stable?
2. Is the application's performance scalable?

An *n*f-*m*s/*scale* label means *n* fast cores and *m* slow cores running at *1/scale* the speed of the fast cores. The total compute power of this system is (*n* + *m/scale*)). Symmetric configurations are 4f-0s, 0f-4s/4, and 0f-4s/8, and asymmetric configurations are 3f-1s/4, 3f-1s/8, 2f-2s/4, 2f-2s/8, 1f-3s/4, and 1f-3s/8. Performance asymmetry was validated using runtimes of computationally intensive micro benchmarks.

3.1 SPECjbb

SPECjbb2000 [19] is an online-transaction processing business Java application emulating a three-tier system. A thread represents an individual terminal, and maps to a specific warehouse. Increasing the number of warehouses increases concurrency in the workload. A probability distribution determines queries to the system. The throughput of business operations per second is the primary performance metric. The backend database is memory resident

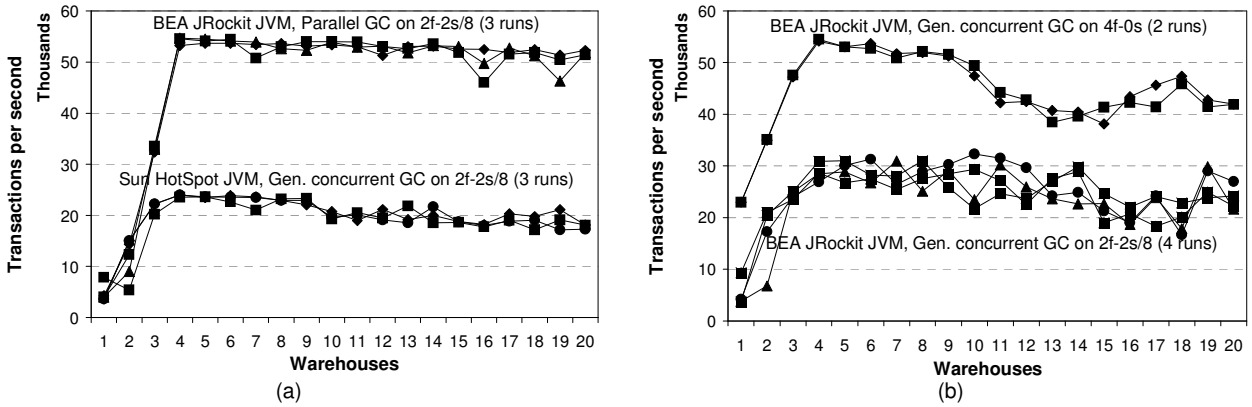

Figure 1. SPECjbb performance predictability.

(25MB) and has 20 warehouses. The workload focuses on the middle tier running the Java application server.

Our application server operating system is Linux 2.6, and we study two virtual machines for the application server: BEA Weblogic JRockit (8.1) and Sun Hotspot (1.4.2). The virtual machine flags were set for the highest machine optimization.

Since garbage collection (GC) is integral to managed runtime systems, we include their effect in this study. We use two garbage collectors: a parallel and a concurrent generational. A parallel collector interrupts all application threads prior to performing collection, and is well suited for high-throughput long-running workloads. The generational concurrent collector runs concurrently with the application, reclaiming objects. This collector is well suited for applications requiring minimal pause times and those that are unaffected by the collectors interference.

3.1.1 Analysis

Figure 1(a) shows SPECjbb throughput (increasing warehouses increases concurrency) with two different virtual machines (BEA JRockit with parallel GC and Sun HotSpot with a generational concurrent GC) running on a 2f-2s/8 asymmetric configuration. Multiple runs are shown for each configuration to determine predictability. The absolute performance variance for the HotSpot configuration is higher. Minor instability exists with JRockit. Figure 1(b) shows SPECjbb throughput (with increasing warehouses) with BEA JRockit and a generational concurrent GC (instead of the parallel GC). The new collector has a significant negative impact on application behavior. Instability in asymmetric configurations increases significantly across multiple runs, and increases as concurrency in the system increases. This increased instability is due to the concurrently running garbage collector interfering with the main application.

Figure 2(a) shows scalability and predictability (when run multiple times) as computational power is varied. For symmetric configurations (4f-0s, 0f-4s/4, and 0f-4s/8), performance decreases predictably and linearly with computational power. The workload is inherently stable and predictably scalable on a symmetric system. While asymmetric configurations (3f-1s/4, 3f-1s/8, 2f-2s/4, 2f-2s/8, 1f-3s/4, and 1f-3s/8) scale, they show significant variability (as shown by the error bars in the figure).

We investigated various potential sources of instability: scheduling, locking and synchronization, and cache

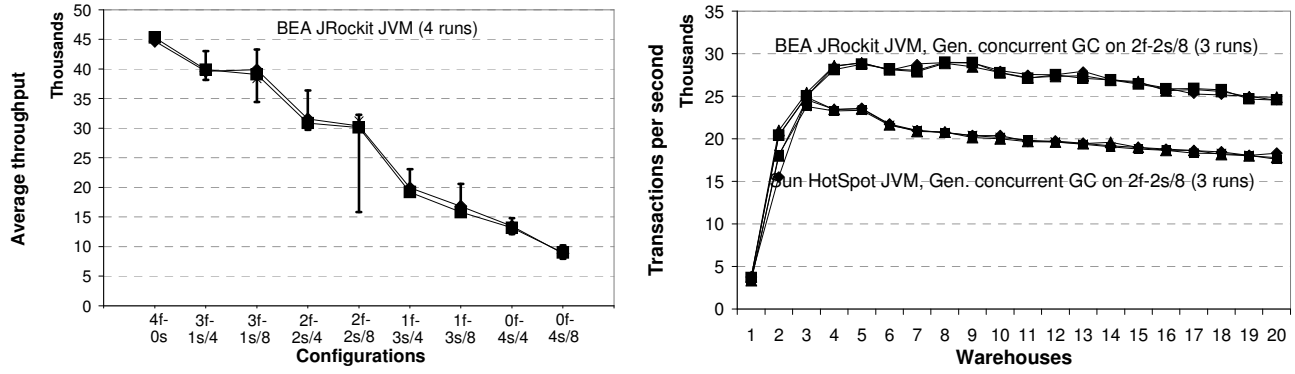

Figure 2. SPECjbb. (a) Scalability & predictability. Error bars indicate variability in throughput for multiple runs. (b) Predictability with an asymmetry-aware kernel scheduler.

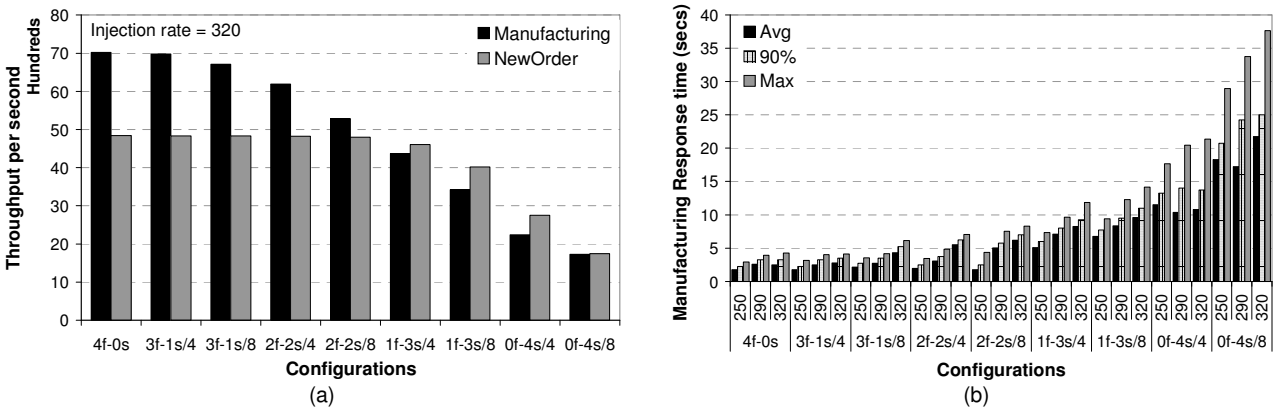

Figure 3. SPECjAppServer. (a) Performance scalability. (b) Performance predictability measured as response time.

thrashing. We identified the operating system scheduler as the primary source of instability. Instability is often due to work imbalance among various threads, and a computationally important thread's schedule—whether it runs on a slow processor or a fast processor—will vary performance significantly. A kernel scheduler typically aims to keep all processors occupied approximately with the same load.

To better balance computational power, we implemented a new kernel scheduler. We made the scheduler aware of the underlying hardware's performance asymmetry. In the new algorithm, the kernel scheduler ensures faster cores never go idle before slower cores. A process is explicitly migrated from a slow core to an idle fast core, if one is available. Figure 2(b) shows the throughput using the new kernel scheduler. As we can see, the new scheduler eliminates the application instability (compare this with Figure 1 which showed significant instability for the same configuration). For SPECjbb, exposing performance asymmetry to the operating system for smart scheduling is effective in fixing instability.

3.1.2 Discussion

Garbage collection has significant impact on throughput in an asymmetric system. The current study used single-thread concurrent garbage collection and a parallel multi-threaded non-concurrent collector. Future garbage collector designs should take into account underlying performance asymmetry, to ensure stable application behavior.

3.2 SPECjAppServer

SPECjAppServer2002 [19] is a complex client/server business J2EE™ application to measure the scalability and performance of J2EE servers and EJB containers. The setup consists of three machines: a front-end driver, a middle-tier jAppServer, and a backend database server. The study focuses on the jAppServer, and its interaction with asymmetry.

SPECjAppServer models four business domains, each with its own database and applications. These domains interact as needed. We focus on two of these domains: manufacturing and customer. The customer domain focuses on order processing, and the manufacturing domain handles production scheduling.

A driver generates requests for orders at a specific injection rate to the jAppServer using a pre-defined transaction mix. A complex sequence follows involving all domains, and the request must be serviced within a response time requirement. If the jAppServer cannot respond within a fixed time, the driver is informed, and the injection rate of requests is scaled down. This feedback loop is an integral part of the workload. SPEC rules require specified and actual injection rates to be identical for conformance. Our baseline setup where all cores have equal performance satisfies this requirement. Introducing asymmetry makes some runs non-conforming, but correct. Such runs are acceptable for this paper since they provide intuition on how asymmetry affects application behavior.

The front-end driver runs on a 4-way 2.8GHz Intel Xeon multiprocessor with 4GB memory, and the back-end database server is Microsoft SQL with Windows Server 2003 running on a 4-way Pentium III multiprocessor. These machines are powerful enough to stress test the jAppServer, and are not bottlenecks. A gigabit network connects all machines.

The jAppServer uses BEA Weblogic (8.1) application server and a BEA JRockit (8.1) virtual machine. Run parameters conforming to SPEC are used. We assume two ordering and two manufacturing agents, and the runs include a 600 seconds ramp-up, a steady state of 1800 seconds, and a 300 seconds ramp-down.

3.2.1 Analysis

Figure 3(a) shows SPECjAppServer throughput for transactions in the manufacturing and customer (NewOrder) domains. Average throughput for 4f-0s, 3f-1s/4, and 3f-1s/8 is mostly constant, and sees a linear reduction for the remaining configurations. The workload is predictably scalable on symmetric systems. The throughput specified for the configurations 2f-2s/8, 1f-3s/4, and 1f-3s/8 actu-

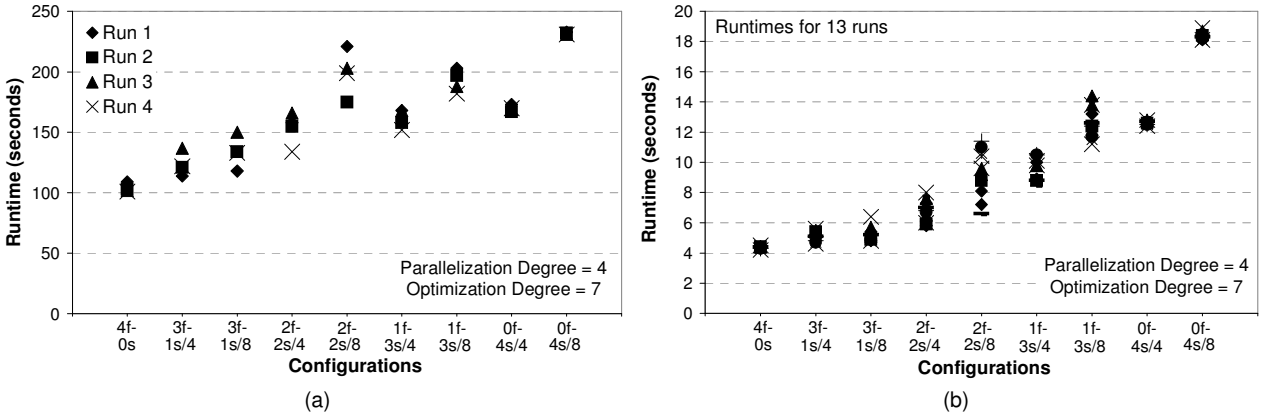

Figure 4. TPC-H. (a) Runtime for power run (all queries). (b) Runtime for one query (Query 3).

ally reflect a slower injection rate because the slow system cannot sustain a high rate and the feedback provided to the driver slows it down. The configurations 3f-1s/4 and 3f-1s/8 can sustain the specified injection rates and thus provides the same throughput.

Throughput shown in Figure 3(a) demonstrates scalability, but does not directly demonstrate stability. We investigated numerous secondary metrics and found no instability under symmetric and asymmetric configurations. Figure 3(b) shows one such metric, the manufacturing domain response time (for three different injection rates). Three bars plot average, maximum, and 90%ile response times. The response times are not constant, mainly because of complex interactions in the system, but they scale well. The 90%ile response is closer to the average, thus indicating a significant number of transactions take an average time to complete. Further, the difference between the 90%ile and the average response times scales and is stable across various configurations. These bars would have reflected any significant instability introduced due to asymmetry.

3.2.2 Discussion

SPECjAppServer adapts to dynamic performance variability by automatically scaling back and performing load balancing. This allows for stability and prevents the system from overloading. In contrast, SPECjbb discussed earlier suffered from significant instability due to performance asymmetry. This suggests that application design is an important consideration when dealing with performance asymmetry. Exposing asymmetry to software developers and application optimizers can ensure application stability.

3.3 TPC-H

TPC-H [20] is a decision support benchmark consisting of 22 non-trivial queries. Each query has varying complexity, and performs concurrent data updates on a common database. The database server used is IBM DB2 (8.2) running on Linux 2.4. We use a memory-resident setup (1 setup (1 GB with a scale factor of 1) to isolate the impact of processor execution on the database server.

The database server uses the *degree of intra-query parallelization* parameter to parallelize a query into subqueries and execute them in parallel. It also uses the *degree of optimization* parameter to optimize the query plan and its execution. Parallelization and optimization of TPC-H queries significantly improves their performance.

We focus on TPC-H query execution time during a power run. The power run measures the raw query execution time with a single active user. We discard the first few power runs and warm up the buffer spaces.

3.3.1 Analysis

We first run the benchmark with the highest optimization degree (seven) and a parallelization degree of four. Figure 4(a) shows the runtime for the power run (where all queries are run in series to completion). Multiple such runs are shown. The symmetric configurations of 4f-0s, 0f-4s/4, and 0f-4s/8 show stability across multiple runs—these points are closely clustered. TPC-H also shows good scalability, when compute power is varied. However, the asymmetric configurations (3f-1s/4, 3f-1s/8, 2f-2s/4, 2f-2s/8, 1f-3s/4, and 1f-3s/8) display significant variability across multiple runs.

To understand behavior of individual queries on performance asymmetric systems, we looked at various individual queries. Figure 4(b) shows multiple runtimes for one specific query—query number 3. Similar to the power runs, symmetric configurations are stable but asymmetric configurations show significant instability.

We explicitly turned off intra-query parallelization (graph not shown). The query showed two distinct runtimes over multiple runs—one where the runtime corresponds to the fastest processor, and another where the runtime corresponds to the slowest processor. Thus, the scheduling decisions have an impact on the application stability and this information needs to be available to the DB2 server for helping it make scheduling decisions.

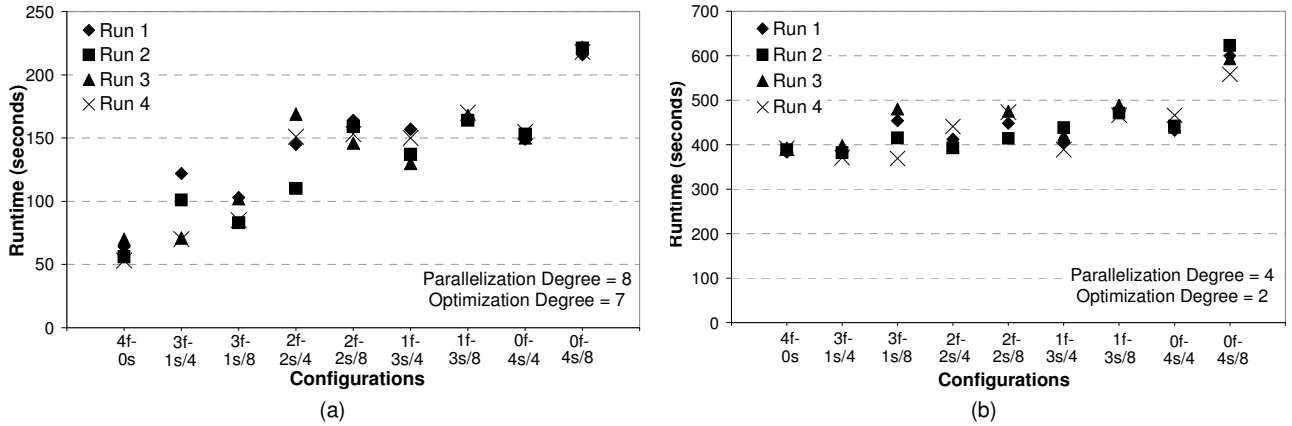

Figure 5. TPC-H power run. (a) High parallelization degree. (b) Low optimization degree.

We expected that increasing the degree of parallelization might reduce asymmetry effects by forcing more load on certain processors. Figure 5(a) shows the impact of increasing the degree of parallelization to eight. Contrary to our expectation, the runtime for various configurations show even more variance, at times twice the variance of the configuration with the parallelization set to four.

Since the parallelized queries execute on an asymmetric processor setup, the scheduling decisions to run certain parallel sub-queries on slow or fast processors over multiple runs affects the stability of the application. Our modified kernel scheduler did not fix the instability. The DB2 server controls the scheduling of query execution on server processes, which are bound by the server to various processors, thus making our kernel fix ineffective.

We approximated changes to the program itself by controlling the query optimization degree—the higher the degree, the more aggressive the query plan. The results we have discussed so far have used the highest optimization degree. Figure 5(b) shows the impact of reducing the optimization level significantly. Expectedly, the runtimes have also slowed down for various configurations as compared to the highest optimization level. However, the instability has significantly decreased for the asymmetric configurations, at times nearly a factor of 10 lesser than with the high query optimizations.

3.3.2 Discussion

The query optimization experiment strongly suggests that the application itself, and not the operating system scheduler, contributes to the instability. This suggests exposing hardware performance asymmetry to the application. Query plan generators already take into account latency of memory accesses and disk access when computing cost and the target plan. Incorporating performance asymmetry and compute power of available processors will ensure stable performance and execution behavior for these queries.

3.4 Apache and Zeus web servers

We evaluate two commonly used webservers: an open-source webserver, Apache (2.0.40), and a commercial webserver, Zeus (4.3).

Apache maintains several idle processes waiting for incoming requests. A single control process launches child processes, and these processes wait for incoming requests. Optimally selecting the number of such pre-forked processes and the maximum number of such processes allowed prevents system thrashing. A process handles a predefined number of requests, and then terminates and recycles. The control process also terminates excessively idle processes.

Zeus utilizes a small, fixed number of single-threaded I/O multiplexing processes, and these processes handle tens of thousands of simultaneous connections.

We use *ApacheBench* to drive these webservers. We calculate processing performance of a single static file. This allows us to focus on multi-threaded webserver behavior in the presence of performance asymmetry, and removes dependency of results on the web-caching infrastructure that is otherwise necessary for complex setups. We emulate two modes: (a) heavy load and full utilization processing 60 requests concurrently with up to 1,000,000 requests in total and (b) light load with 10 concurrent requests with up to total 100,000 requests.

3.4.1 Analysis

Figure 6(a) shows system throughput for Apache under heavy and light load, and Figure 7 shows the same for Zeus. We plot six runs for each configuration to determine stability. Performance-symmetric configurations (4f-0s, 0f-4s/4, and 0f-4s/8) are scalable and show stability for both light and heavy load; the throughput of these runs cluster together for both Apache and Zeus.

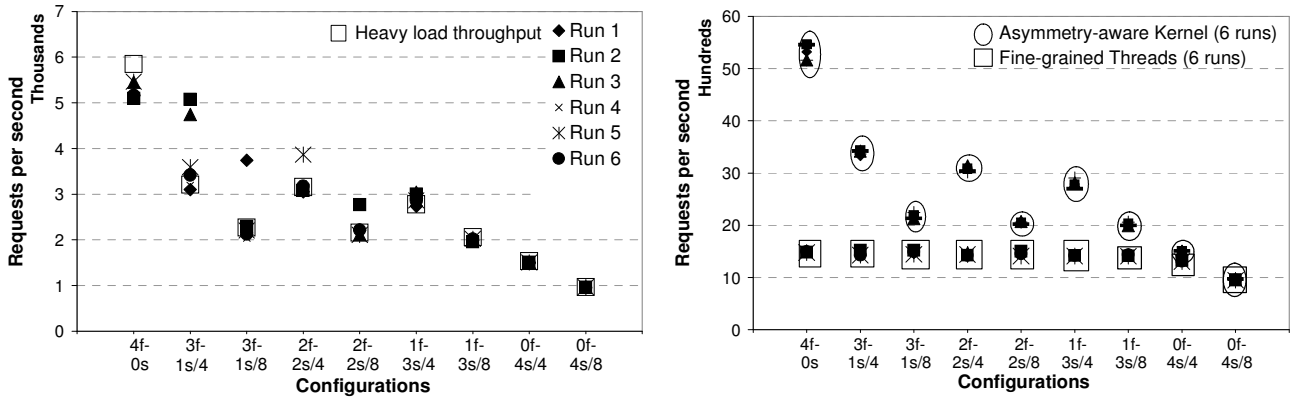

Figure 6. Apache throughput. (a) Light load. (b) With two techniques to reduce asymmetry impact.

However, as can be seen by the vertical spread of the data points in Figure 6(a) for a given asymmetric configuration, Apache performance under light load on asymmetric setups is significantly unstable. The process scheduling decisions and the server design contribute to the instability. Apache forks processes in advance to handle incoming web serving requests. Since these processes are visible to the kernel, the kernel scheduler decisions affect their scheduling. Sometimes the kernel scheduler places processes on slower cores even though a faster core is available because it is agnostic to the relative speed of the processors.

We found Apache performance under heavy load to be stable with no variance over multiple runs. The throughput with varying configurations is stable and scalable and a function of the underlying computational power of the system, because in a throughput benchmark under heavy load, each processor is always busy. Instability with performance-asymmetry typically arises in throughput-oriented applications when some processors are idle. In such situations, threads may randomly schedule on fast or slow processors.

Unlike the Apache webserver, Zeus displays significant variance and instability for both heavily loaded and lightly loaded systems. However, Zeus provides a significantly higher throughput than Apache does, up to a factor of 2.5. Due to significant instability, we cannot determine whether Zeus is predictably scalable. Since Zeus is a commercial product and we do not have access to its source code, we cannot isolate the reasons for instability.

We replaced the Linux kernel scheduler with our modified scheduler described earlier in Section 3.1.1. Our new scheduler is aware of the performance asymmetry of the underlying system and makes scheduling decisions accordingly. Figure 6(b) shows the result for Apache with light load. As can be seen, the kernel fix solves the instability problem and the runs are now repeatable.

We ran Zeus with our modified Linux kernel scheduler. The scheduler did not have any effect on the significant instability, suggesting that Zeus runs its own threading scheduler. This again demonstrates that simply exposing the asymmetry to the operating system is not sufficient. The webserver also needs to be aware of the asymmetry.

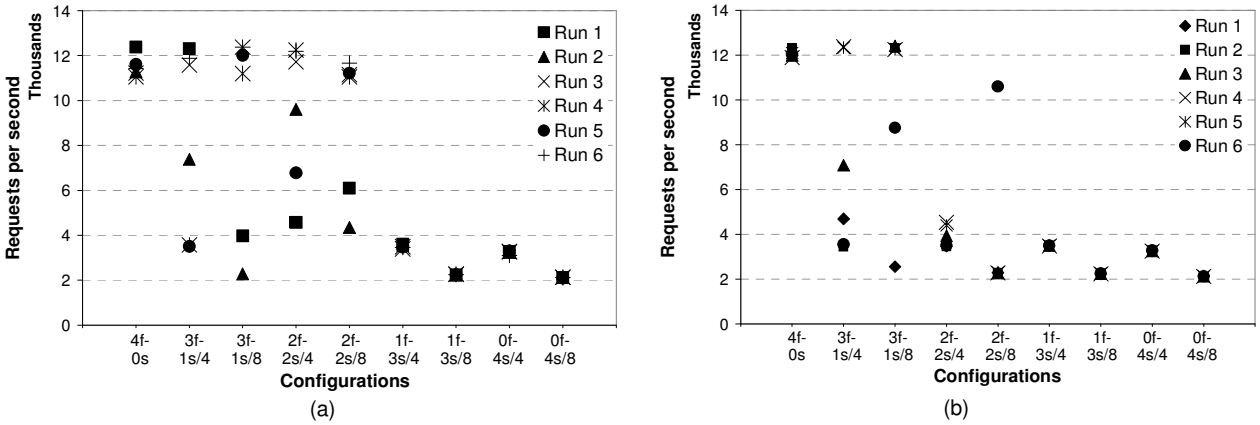

Figure 7. Zeus throughput. (a) Light load. (b) Heavy load.

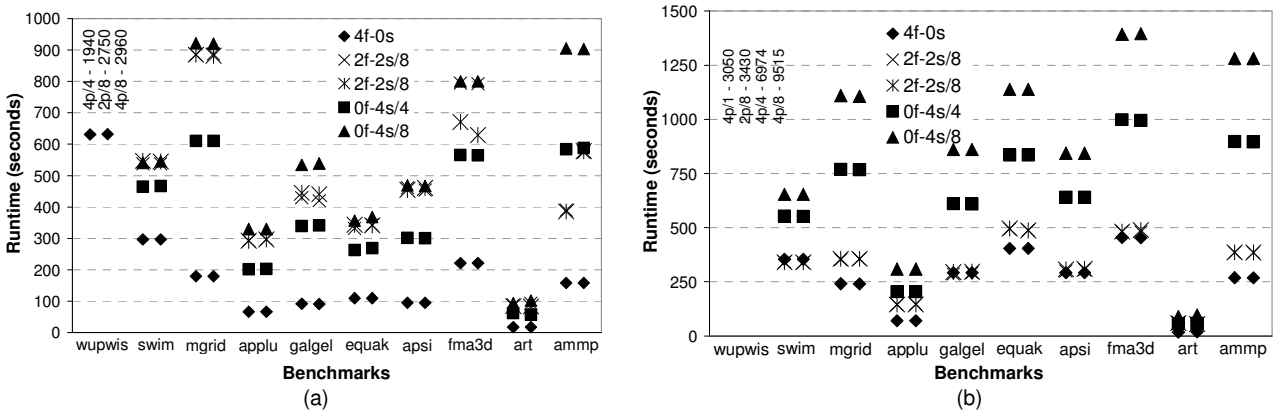

Figure 8. SPEC OMP runtimes. (a) Unmodified source. (b) Source modified to use parallelization directives.

3.4.2 Discussion

The degree of instability varies as asymmetry is varied. Higher instability exists for the 3f-1s/8 and 3f-1s/4 configurations over the 2f-2s/8 and 2f-2s/4 configurations. Introducing slight asymmetry (for e.g., a system with mostly fast processors but one slow processor) seems to introduce more instability than a system in which the compute power provided by the fast processor is a small fraction of the total compute power in the system.

Fine-granularity threading impact: Since instability occurs due to poor load balance, it could be eliminated if for example a large number of short-lived processes were available. In Apache, we can vary the lifetime of a request handling process after which time the process is re-cycled. So far, we assume optimal server parameters and the re-cycling occurs after handling 5,000 requests. Since request-handling processes take a short time to complete, we reduce the re-cycling threshold to 50 requests. This creates many processes. Figure 6(b) shows the results. The throughput of such a configuration is significantly lower than optimal parameters, and the throughput does not scale. This is because of the frequent re-cycling of processes, which causes significant overhead. However, instability under light load disappears. Instability disappears because now the scheduler has large number of short-lived processes available for scheduling. This results in automatic load balancing, however with low performance. The randomized and short use of fast and slow processors minimizes asymmetry's negative effects.

This suggests an alternative approach to eliminating asymmetry: dividing the task into a finer granularity if the overhead of managing large number process creations is acceptable.

3.5 SPEC OMP

SPEC OMP is a high-performance scientific application suite consisting of parallelized FORTRAN programs based on OpenMP libraries. We compile the suite using the Intel Fortran compiler (8.1) with the highest optimization flags. We use the OMPM2001 medium input set with a minimum memory requirement of over 1.6 GB. The OpenMP implementation uses the `pthreads` library in Linux to support various parallelization primitives. The benchmarks primarily use work-sharing parallel and for-all constructs to parallelize loop executions. A barrier at the end of loops synchronizes different threads running on the processors. These applications infrequently use critical-section synchronization constructs.

Figure 8(a) shows the runtimes for the SPEC OMP benchmarks (`gafort` is not shown because of compilation issues). Multiple runs for a given configuration are shown. As can be seen, the symmetric configurations (4f-0s, 0f-4s/4, and 0f-4s/8) are stable and scalable. Many benchmarks in the suite are also stable in the 2f-2s/8 asymmetric configuration. However, they do not show predictable scaling. Except for `ammp`, other benchmarks have a 2f-2s/8 runtime closer to the 0f-4s/8 runtime. The 2f-2s/8 runtime for `galgel` and `fma3d` is worse than a 0f-4s/4 runtime, but a 2f-2s/8 configuration has more computation power than a 0f-4s/4 configuration. In the 2f-2s/8 configuration, the slowest processor limits application performance, thus forcing it to behave similar to a 0f-4s/8 configuration. 2f-2s/8 is a little better than 0f-4s/8 as the faster processors in 2f-2s/8 can improve serial performance.

To understand why SPEC OMP programs on asymmetric systems are not predictably scalable, we analyzed their algorithms. The applications mainly use do-all and parallel loops. OpenMP [18] provides three major parallelization modes: static, dynamic, and guided parallelization of loops. In static mode, equal division of loops among processors occurs at the beginning of execution. In guided and dynamic modes, processors request more work in chunks, as they complete work. The two modes differ in the work assigned to the requesting processors. In guided mode, all processors start with the same chunk size, and the chunk size decreases exponentially as processors finish execu-

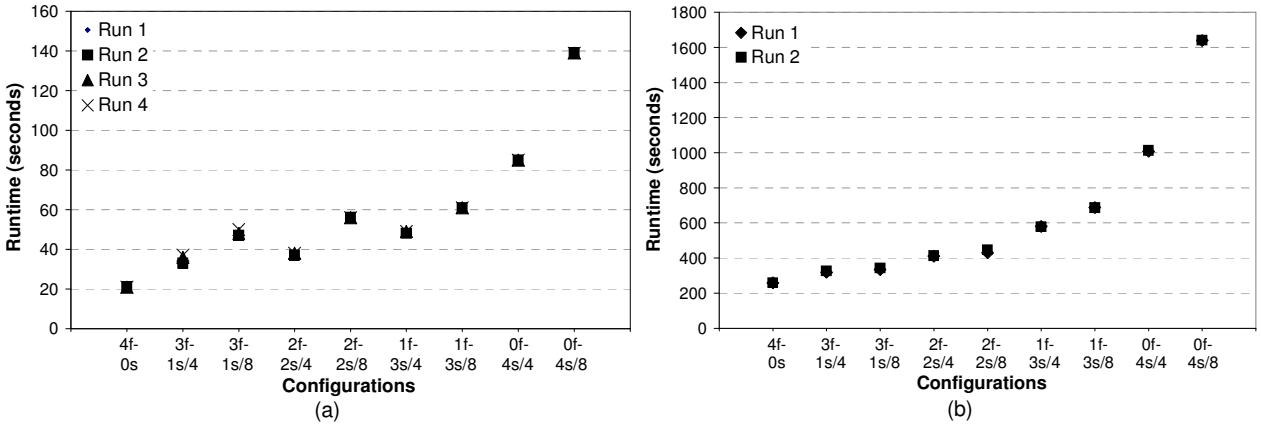

Figure 9. Execution times for multiple runs. (a) H.264 multithreaded media encoding. (b) PMAKE.

tion, while dynamic mode assigns constant chunk sizes. Most loops in SPEC OMP are statically parallelized and only a few have guided parallelization directives.

Intuitively, static parallelization should limit scalability on an asymmetric system, and we observe this in Figure 8(a) where the slowest processor limits performance. While all processors get equal work, they do not have the same performance. Guided parallelization performs better than static parallelization due to on-demand allocation of jobs. However, slow processors unaware of its capabilities might request chunks that are same size as fast processors. This leads to scalability issues with asymmetric systems. We next discuss why some benchmarks perform better in Figure 8(a) than expected.

`ammp` has seven large parallel tasks. Each task is a parallel for-loop over multiple iterations. For the runs in the figure, the OpenMP library, mapped two iterations each to the two fast processors, and one iteration each to the two slow processors. Such an assignment makes the application susceptible to scalability issues when it executes on an asymmetric system because the mapping library is unaware of the system's asymmetry, and could easily map them in a different order. This would affect performance.

`galgel` has 30 parallel regions with short loop bodies. Three most commonly executed parallel regions had a no-wait directive, which means the application need not wait at the loop end. This allows faster processors to continue without waiting for slower processors. Many loops in `galgel` use guided parallelization.

To fix the scalability issues, we focus on the application's work assignment. To prevent scheduling decisions from affecting performance, we use dynamic parallelization for all loops in all benchmarks. However, for loops executing a large number of iterations, we chose a large chunk size to reduce allocation overhead. Figure 8(b) shows the runtimes with our modifications to the parallelization algorithm. These runtimes are higher than Figure 8(a) because our modifications were not focused on performance tuning but on eliminating asymmetry effects. Hence, these absolute runtime numbers are not directly comparable. We notice significant performance benefits of asymmetry—runtimes of 2f-2s/8 are near those of 4f-0s. This is due to efficient dynamic scheduling of loops. Asymmetric configurations perform better than the midpoints of 4f-0s and 0f-4s/8, clearly indicating that asymmetric systems can be effective for power/performance efficiency.

3.6 H.264 multithreaded media encoding

The H.264 workload [2, 10] is a high-performance multi-threaded version of the H.264 video encoder [7], a new video coding standard providing superior compression while preserving image quality. The standard divides a video picture, called frame, into numerous small blocks, called macro-blocks. These blocks are processed, and then re-synchronized and re-arranged for final delivery.

The application has five concurrent threads. A main thread handles image pre-processing and post-processing, and consumes 2-5% of CPU time. Pre-processing involves reading raw image data and setting up parameters, and post-processing checks encoding status, generates the output bit stream, and performs image interpolation and reconstruction activity. Pre-processing and post-processing are sequentially handled for correctness.

The encoding process is parallel and involves operations on macro-blocks. This deals with motion estimation and selecting the optimal coding mode for each macro-block. A spatial dependence exists among various macro-blocks within a frame: macro-block encoding occurs only after its adjacent (upper left and right in an image), blocks are encoded. The application exploits temporal parallelism: parallel encoding of frames occurs by estimating and compensating predicted frames.

Figure 9(a) shows execution time of H.264 for four different runs on various processor configurations. All configurations show stability across multiple runs, and are predictably scalable. The application has abundant parallelism: there is significant slowdown going from 4f-0s to

Table 1 Results summary

Application	Section	Class	Is performance predictable?	Is scalability predictable?
SPECjbb	Section 3.1	MRTE	No *(Yes with asymmetry aware kernel)*	Yes
SPECjAppServer	Section 3.2	MRTE	Yes	Yes
TPC-H	Section 3.3	Database	No *(Yes, if application changes)*	Yes
Apache	Section 3.4	Web server	No *(Yes with asymmetry aware kernel)*	Yes
Zeus	Section 3.4	Web server	No	Yes
SPEC OMP	Section 3.5	Scientific	Sometimes *(Yes with application change)*	No *(Yes with application change)*
H.264	Section 3.6	Multimedia	Yes	Yes *(asymmetry helps perf.)*
PMAKE	Section 3.7	Development	Yes	Yes *(asymmetry helps perf.)*

3f-1s/8; replacing one fast core with a slow core brings down the runtime significantly since all threads need to wait for the slower core. In the 2f-2s/8 and 1f-3s/8 configurations, the slowdowns are smaller since the fast processors do more work; the slower processors do less work. Going from a 1f-3s/8 to a 0f-4s/8 shows a significant drop since the 0f-4s/8 does not have any fast processor to take over work.

This application demonstrates how some performance asymmetry is good for performance. The slowdowns induced by adding some asymmetry is significantly lesser than a system in which all cores are slow. A 1f-3s/8 system (performance asymmetric) is significantly better than a 0f-4s/4 or a 0f-4s/8 (symmetric, all slow cores) because of the availability of one fast core.

3.7 PMAKE

The PMAKE application performs a parallel compilation of the Linux kernel (~7900 C files). We run PMAKE with "make –j4" indicating the number of processors in the system.

PMAKE shows stable and scalable speedups for all configurations (Figure 9(b)). Similar to the H.264 application, one fast processor improves performance as compared to when all processors are slow. Having one fast processor can significantly improve performance because it can provide high utilization when necessary.

4. Results summary

Table 1 qualitatively summarizes our results and Figure 10 quantitatively shows the predictability and scalability of the various applications. The figure also shows error bars for each configuration. These error bars represent the performance variation when the benchmark is run multiple times on the same configuration. All speedups are over the corresponding 0f-4s/8 configuration to show scalability. We now discuss the key points of the study below.

1. **Asymmetry adversely affects performance predictability of shared-memory workloads**.

SPECjbb, a managed runtime (MRTE) server, displays significant performance instability in the presence of asymmetry, and the underlying virtual machine and garbage collector can exacerbate this instability. Other servers also show similar instability (query processing in DB2/TPC-H, and Apache and Zeus web servers).

Applications with tight coupling among different threads (e.g., SPEC OMP), display stability but provide poor scalability on asymmetric systems. This is because applications and optimizers assume all processors provide equal performance. In these applications, the slowest core forces faster cores to idle, waiting for the slowest core to complete its task.

2. **Making the operating system kernel asymmetry-aware helps eliminate unpredictability in some applications. In others, the application also needs to be asymmetry-aware.**

The performance unpredictability in SPECjbb and Apache was eliminated after we made the operating systems scheduler aware of the performance asymmetry in the system. However, other applications needed changes in their structure.

We approximate application changes in DB2/TPC-H by varying the degree of parallelization and optimization levels of TPC-H queries. We saw instability disappear as we reduced the degree of query optimization significantly. This suggests the role of application optimizers in ensuring stability under performance asymmetric conditions. We changed the load balancing OpenMP directives in SPEC OMP programs. Using the dynamic parallelization directives helped eliminate unpredictability.

3. **An asymmetric chip multiprocessor is better than a chip multiprocessor where all cores are slow.**

We observe the benefit of having a fast core, specifically in executing serial portions of multi-threaded programs. For example, in PMAKE and SPEC OMP, a configuration with two fast and two slow cores does better than the expected midpoint performance between

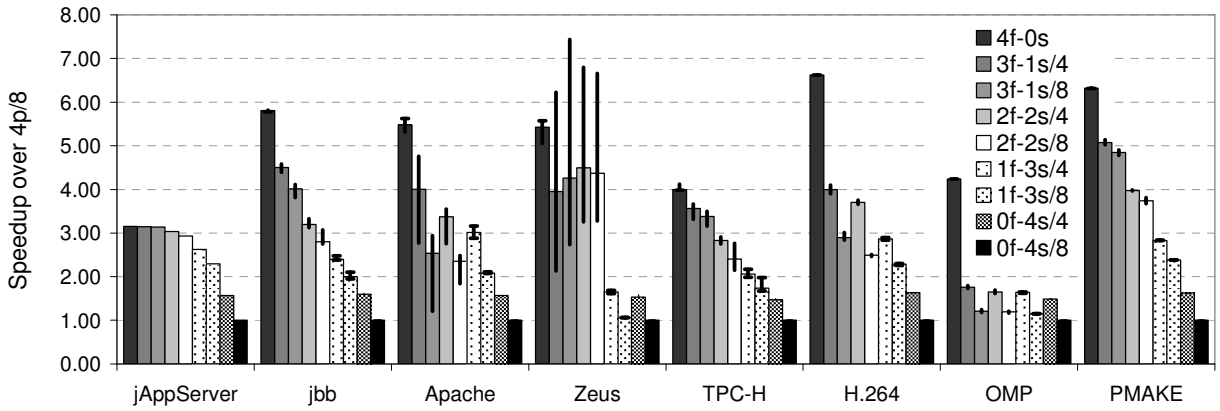

Figure 10. Performance predictability and scalability for all benchmarks. For each benchmark, 9 configurations are shown. The left-most bar (4f-0s) and the two right-most bars (0f-4s/4, 0f-4s/8) are the symmetric configurations. Speedups (y-axis) are normalized to the 0f-4s/8 configuration. Total computation power of the system decreases as we go from left (4f-0s) to right (0f-4s/8) for each benchmark. The benchmark performance as total computation power is varied (scalability) can be seen by directly comparing the bars for each benchmark. SPEC OMP and H.264 show that performance is limited by the slowest core in the system. The error bars show the performance variance over multiple runs for the benchmark on a given configuration. SPECjbb, Apache (light load), Zeus (light load), and TPC-H show significant variance for the asymmetric configurations (2nd through 7th bar for each benchmark). The symmetric configurations for all benchmarks do not show any variability.

that of a system with four fast cores and a system with four slow cores.

4. Mechanisms for exchanging asymmetry information between hardware and software, and application design methods tolerant of asymmetry need investigation.

While the instability introduced due to performance asymmetry varies across workloads, the existence of such instability makes it necessary for researchers to rethink the hardware and software interface and structuring of multi-threaded software. Exposing the relative performance of processors in a system to the operating system and software scheduler may be sufficient, and absolute information of each processor's performance may not be necessary.

We show evidence of the benefits of fine-grained threading to help alleviate the problem due to asymmetry for the Apache webserver. Providing numerous short-running threads allows the kernel to cope with asymmetry. However, the overhead associated with managing many threads may become large.

5. Related work

Enslow [3] presented a survey of multiprocessor organizations and predicted future trend of asymmetric processing, with processors dedicated to a hierarchy of functions. Withington [21] called this organization a "poly-processor." Miled et al. proposed a heterogeneous hierarchical organization, called HPAM [14], and studied instruction and data temporal locality of statically parallelized benchmarks. Figueiredo et al. [4] proposed a lelized benchmarks. Figueiredo et al. [4] proposed a similar organization for distributed shared memory systems. Bender et al. [1] proposed a scheduler for scheduling parallel programs on heterogeneous multiprocessors, and our kernel scheduling algorithm uses this prior work.

The CDC6700 was an asymmetric setup of two processors—the CDC 6400 and CDC 6600 [15].

Researchers have investigated single-chip systems with multiple asymmetric cores. Kumar et al. [9] demonstrated performance such a system's benefits by using simulation and SPEC CPU2000 benchmarks. They used phase and execution profile information of these single-thread applications to schedule optimally. Kumar et al. [8] identified asymmetry as beneficial for power reduction. A program schedules on a core with the best performance to power ratio. Grochowski et al. [5] and Morad et al. [17] studied the usefulness of such cores for saving energy and improving throughput.

Moncrieff et al. [16] and Menasce et al. [13] analytically studied tradeoffs of fast and slow processors in heterogeneous systems. They observe that a system with many slow and few fast processors are cost and performance effective. Liu et al. [12] studied optimal scheduling of independent programs on a pre-emptive heterogeneous multiprocessor system. Miller [15] presented scheduling algorithm for an asymmetric system called *Single Architecture Heterogeneous Multiprocessor* or SAHM, which did not support multi-programming.

6. Concluding remarks

Predictability and scalability of performance are important for successful system deployment. We have presented

a detailed study of the behavior of commercial workloads running on a multicore system where individual cores have different performance. We observe that such asymmetry in performance can have unintended negative impact on workloads, making their performance difficult to predict. This occurs because software developers assume all cores provide equal performance, and thus do not worry about the interactions of core performance with the application algorithm structure. We observe that in workloads that consider compute capability during runtime (e.g., SPECjAppServer uses performance feedback) the impact of performance asymmetry is non-existent. This suggests robust application designs that can adjust dynamically to the varying compute power of the system.

We demonstrated how exposing performance asymmetry to the operating system kernel, and using this information to make scheduling decisions, eliminated unpredictability on numerous applications. However, in other applications (e.g., TPC-H and SPEC OMP), this was insufficient. The application structure itself needed to change to adjust to the asymmetry.

We conclude that some degree of performance asymmetry is beneficial. This is because all applications, whether multi-threaded or single-threaded, have serial portions, and providing a high-performance core helps speed these serial portions. We conjecture that to eliminate unintended interactions between applications and performance asymmetry, the compute power from the high-performance core should be a small fraction of the total compute power of the system.

We argue that computer architects should consider the implications of multicore proposals on application behavior, and that the developers must design applications that are robust enough to dynamically deal with changing compute power.

Acknowledgements

We thank Guri Sohi for comments on earlier drafts of the paper. We thank Kai Ming Chan, Yen-Kuang Chen, Kingsum Chow, William Clifford, Kshitij Doshi, Shih-Lien Lu, and Hamesh Patel for assistance in setting up the workloads. This work was supported in part by National Science Foundation grants CCR-0311572 and EIA-0071924, and donations from Intel Corporation.

References

[1] M. A. Bender and M. O. Rabin. Scheduling Cilk Multithreaded Parallel Programs on Processors of Different Speeds. In *Proceedings of the 12th Annual Symposium on Parallel Algorithms and Architectures*, July 2000.

[2] Y.-K. Chen, X. Tian, S. Ge, and M. Girkar. Towards Efficient Multi-Level Threading of H.264 Encoder on Intel Hyper-Threading Architectures. In *Proceedings of the 18th Annual International Symposium on Parallel and Distributed Symposium*, April 2004.

[3] P. Enslow. Multiprocessor Organization Survey. *ACM Computing Survey*, 9(1), 1977.

[4] R. Figueiredo and J. Fortes. Impact of Heterogeneity on DSM Performance. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, February 2000.

[5] E. Grochowski, R. Ronen, J. Shen, and H. Wang. Best of Both Latency and Throughput. In *Proceedings of the International Conference on Computer Design*, October 2004.

[6] Intel, *System Programming Guide*, vol. 3: Intel Corporation, 2004.

[7] ITU, *Advanced Video Coding for General Audiovisual Services - Recommendation H.264*: International Telecommunication Union - ITU, May 2003.

[8] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, December 2003.

[9] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[10] E. Li and Y.-K. Chen. Implementation of H.264 Encoder on General-Purpose Processors with Hyper-Threading Technology. In *Proceeedings of SPIE Visual Communications and Image Processing*, January 2004.

[11] K. Li. The Case for Asymmetric Multiprocessor Architecture. In *ACM Workshop on General Purpose Computing on Graphics Processors*, 2004.

[12] J. Liu and A.-T. Yang. Optimal Scheduling of Independent Tasks on Heterogeneous Computing Systems. In *Proceedings of the 1974 Annual Conference*, 1974.

[13] D. Menasce and V. Almeida. Cost-Performance Analysis of Heterogeneity in Supercomputer Architectures. In *Proceedings of the 4th International Conference on Supercomputing*, June 1990.

[14] Z. B. Miled and J. Fortes. A Heterogeneous Hierarchical Solution to Cost Efficient High Performance Computing. In *Proceedings of the 8th International Symposium of Parallel and Distributed Processing*, October 1996.

[15] L. J. Miller. A Heterogeneous Multiprocessor Design and the Distributed Scheduling of Its Task Group Workload. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, May 1982.

[16] D. Moncrieff, R. E. Overill, and S. Wilson. Heterogeneous Computing Machines and Amdahl's Law. *Parallel Computing*, 22(3), 1996.

[17] T. Morad, U. Weiser, and A. Kolodny, ACCMP - Asymmetric Cluster Chip Multi-Processing. CCIT Technical Report #488, 2004.

[18] OpenMP Architecture Review Board OpenMP Specifications for Fortran/C/C++ Version 2.0, 2002

[19] Standard Performance Evaluation Corporation http://www.spec.org/

[20] Transaction Processing Council http://www.tpc.org/

[21] F. Withington. Beyond 1984: A Technology Forecast. In *Datamation*, January 1975.