Computer Systems Laboratory
Cornell University, Ithaca, NY 14853

A Characterization of the PARSEC Benchmark Suite for CMP Design

Major Bhadauria, Vince Weaver and Sally A. McKee

Technical Report No. CSL-TR-2008-1052

September 2008

A Characterization of the PARSEC Benchmark Suite for CMP Design

Major Bhadauria and Vince Weaver and Sally A. McKee

Computer Systems Lab
School of Electrical and Computer Engineering
Cornell University

E-mail: {major | vince | sam}@csl.cornell.edu

Abstract

The shared-memory, multi-threaded PARSEC benchmark suite is intended to represent emerging software workloads for future systems. It is specifically intended for use by both industry and academia as a tool for testing new Chip Multiprocessor (CMP) designs. We analyze the suite in detail and identify bottlenecks using hardware performance counters. We take a systems-level approach, with an emphasis on architectural design space features that vary with CMPs. We narrow the architectural design space, and list several areas worth investigating for improved benchmark performance.

1 Introduction

Architects have transitioned designs to a multi-core architecture, consisting of multiple processors on a single chip (CMP). To leverage this design change, programs are multi-threaded to utilize improvements in computing hardware. The recently released PARSEC benchmark suite [4] claims to represent the current and future set of applications that multi-processing hardware will be used for, and is used to drive future CMP research. If this suite is used to evaluate the performance of future hardware, it is critical to identify what hardware resources are important for the performance and design, and which are irrelevant. Unlike other multi-threaded shared memory benchmark suites, PARSEC is not a high performance computing (HPC) benchmark suite like NAS [2], SPEC 2006 [14], or SPLASH-2 [15], and characterizations made on those suites are not valid here [3].

Before using benchmarks to evaluate proposed future hardware, it is important to consider how current hardware fares. By identifying both the successes and imperfections with current solutions we hope to accelerate development of future research for power efficient, scalable CMP design. Power envelope and delay are currently

first-order design requirements, which can be constrained by system level choices such as core interconnect, chip communication (IO), and caches. These constraints are different than the core micro-architectural related ones of the previous era. This study is useful for the academic community, as it allows others to avoid time-consuming system-level simulation.

Our characterization on real hardware shows PARSEC has high temporal locality and does not significantly depend on memory for performance gains. PARSEC performance foreshadows revisiting previous generation architectures where memory speed was of lesser concern, and design was concentrated on faster processing elements and cores.

We investigate several features inherent in current CMP designs, chiefly memory speeds, cache sizes, bus communication traffic, and SMT performance. We also examine current micro-architectural resource bottlenecks, performance effects of using out of order execution, performance scaling with increasing threads, and using a 64-bit ISA over a 32-bit one. We also examine software instruction count and workload balancing across cores.

The contributions of this work are threefold:

- We show what features of current and future computing architectures are not necessary for high performance.
- We pinpoint sources of performance bottlenecks.
- We suggest specific architecture improvements and areas worth investigating for improving performance.

2 Characterization Metrics

Researchers have previously performed a detailed study of the PARSEC suite using functional simulation [4]. We gather additional results, using real hardware to cover metrics that cannot be obtained via non-cycle accurate simulation. We use cycle-accurate simulation for metrics that cannot be collected with the hardware available. Iyer et al. [8] explore resource bottlenecks for commercial Java servers, examining tradeoffs of cores, caches and memory. Alam et al. [1] characterize performance of scientific workloads on multi-core processors. These studies investigate special-purpose workloads; we investigate a broader-range of workloads to determine which CMP architectural features are most useful for general purpose CMPs.

2.1 Cache Behavior

We investigate overall cache miss rates as well as L2 cache locality. We find cache efficiency by examining the amount of evicted data compared to total accesses. If the eviction rate is lower than the miss rate, then misses are due to cache invalidations, not capacity. Therefore in this case increasing cache size would not improve performance. We also examine snoop traffic, which occurs when a processor checks neighboring processors for data it needs. Snoop requests can saturate the communication bus between cores. We investigate data bus traffic caused by writeback and core-to-core transfers. If there is sufficient communication or data traffic, then wider busses or independent crossbars might be needed for maximum performance.

2.2 DRAM Latency and Bandwidth Dependence

We investigate the effects of memory speed and bandwidth on multi-thread performance. Memory chip frequencies have previously been shown to be a significant bottleneck for modern systems [9]. Modern designs have increased these frequencies; we investigate the impact of these improvements.

DRAMs are a multi-dimensional structure consisting of banks, rows and columns. Maximum theoretical bandwidth is only obtained when data are read from the same open row within a bank, or when data is interleaved across different banks. We examine how often this theoretical maximum can actually be achieved, since it assumes every memory access is a page hit. A memory access can result in a page miss (when the row to be accessed is not already open) or a page conflict (when the current row needs to be closed).

We examine halving memory bandwidth, reducing memory speed, and reducing command time latencies. Modern machines have twice the memory channels of previous generations, which requires twice as many wire traces on the motherboard and increases the number of IO pins on the CMP substrate. This leads to increased system cost and complexity. Such complicated bandwidth solutions might be superfluous depending on the number of processors on chip, the target market, and achieved performance gains.

2.3 Throughput

We define chip traffic as the number of main memory accesses for every retired instruction. The ratio of committed instructions to main memory accesses illustrates the program's reliance on memory. Although modern systems employ prefetchers, the latency to main memory cannot always be masked. The chip traffic ratio quantifies how reliant the program is on memory.

2.4 Thread Scaling

Programmers might ideally expect program performance to scale linearly with increasing thread counts (provided there is little data contention). We examine how well the PARSEC suite scales on a variety of hardware. We investigate the implications of previous assumptions based on synchronizing overhead, un-parallelized code sections, and redundant computations. While previous work [4] found upperbound increases to be linear based on number of retired instructions per thread, we investigate what the actual increases are on real hardware. In addition to thread scaling, we examine performance scaling via simultaneous multi-threading (SMT), where multiple threads are run on a single processor.

2.5 Micro-architecture Performance

We investigate micro-architecture bottlenecks: how often the processor stalls due to lack of resources. Bottlenecks come from reorder buffers (ROB), reservation stations (RS), load store queues (LSQ) and floating point units (FPU). We also examine stalls due to mispredicted branches. We identify the significant system bottlenecks, and show which areas warrant further research for improvement. The ROB is critical for out of order (OoO) processors to achieve a higher level of instruction level parallelism (ILP). However, in isolation, full ROB or RS queues may not be indicative of a problem, as instructions may have stalled due to memory latency, lack of execution units, or lack of ILP. FPU stalls may indicate a dependence on FLOPS, and a need for more or faster FPUs. For completeness, we examine the performance and cache effects of executing 64-bit code versus 32-bit code, due to significant previous wide-spread investment and use of 32-bit hardware.

Name	Description
blackscholes	calculates portfolio price using Black-Scholes PDE
bodytrack	computer vision, tracks a 3D pose of a markerless human body
canneal	synthetic chip design, routing
dedup	pipelined compression kernel
facesim	physics simulation, models a human face
ferret	pipelined audio, image and video searches
fluidanimate	physics simulation, animation of fluids
freqmine	data mining application
streamcluster	kernel that solves the online clustering problem
swaptions	computes portfolio prices using Monte-Carlo simulation
vips	image processing, image transformations
x264	H.264 video encoder

Table 1. PARSEC Benchmark Overview

3 Setup

The PARSEC multi-threaded benchmark suite consists of 12 benchmarks from Princeton, Intel and Stanford. PARSEC represents a diverse set of commercial and emerging workloads. We briefly describe them in Table 1; they are explained in-depth in the original PARSEC technical report [4].

We evaluate the multi-threaded shared memory PARSEC suite v1.0. We compile the benchmarks with the GCC 4.2 C and C++ compilers on Linux kernel 2.6.22.18 or later. Our Sun machine uses Solaris and the SUN C compiler. We compile the benchmarks on a 64-bit machine in 64-bit mode using statically linked libraries. For 32-bit runs, we use 32-bit machines to generate statically linked 32-bit binaries. We use the full native input sets, as we are executing on actual hardware.

We test a wide variety of system configurations that vary extensively in memory hierarchy, system frequency, and microarchitecture. Table 2 lists the machines and relevant architectural parameters. Athlon1 is a traditional single core processor. PentiumD4SMT is dual-core with private L1 and L2 caches. Phenom4 is a quad-core CMP with private L1 and L2 caches and a shared 2 MB L3 cache. Conroe2 is a dual-core CMP with private L1 and shared L2 caches. Conroe4 is two Conroe2 CMPs connected as an SMP in a single package. The Xeon8 is like the Conroe4, but using FB-DIMM memory instead of DDR. NiagaraII-8 is a 8-core CMP with private L1 and a shared L2 cache, also using FB-DIMM memory instead of DDR. The Phenom4 CMP has a two channel integrated memory controller, and the NiagaraII-8 has four integrated memory controllers. The NiagaraII-8 supports up to 8 threads per core, and the PentiumD4SMT supports up to 2 threads per core.

We use hardware performance counters (both per-processor and system wide) to gather data. We use the `pfmon`

Name	Architecture	Frequency	Processors	L1 DCache	L2 Cache	FSB (MHz)
Athlon1	x86 32-bit ISA	1.8 GHz	1	64 KB	256 KB	400
PentiumD4SMT	x86 64-bit ISA	3.46 GHz	4 (8 SMT)	16 KB	2 MB	1066
Phenom4	x86 64-bit ISA	2.2 GHz	4	64 KB	512KB	1800
Conroe4	x86 64-bit ISA	2.4 GHz	4	64 KB	4 MB	1066
Conroe2	x86 64-bit ISA	1.8-2.7 GHz	2	64 KB	2 MB	800
NiagaraII-8	x86 64-bit ISA	1.15 GHz	8 (64 SMT)	8 KB	4 MB	800
Xeon8	x86 64-bit ISA	2.32 GHz	8	64 KB	4 MB	1066

Table 2. Machine Parameters

tool from the perfmon2 performance counter package [6] to read the counter values, and the Linux `time` command to gather actual elapsed program execution time. We find little variation between multiple runs for most benchmarks, and ignore differences of a few percent as operating system noise when comparing architectural choices. Certain programs perform markedly different when running locally versus having the executables and input fils served over NFS: `bodytrack`, `dedup`, `facesim`, `vips` and `x264`. `vips` and `x264` exhibit significant variation in execution time across multiple runs. We run these benchmarks five times and take the average time. The bandwidth on the network and HDD (when run locally in separate runs) play a large role in performance.

We use the cycle-accurate simulator PTLsim [16] for evaluating gains from OoO cores over in-order ones. PTLsim models a 3-wide issue AMD K8 processor, and has been validated. It incorporates a 16 KB Gshare branch predictor, three eight entry integer queues, and one 36 entry FP queue. The memory hierarchy consists of a 64 KB 2-way L1 Dcache, 1 MB 16-way L2 cache and main memory with respective latencies of one, ten and 112 assuming a 2.2 GHz clock frequency. Due to simulator constraints, the small input sets are used and two benchmarks (ferret and x264) that could not run are omitted.

4 Evaluation

We investigate architectural performance across a variety of hardware configurations. For `fluidanimate`, we omit three, five, six, and seven thread configuration, as the benchmark only runs with power-of-two threads. The data for `facesim` at 5 and 7 thread counts are missing, as the benchmark suite does not have input sets for these thread counts.

4.1 Cache Performance

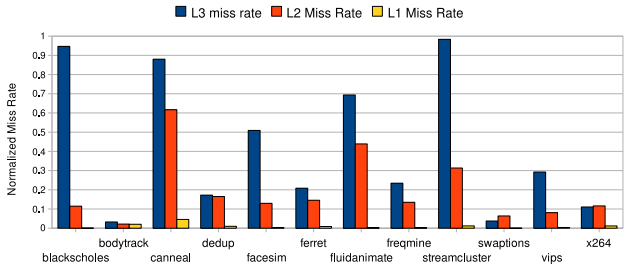

Figure 1. Phenom4 4-Thread L1,L2,L3 Cache Performance

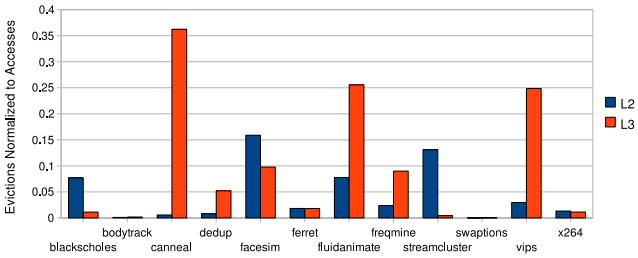

Figure 2. Phenom4 4-Thread L2, L3 Data Evictions

We examine the waterfall effect of data accesses on a 3-level data cache hierarchy using the Phenom4 platform. We separately examine miss-rates for L2 caches from 256 KB to 4 MB. Figure 1 graphs miss rates normalized to total data cache accesses (this does not include snoops, instruction cache accesses, hardware prefetches, or TLB lookups). All benchmarks have very low L1 cache miss rates, the highest being 5% for `canneal` and 2% for `bodytrack`. These miss rates have a trickle-down effect for `canneal`, resulting in high L2 and L3 cache misses. Most of the L2 and L3 cache misses appear to be cold and capacity misses, where increasing the cache size will not help. Only prefetching can reduce these cold misses. Although prefetching can increase capacity misses, the high temporal locality of programs indicates capacity misses would be negligible. The L3 cache is not useful here, since it is used to store data that is evicted from the L2 cache. Since this data is not accessed again until much later, the L3 cache has a very high miss rate as the temporal locality of the programs do not extend to the L3. For the L3 to effectively capture the L2 cache misses, it would have to be several orders of magnitude larger, based on prior studies [4]. We verify the high temporal locality of the L2, which would cause the artifact we see with the L3 cache, by examining the number of L2 line evictions compared to the L2 accesses.

Figure 2 graphs the data evictions from the cache normalized to total cache accesses for four threads. The evictions are lower than the cache misses, the L2 average miss rate was 20%, but the eviction rate is only 5%. This is because the data that is being brought into the cache is not evicting valid data, but cache lines that are invalidated from coherence checks. If the data that was requested was originally evicted due to invalidation from another processor, increasing cache size will not help, although reduced line sizes might if it is a case of false

sharing. Therefore, line fills may not require going off-chip to main memory. Since only 5% of all L2 accesses result in cache-line evictions, the L2 cache does have high temporal locality.

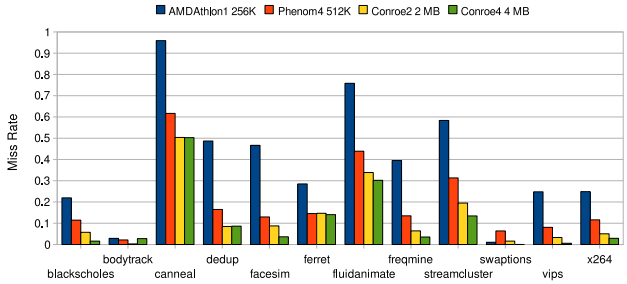

Figure 3. Cache L2 Miss Rate scaling from 256KB to 4 MB on Various Machines

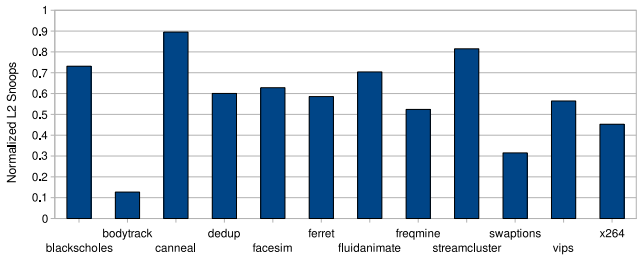

Figure 4. L2 Snoops Normalized to Total L2 Tag Accesses - Phenom4

We verify that doubling cache sizes does not help by comparing the cache performance going from Conroe2 (2 MB cache) to Conroe4 (4 MB cache). Figure 3 graphs cache performance for the shared L2, going from 2 MB to a 4 MB cache, and also includes data from other machines for perspective (all running with just one thread). `canneal`, `dedup`, `ferret`, and `fluidanimate` have approximately the same amount of cache misses. This is due to their working sets being so large that the cores miss regardless of cache size (up to 4 MB). On average, doubling the L2 cache size reduces the L2 miss rate to 11% from the original 13% of the 2 MB L2 (a 2% reduction). Looking at tradeoff between cache size and processors, 512-1024 KB seems the best size for area tradeoff. The extra area saved can be better used adding processing cores. Alternatively, an L2 hardware prefetcher that can effectively predict the next few lines should be sufficient to mask latency. We examine cache snoops, but not other related metrics such as number of lines being shared, writes to the same line, etc. This has been comprehensively covered at the functional level in [4]. Figure 4 graphs the snoops on the L2 cache lines for Phenom4, running with four threads. The number of L2 cache snoops are normalized to total L2 tag accesses. We choose the Phenom4 because it has the largest private caches (512 KB), compared to its Intel and Sun platform counterparts. Thereby it should have the least amount of snoops due to cache misses. The majority of the cache accesses are comprised of snoops. The clear exception being `bodytrack`, which has a small working set size (512 KB) and medium data exchange [4]. Clearly, future CMPs will need to tackle the issue of snoop traffic, as it consists of a significant amount of the L2 cache lookups. `canneal`, `dedup`, `ferret`, and `x264` have a high number of snoops due to

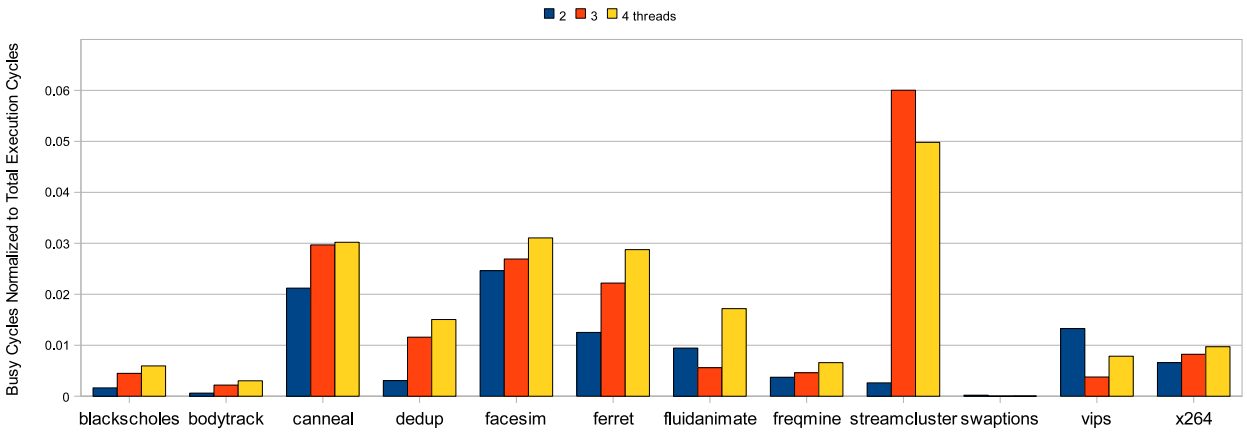

Figure 5. CPU Clock Cycles Data is Transferred on the Bus from the Core-Conroe4

the high amount of data exchange [4]. The other benchmarks which have a high number of L2 cache data accesses and snoops are due to the large working sets, and snoops performed to find data rather than for coherence. Unless caches are an order of magnitude larger, this will be a persistent issue for future CMPs. However, since only the tag portion of the L2 is accessed on a snoop, the power overhead might be negligible depending on the system architecture [10].

A metric related to snoops is the percentage of time the bus is being used to send data from the processor back to the shared cache or to another processor. This is useful for pinpointing how busy the bus is relative to overall clock-cycles. Figure 5 graphs the clock cycles the data bus is busy transferring data from the private cache back to the shared cache, or to other cores that need it, normalized to total executed clock cycles. Most benchmarks do not spend a lot of cycles transferring data between cores, the maximum being `streamcluster` which keeps the data bus busy 5% of the time for the four thread case. This shows there is capacity available for prefetching data from one core to another, especially for programs that involve a lot of swapping operations. However, traffic can increase with threads, and this data does not include cycles committed for communicating snoop checks.

4.2 DRAM Memory Accesses and Performance Sensitivity

We examine the DRAM memory access patterns of PARSEC programs at varying thread counts. We are interested in DRAM open page hits, which are DRAM accesses that occur on an already open row. Open page

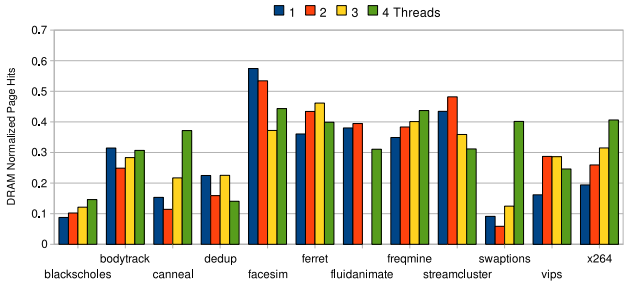

Figure 6. DRAM Page Hit Rates

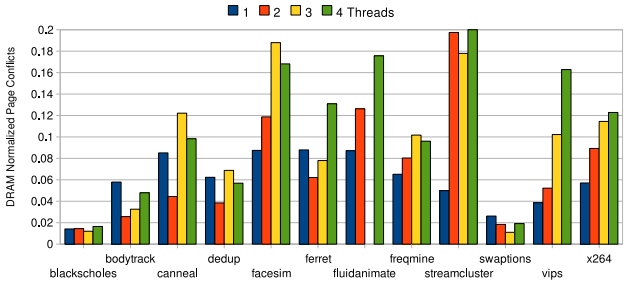

Figure 7. DRAM Page Conflict Rates

hits are a good indicator of how often the available bandwidth is being used. Figure 6 shows DRAM open page hit rates, normalized to total DRAM accesses on Phenom4. DRAM page hits are quite low, with all benchmarks exhibiting less than 50% hit rate for the highest thread count. The benchmarks also show different trends with increasing threads. For example, `blackscholes`, `freqmine`, and `x264` show hit rates increasing with threads. In contrast, benchmarks `ferret`, `fluidanimate`, `streamcluster`, and `vips` show parabolic curves with increasing threads. Hit rates that increase with increasing threads occur when threads are accessing the same open rows. However, should their memory requirements not overlap, increasing threads would decrease DRAM hit rates, since the memory accesses correspond to different rows within a bank. A parabolic curve can occur when low thread counts exhibit the former scenario, but with increasing threads the memory requirements exceeds a single open row and spans multiple ones, leading to higher DRAM page conflicts.

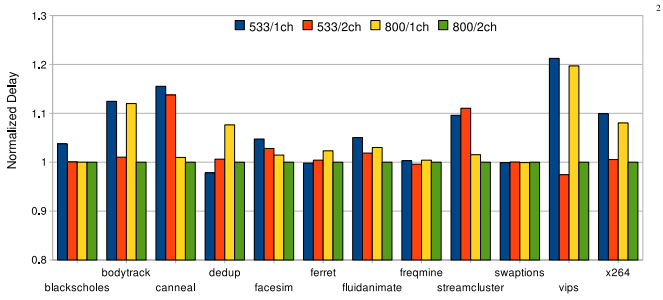

Figure 8. Phenom4 Single Thread Performance With Varying Memory Parameters

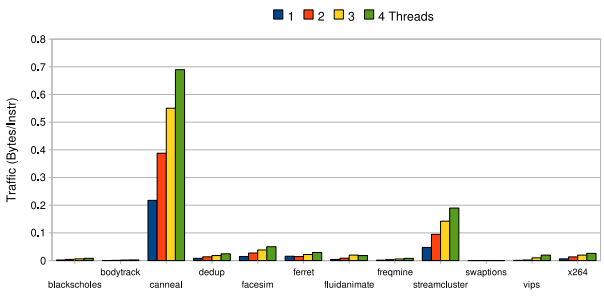

Figure 9. Phenom4 Thread Traffic (Bytes/insn) at Varying Thread Counts

We confirm this hypothesis by examining how DRAM conflicts change with increasing threads. A DRAM page conflict occurs when a request for a memory access falls on a row other than the one currently open. Figure 7 illustrates DRAM page conflicts normalized to total DRAM accesses on Phenom4. `blackscholes`, `freqmine`

and x264 have increasing DRAM page conflicts with increasing threads. This seems counter-intuitive, since those same benchmarks have increasing hit rates. One would assume the two characteristics would be inversely proportional, but conflicts are not the same as page misses. As accesses grow, hits increase, as they grow and expand past the currently open page, the number of page conflicts increase as well. The miss rates are not graphed, since they can be deduced from Figures 7 and 6.

The DRAM access patterns prove that the maximum theoretical bandwidth is rarely achieved. Low latency memory would be better than high bandwidth memory, since the maximum theoretical bandwidth can rarely be leveraged due to memory access patterns. We vary the memory speed, and bandwidth and examine their effect on performance. Figure 8 graphs single-thread benchmark performance on Phenom4 normalized to a 400 MHz FSB, dual-channel memory configuration. We examine single thread performance, since memory sensitive benchmarks might bottleneck at higher thread counts regardless of memory speed, masking the contention. The tested cases were 266 MHz FSB (533) in single (1ch) and dual-channel (2ch) modes, and 400 MHz FSB (800) also in 1ch and 2ch modes. These are speed and bandwidth reductions of 33% and 50% respectively. We keep clock-cycle latencies constant, thereby reducing clock speed increases command delay, such as opening, closing and reading from open pages. Results show benchmarks are dependent on different dimensions of memory for performance. Figure 8 shows that bodytrack, vips and x264 are sensitive to reductions in bandwidth. Halving the bandwidth results in an increase of delay by 12%, 21% and 10% respectively. canneal and streamcluster are sensitive to memory latency, suffering delay increases of 16% and 10%. Increased memory speeds and bandwidth add overhead to system cost by requiring higher speeds for the motherboard's FSB and the memory modules, as well as twice the number of traces on the motherboard for dual-channel memory. Our performance analysis shows we can tradeoff complexity and power (required for higher bandwidth and frequencies), with minimal performance degradation depending on benchmark.

Figure 9 graphs total memory traffic in bytes per instruction committed. Since bodytrack has little traffic, prefetching should sufficiently mask the memory gap if a low bandwidth system is employed. canneal and streamcluster require low-latency off-chip accesses, since we earlier saw they were more sensitive to latency than bandwidth. The other benchmarks have sufficiently low traffic demands at low thread counts to not be an

issue.

The PARSEC suite appears bound more by computation than memory, since our earlier reductions of memory speeds and bandwidth did not lead to linear degradation of performance. We verify this by examining the performance improvement achieved by scaling frequency, while keeping memory speeds fixed for the two thread case. We omit the performance graph for the Conroe2 system due to space constraints. However, increasing frequency by 33% (2.4 GHz) and 50% (2.7 GHz) results in average improvements of 34% and 50% respectively. While scaling memory speeds and bandwidth had little effect on performance, processor frequency has a direct and linear effect. Whether this changes at higher thread counts remains to be seen.

4.3 Thread Scaling Overheads

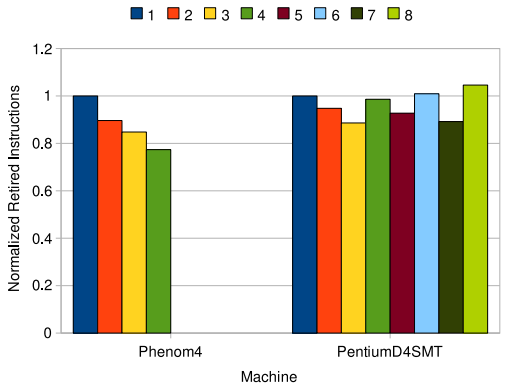

Figure 10. Total Retired Instructions for Canneal

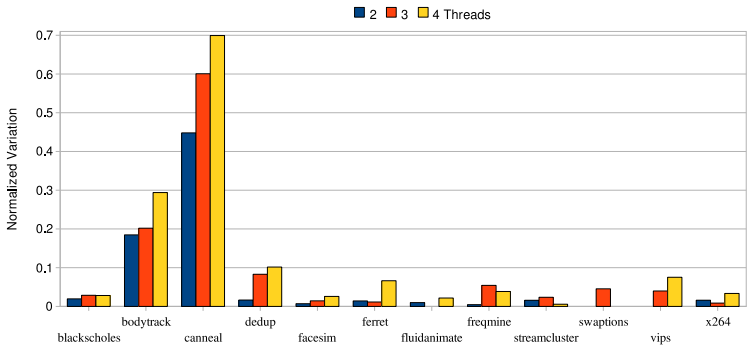

Figure 11. Instruction Variation Over Different Thread Counts

We investigate two metrics that affect multi-thread scalability: amount of serial execution, and the increased instruction overhead needed for multiple threads. We analyze instructions retired from one to four thread configurations, and found the count does not change for any of the benchmarks except for `canneal`. This shows that the PARSEC suite has little increased instruction overhead for parallelization. We graph the retired instructions for `canneal` in Figure 10 on Phenom4 and PentiumD4SMT normalized to the single-threaded case. There is a 5-10% reduction per thread increase on Phenom4. PentiumD4SMT results are unpredictable, with some thread combinations having fewer retired instructions, and others having more. These results are an artifact of the software and not an issue to be dealt at the hardware level.

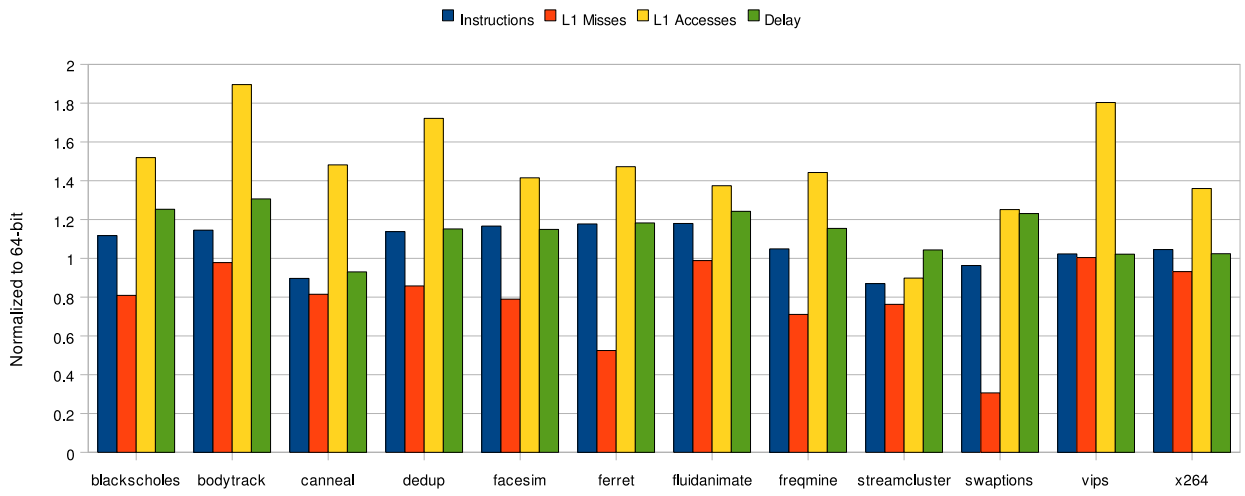

Figure 12. Effects of Executing 32-bit Versions

When a program has portions that are not parallelizable, there is only a single (master) thread that is running. This results in some cores executing more instructions than others, but the total number of instructions retired does not change. Figure 11 graphs variation between threads, normalizing the processor that retires the fewest instructions by the processor that retires the most. The higher the variation, the higher percentage of total instructions that are run by a single processor. There is some error with this technique since some serial portions can be running on one core and than migrate to another. We compare across multiple executions and machines and find variation to be negligible. `bodytrack` and `canneal` both have a large variation in retired instructions which grows with threads. As CMPs grow to more cores on chip, the serial portions of these benchmarks will bottleneck performance. One solution is to have a heterogenous architecture, where one processor executes at a higher frequency [13] or has a wider pipeline, so it can execute instructions faster than other processors which are idle at these points. This should not increase the power envelope of the chip since other cores are idle when the high speed core is active.

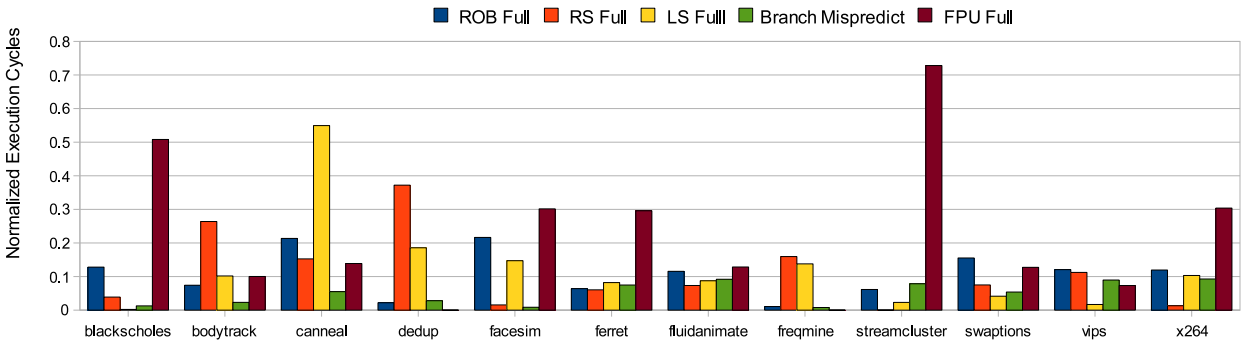

Figure 13. Resource Stalls for Phenom4 with 4 Threads

4.4 Micro-architecture Design Choices

64-bit instructions can require fewer operations to perform functions compared to 32-bit code, however it can also lead to increased cache misses due to longer pointer addresses and larger data structures. We are interested in the change in instructions, performance and effect on L1 caches. Figure 12 graphs the number of instructions retired, L1 misses, L1 accesses and delay normalized to the 64-bit code run on Phenom4 with four threads. Switching to 32-bit mode requires executing more instructions in most cases, the exceptions being `canneal` and `streamcluster`. On average there are 6% more instructions, 21% fewer L1 misses, 47% more L1 accesses, and 14% increase in delay. Although the L1 cache miss rate is much lower since the misses go down while the number of accesses go up, the increase in instructions being retired results in an increase in delay. Thereby execution units are not being effectively used, resulting in several operations which previously required one. The improvement in cache performance is not useful, since the L1 cache already had an almost 0% miss rate. `canneal` is the exception with a reduction in instructions, misses, and delay.

While the LSQ may fill up from longer memory latencies due to CMP size scaling, core metrics such as branch misprediction and full FPUs should remain unchanged. Figure 13 graphs clock cycle stalls due to lack of entries in the reorder buffer (ROB Full), reservation stations (RS Full), load/store buffers (LS Full), branch mispredictions and FPU backlog. `blackscholes` and `streamcluster` spend over 50% of the cycles stalled waiting for FPU, since 43% and 52% respectively of all operations in those programs are floating point [4]. FPU stalls

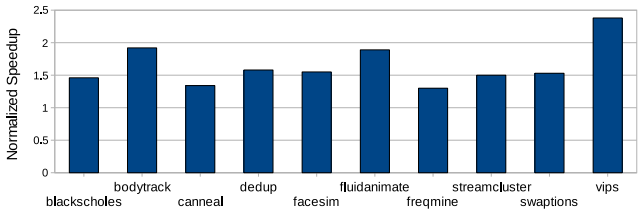

Figure 14. OoO Performance Improvement Over In-Order Single Thread

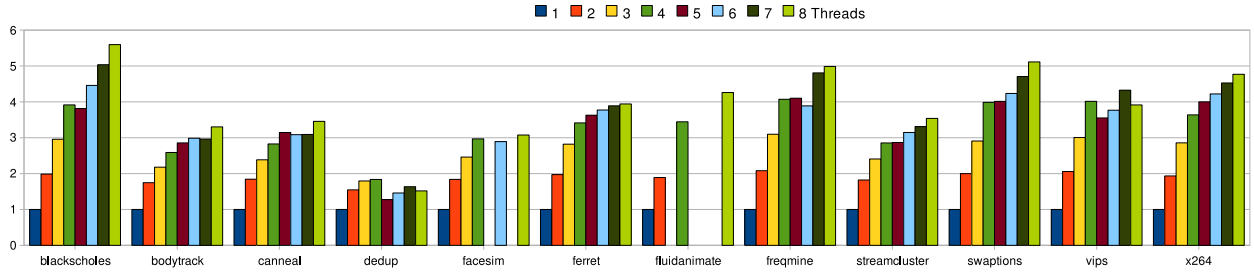

Figure 15. PentiumD4SMT 8-SMT Machine at 1-8 Threads (5-8 Virtual)

account for 30% of the stalls for more than half the floating point benchmarks (three PARSEC benchmarks have little or no floating point operations). This is surprising, since bandwidth usage was previously thought to be the most severe limitation of performance [4], but our earlier analysis of bandwidth at low thread counts shows this not to be the case.

When investigating the tradeoffs of OoO cores, we do not quantitatively analyze drawbacks of added die-area and power consumption since they are process technology specific. OoO scheduling can lead to power inefficiencies due to increased logic units and from the execution of mis-speculated instructions and load replays. For perspective, the AMD K8 processor with 128 KB of L1 cache and 512 KB of L2 has 68.5 million transistors. The Pentium 4 with 20 KB of L1 (12K trace cache and 8KB dcache) and 512 KB of L2 cache has 55 million transistors [5]. The recently released Intel Atom processor which is in-order with 32 KB of L1 and 512 KB of L2 has 47 million transistors [7]. While the reduction in transistor count is not significant between in-order and OoO, the power consumption is considerably lower (as has been the case with other in-order designs from ARM and Sun [11]). The Sun Niagara 2 with eight cores and eight SMT threads per core is 500 million transistors (with 4 MB of shared L2 across all cores).

The performance of an x86 OoO processor is normalized to one that has OoO scheduling disabled in Figure 14.

The performance improvement of OoO scheduling varies with benchmark, with improvements of 22% to 137%. However, depending on the power requirements and thread scaling achievable, higher performance might be possible by replacing a single OoO core by several in-order cores or implementing SMT support within a single core. Multi-threaded programs allow us to reduce instruction level parallelism for explicit thread level parallelism, reducing the burden on the hardware to extract parallelism at run-time. This can lead to improved power efficiencies assuming programs scale as well with threads as they do with OoO cores.

We test SMT performance on PentiumD4SMT (8 SMT) and Niagara8 (64 SMT), and compare improvements with the micro-architecture stalls on the AMD Phenom4 platform. Although the specific resource sizes differ between the AMD, Intel and Sun CMPs, the benchmarks can still stress the same areas across platforms. Threads can be stalled due to memory latencies, waiting for dependent instructions to finish executing, or thread synchronization. SMT can leverage these instances to achieve performance improvements by executing some threads while other threads are stalled. Figure 15 shows speedups of benchmarks as threads increase from one to eight on four physical processors. We ignore non-SMT performance for the moment, and address it in the next section. Benchmarks `blackscholes`, `bodytrack`, `fluidanimate`, `freqmine`, `swaptions`, and `x264` achieve significant performance gains with virtual cores. `freqmine` benefits because each core has a large amount stalls due to full LSQ. SMT gives `freqmine` a performance boost since other threads can be swapped into the core when one thread is pending for a memory access. `swaptions` does well with SMT due to excess architectural resources available for extra threads.

On the Intel architecture employed by PentiumD4SMT, architectural resources are either duplicated, partitioned, or shared. The caches are shared by each thread, and instruction pointer and thread state information is duplicated. However the ROB, LS buffers, and RS are dynamically partitioned between threads, so the SMT processor has half of these resources per thread compared to running a single thread [12]. This leads to reduced performance gains where these resources are a bottleneck, such as `canneal`, `dedup` and `facesim`.

`blackscholes` is an FP limited benchmark [4], as is `streamcluster` for large input dimensions; it is trying to minimize the distance between points, statically partitioning the data between threads. Due to the low ILP in these benchmarks, some FPUs are idle which can then be utilized by hyperthreading. In contrast, `canneal`

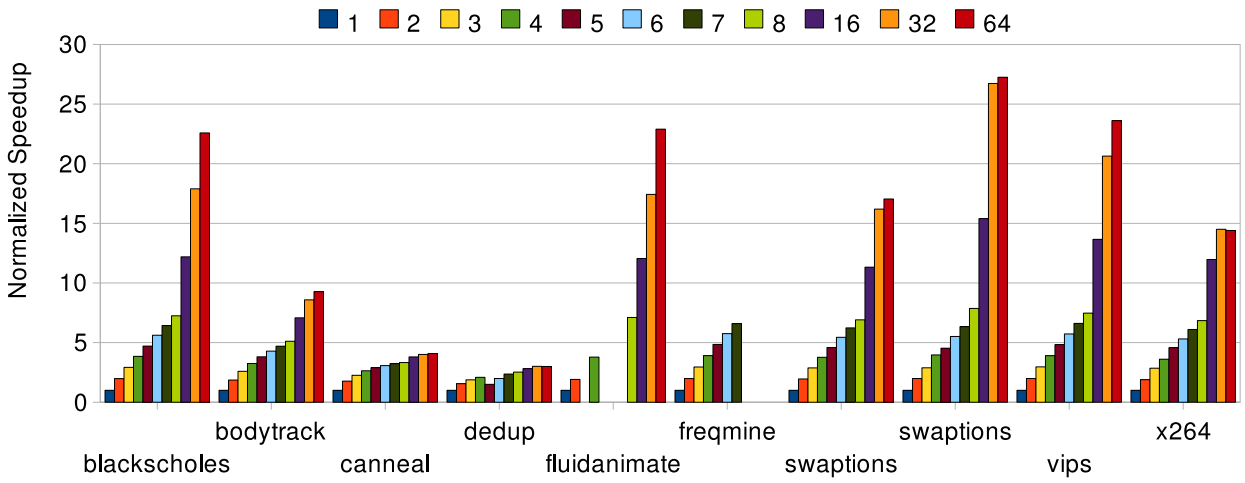

Figure 16. NiagaraII-8 Performance Scaling 1-64 Threads (9-64 Virtual)

performs more data swaps than actual arithmetic. While swaps are evaluated, locks are not held, changing execution path if one of the swap elements has been replaced by another thread within that time [4]. Unfortunately the LS buffer size is reduced when running multiple threads on the SMT, leading to sub-par performance gains with `canneal`. `dedup` is primarily an integer application, using the integer units for hash computations, checksums and compression. When the integer units are full, instructions queue up in the reservation stations. Once a sufficient backlog exists, the processor must stall to wait for the reservation stations to empty. Increasing threads does not help since the number of reservation station entries and integer units fail to increase in tandem. `x264`'s parallelization scheme contains dependencies between the current frame and the previous frames which are split across threads. When one thread is stalled due to FPU, it ends up slowing down other threads as well. It is recommended that more threads than cores are used to speedup this benchmark, and with multiple virtual cores (SMT) we find it does help. Based on improvements from SMT, and the code structure, OoO execution fails to leverage existing parallelism to the level that SMT achieves. Figure 16 shows results for the Niagara8 platform scaling from 1-64 threads. Leveraging SMT on the SUN platform results in performance trends similar to the PentiumD4SMT platform. Since each of the NiagaraII-8 physical cores only have two ALUs and one FPU shared among threads, there is contention for resources at high thread counts, resulting in smaller gains for some benchmarks. Since the cores are in-order, Gains are higher on the PentiumD4SMT platform than the NiagaraII-8, since the OoO

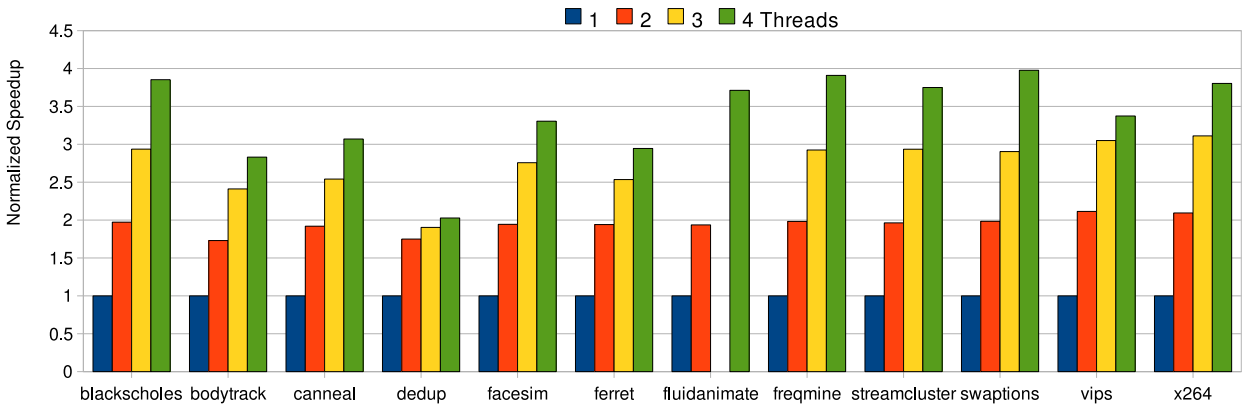

Figure 17. Phenom4 Performance Scaling 1-4 Threads

platform is already keeping execution units busy from other independent instructions, so there are fewer resources for executing instructions from another thread. The NiagaraII-8 uses round-robin scheduling of the instruction issue queue to achieve fine grain multi-threading, but not true SMT since instructions across different threads are not executed simultaneously. Future SMT designs can provide greater benefits if they include more resources per thread, so when the number of threads increase, there are adequate execution units and data queues available.

4.5 Thread Scaling Performance

It would be difficult to gain insight from comparing raw execution time across platforms, since the platforms operate at different frequencies, cache sizes, micro-architectures and even ISAs in some cases. However, by normalizing each platform to their single-thread performance we can observe scaling performance across different platforms. Figures 15, 16, 17, and 18 graph performance normalized to their single thread performance. Benchmarks `blackscholes`, `fluidanimate`, `freqmine`, `swaptions`, `vips` and `x264` scale well across all platforms. This is notable since PentiumD4SMT and Xeon8 require communicating over the FSB for all core to core communication and do not possess on-chip memory controllers. Depending on application, it might be cost-effective to use SMP designs, since very large CMP designs lead to smaller chip yields. `vips` does not scale perfectly at the highest thread count for Phenom4, which is attributed to the increasing bandwidth demands at higher thread counts. `streamcluster` scales better on large CMP designs such as Phenom4 and NiagaraII-8

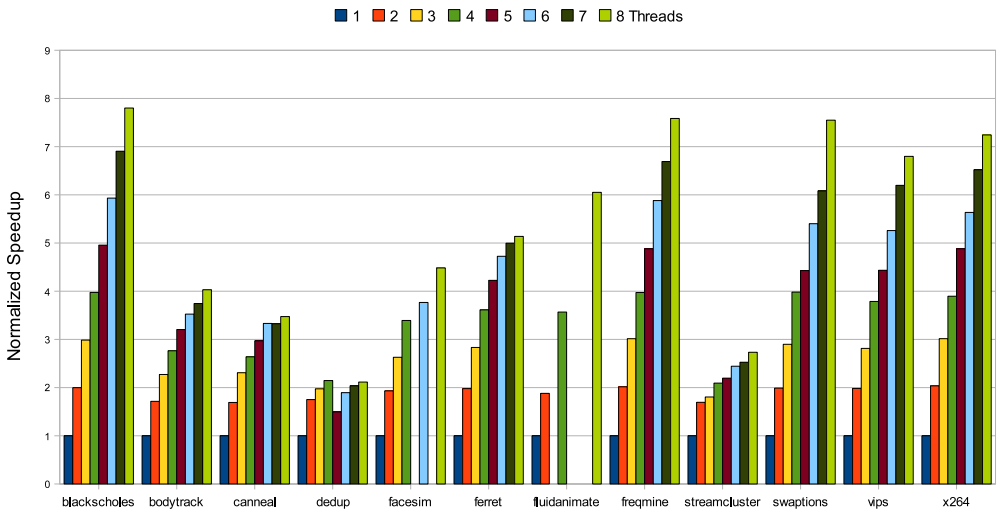

Figure 18. Xeon8 Performance Scaling 1-8 Threads

than on PentiumD4SMT or Xeon8. `streamcluster` has the highest number of FPU stalls as well as core to bus data transfers, and the second most L2 snoops. `streamcluster` is also sensitive to memory latency, as illustrated earlier. The SMT improvements on Niagara2 are most likely due to the swapping of threads when one thread is waiting on communication. This saturates at higher thread counts when all threads are stalled waiting for the single FPU on the core. `facesim` and `ferret` scale badly at high thread counts, which is surprising since they have little data exchange and are not communication bound.

`bodytrack`, `canneal`, and `dedup` scale badly on all tested platforms. `bodytrack` and `canneal` have a large variation in retired instructions at higher thread counts, especially `canneal` with a 70% difference. This bottlenecks performance as the single thread portion or work imbalance accounts for a larger portion of total execution time. `canneal` is also very sensitive to memory latency, which is exacerbated by the increase in threads. `canneal` also has the largest percentage of off-chip memory traffic, L2 cache misses and L2 tag snoops; making it more susceptible to being IO bound at higher thread counts. `bodytrack` also shows the least improvement with SMT, since increasing threads fails to reduce execution time of serial portions. `dedup`'s performance is interesting, since it does not exhibit signs of being limited by our previous architectural metrics like `bodytrack` and `canneal`. `dedup` could be slowed from HDD I/O, since smaller input sets (simmedium) exhibit linear scaling.

While current hardware plays a role in limiting the scalability of some applications, half of the PARSEC suite is

sufficiently computation bound to scale well regardless of platform. Unless communication latencies are improved, `streamcluster` and `facesim` will be saturated at higher thread counts. Currently, `bodytrack`, `canneal`, `dedup`, and `facesim` fail to scale well, although for `dedup` this is due to the benchmark IO rather than the underlying hardware. Based on our evaluation, off-chip memory traffic will not bottleneck systems as much as cross-thread communication will, thereby requiring better methods of sharing data across cores. Current native four and eight core designs are insufficient in this regard.

5 Conclusions

We perform a comprehensive characterization study of the PARSEC suite on a variety of hardware. We find most benchmarks are computation bound, and half of the suite scales linearly across all tested platforms.

While the memory requirements are significant for some PARSEC programs, the massive area required for larger caches for improving miss rates is impractical and can be better spent on hardware prefetchers or increasing system resources. At the thread counts studied, bandwidth and memory speeds were not a performance bottleneck, at least when compared to instruction level parallelism, micro-architectural resources and core-to-core communication. The expensive system cost for faster memory or increased bandwidth can be better spent on SMP systems, increased on-chip processors, or incorporating SMT support. However, even with optimum hardware resources, increasing thread counts achieves sub-linear scaling for some benchmarks, potentially due to the increasing costs of cache to cache communication. Even the native quad and eight core processors in our study failed to achieve linear performance improvements on half the benchmarks within the suite. We did not find a large victim L3 cache to be advantageous, and the area requirements are better suited for wider core to core busses, to reduce data transfer time.

The PARSEC benchmark suite represents a diverse workload for many different industries, and is not constrained solely to the high performance scientific community. We hope our results have scaled down the massive architectural design space; and provided a quantifiable basis for driving work into fruitful areas of research, when designing efficient CMPs for this heterogeneous domain.

References

[1] S. R. Alam, R. F. Barrett, J. A. Kuehn, P. C. Roth, and J. S. Vetter. Characterization of scientific workloads on systems with multi-core processors. In *Proc. IEEE International Symposium on Workload Characterization*, pages 225–236, Oct. 2006.

[2] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Report NAS-95-020, NASA Ames Research Center, Dec. 1995.

[3] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Proc. IEEE International Symposium on Workload Characterization*, Sept. 2008.

[4] C. Bienia, S. K. J. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. Technical Report Technical Report TR-811-08, Princeton, Department of Computer Science, Jan. 2008.

[5] D. Deleganes, J. Douglas, B. Kommandur, and M. Patyra. Designing a 3 GHz, 130 nm, Intel R Pentium R 4 processor. *Symposium on VLSI Circuits, Digest of Technical Papers*, pages 130–133, 2002.

[6] S. Eranian. Perfmon2: a flexible performance monitoring interface for Linux. In *Proc. 2006 Ottawa Linux Symposium*, pages 269–288, July 2006.

[7] G. Gerosa, S. Curtis, M. D'Addeo, B. Jiang, B. Kuttanna, F. Merchant, B. Patel, M. Taufique, and H. Samarchi. A sub-1W to 2W low-power IA processor for mobile internet devices and ultra-mobile pcs in 45nm hi-κ metal gate CMOS. *International Solid-State Circuits Conference, Digest of Technical Papers*, pages 256–257,611, Feb. 2008.

[8] R. R. 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. In *Proc. IEEE International Symposium on Workload Characterization*, pages 191–200, Oct. 2006.

[9] S. McKee. *Maximizing Memory Bandwidth for Streamed Computations*. PhD thesis, School of Engineering and Applied Science, Univ. of Virginia, May 1995.

[10] A. Moshovos, G. Memik, A. Choudhary, and B. Falsafi. JETTY: Filtering snoops for reduced energy consumption in SMP servers. In *Proc. 7th IEEE Symposium on High Performance Computer Architecture*, pages 85–96, Jan. 2001.

[11] U. Nawathe, M. Hassan, L. Warriner, K. Yen, B. Upputuri, D. Greenhill, A. Kumar, and H. Park. An 8-core 64-thread 64b power-efficient SPARC SoC. *International Solid-State Circuits Conference, Digest of Technical Papers*, pages 108–109,590, Feb. 2007.

[12] A. Settle, J. Kihm, A. Janiszewski, and D. Connors. Architectural support for enhanced SMT job scheduling. In *Proc.IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 63–73, Sept. 2004.

[13] H. Shikano, J. Shirako, Y. Wada, K. Kimura, and H. Kasahara. Power-aware compiler controllable chip multiprocessor. pages 427–427, Sept. 2007.

[14] Standard Performance Evaluation Corporation. SPEC CPU benchmark suite. `http://www.specbench.org/osg/cpu2006/`, 2006.

[15] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. 22nd IEEE/ACM International Symposium on Computer Architecture*, pages 24–36, June 1995.

[16] M. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, pages 23–34, Apr. 2007.