Our Many-core Benchmarks Do Not Use That Many Cores

Paul D. Bryan Jesse G. Beu Thomas M. Conte Paolo Faraboschi Daniel Ortega
Georgia Institute of Technology
{paul.bryan, jgbeu3, conte}@gatech.edu
HP Labs, Exascale Computing Lab
{paolo.faraboschi, daniel.ortega}@hp.com

Abstract

Increased processor performance can no longer be achieved through reduced feature size due to power density issues. As a result, high performance designs increasingly rely upon multiple processors in order to extract parallelism. Over the next few generations of designs, the core count of such machines is expected to increase, ultimately reaching thousands of processors. As the performance of multi-threaded programs has become increasingly important, the contemporary benchmarks of choice are PARSEC and Splash-2. Both of these suites offer great scalability from the program's perspective, where approximate ideal speedup can be obtained up to a certain processor threshold.

PARSEC and Splash-2 are attractive to many in the community because they offer programs that are highly scalable in the evaluation of large CMP designs. Both benchmark suites offer the *inherent* parallelization characteristics present within their programs as a metric for scalability. However, the assessment of program characteristics including system-level effects may skew overall program scalability results. Because speedup computed through Amdahl's law is extremely sensitive to the serial fraction of any program on the upper end, a minute change in the ratio of serial to parallel work may have dramatic effects upon the maximum obtainable speedup. The operating system, shared libraries, and thread synchronization incur serializing costs upon the overall program execution, which add to the serial code sections, and limit the scalability of these workloads to even a small number of cores. In this study, we show how these effects tragically limit multi-threaded performance, which must be overcome in order for new systems with a large number of cores to be viable.

1. Introduction

Power consumption and heat dissipation have become limiting factors to increased performance of single-processor designs. These constraints have limited the viability of feature size reduction as a technique to produce faster processors. In order to mitigate these effects, current high performance designs have shifted to include multiple processors on chip to compensate for decreased processor frequencies (and increased cycle times). The number of cores is expected to climb with each new design cycle, with the ultimate goal being thousands of processors.

In the past, SPEC [1] benchmarks were sufficient in the evaluation of single processor designs and multi-processor designs including a modest number of cores. As the number of processors has increased, it has become exceedingly difficult to evaluate chip multiprocessors in the context of throughput computing (where separate workloads are executed on individual processors). Typically, this would involve the arbitrary selection of workloads equal to the number of processors, where the total number of "CMP workloads" would be equal to ${}_{B}C_{P}$, which is the combination of B benchmarks taken P at a time (where B is the number of workloads contained in the suite and P is the number of processors in the design). Many problems stem from this type of evaluation and are described in [2].

As the number of processors have increased, PARSEC [3] and Splash-2 [4] have become the benchmark suites of choice. These multi-threaded workloads not only test the characteristics associated with synchronization and coherence, but also provide a standard substrate for individuals to compare their results (as well as mitigating the problems previously mentioned with throughput oriented workloads). Each of these benchmark suites is notably attractive because they advertise highly scalable workloads up to a specified processor threshold. PARSEC v1.0 reports indicate that their workloads approximately scale with the ideal up until 16 processors [5], and Splash-2 reports indicate scalability up until 64 processors [4]. Each of these reports determines the scalability of the program in isolation, which include program characteristics internally. Any external effects associated with the operating system and shared system libraries (which implement synchronization) are excluded from the analysis.

Multi-threaded programs may be broken down into two separate pieces, corresponding to their parallel and serial regions. According to Amdahl's law parallel program speedup is ultimately limited by the serial code sections of an application. Represented by the fraction of the enhanced ($F_{enhanced}$), inherent parallelism indicates the percentage of the program that is fully parallelizable. The remaining serial portion of the program may be calculated by $1\text{-}F_{enhanced}$. Once

the fraction of the enhanced is known, the maximum achievable speedup obtainable by parallelization may be calculated. Figure 1 shows the maximum obtainable speedup for fraction of the enhanced ($F_{enhanced}$) values between 0% and 99%. Assuming *perfect communication* and *no parallelization overhead*, a program must have at least 93.75% fully parallel code in order to obtain 16x speedup, and at least 99% for 100x speedup. Thus, for a real multi-processor system with 16 processors to obtain ideal speedup and utilize all processors, the target program must contain at least 93.75% parallel code. The underlying system cannot diminish the observed parallelization below that percentage (due to synchronization, additional coherence overhead, additional interconnect communication delay, shared libraries, OS scheduler events, load balancing, etc.). As the number of processors increase, multi-threaded program system performance will increasingly rely upon efficient thread execution and communication.

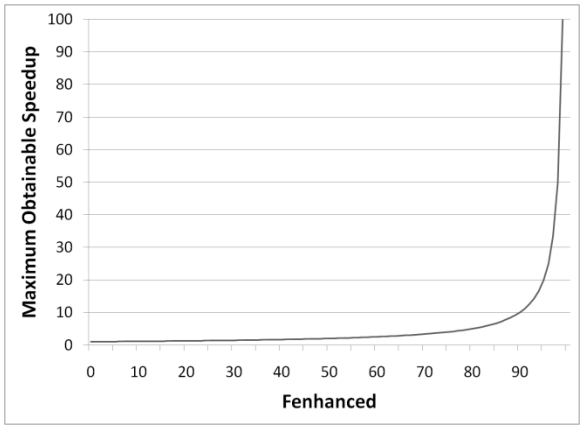

Figure 1: Maximum Speedup (0-99%)

In this study, we evaluate the observed speedups of various PARSEC and Splash-2 workloads. In order to understand the limitations of thread synchronization upon system performance, we characterize thread behaviors for an abstract machine model that incorporates operating system effects. Section 2 briefly describes basic thread synchronization mechanics studied in this work. The experimental framework is described in Section 3. Section 4 describes the characterization of synchronization events within the tested workloads and discusses its impact upon scalability. Section 5 discusses the measured speedups of workloads at varying workload/processor configurations and describes inherent obstacles to large multi-processor systems. Section 6 discusses related work followed by conclusions in Section 7.

2. Thread Synchronization

In multi-core systems, multi-threaded programs result in many concurrently executing tasks, or threads. In the POSIX implementation, each task resides within the context of a single process, allowing certain process resources to be shared, but each task to have separate program counters, registers, and local stack. Multi-threaded programs may spawn more threads than processors in order to avoid blocking while threads are waiting for a resource. However, too many active threads may hurt system performance due to OS context switching.

Thread interaction is governed by the use of synchronization primitives. The correct use of synchronization can prevent race conditions and other unexpected program behavior. Synchronization primitives may be implemented in the kernel or within user level libraries, and usually incorporate atomic operations to perform bus locking and cache coherency management. Atomic operations are a set of operations that are combined so the system perceives it as a single operation that succeeds or fails. Upon failure the effects of all operations are undone so that system state remains invariant. Examples of atomic operations include test-and-set, fetch-and-add, and compare-and-swap.

After a thread has been spawned it will run independent from the parent program unless dictated by synchronization. Shared resources that could be dynamically accessed simultaneously by multiple threads are often placed in critical sections to guarantee mutual exclusivity, typically guarded by locks or semaphores. If no threads are using a particular shared resource, then a thread may enter the critical section and is in a running state. Otherwise, the thread transitions to a blocked state while waiting for critical section access. Previous techniques have been proposed to mitigate the serializing effects due to locks [6-8].

System execution of multiple threads introduces potential non-deterministic effects due to the complex interactions between the scheduler, memory hierarchy and interconnect. Threads competing for a lock request resources from the memory hierarchy, and interconnect if the request misses. The first thread to obtain a cache line in exclusive access will win the lock and continue execution. Waiting threads will then obtain the lock based upon physical machine characteristics. Threads may also be preempted by other processes and/or threads which may change the observed synchronization events and program ordering. Non-deterministic effects may cause a program to behave differently over multiple runs, making debugging difficult.

The concurrent execution of threads may yield significant speedup compared to its single-threaded cousin if: 1) sufficient parallel code regions exist in the program, and 2) they can be executed efficiently with minimal synchronization bottlenecks and other system overheads. In this work, it is shown that

synchronization is an important factor that must be considered when evaluating workload scalability.

3. Simulation Environment

In this study, the linux GNU C library (glibc v2.8) was instrumented to profile synchronization events. Each type of synchronization provided by the Native POSIX Threading Library (pthreads) was instrumented to provide detailed information regarding thread behaviors. The pthreads library API provides for the following types of synchronization: mutexes, read-write locks, barriers, condition variables, thread joining, and semaphores. Read-write locks are multiple readers, single writer locks. Multiple threads can acquire a lock for reading purposes, but exclusive (write) accesses are serialized similar to mutex-based critical sections.

Using the instrumented libc, a number of workloads were simulated using HP Labs' COTSon simulator [9]. COTSon is a system-level simulator that models execution holistically, including: peripheral devices, disk I/O, network devices, and the operating system. Inclusion of the operating system allows for system calls and system interaction to be measured. Internally, COTSon uses the AMD's SimNow simulator [10] to provide fast native emulation of instructions, which are fed as a trace to COTSon timers for detailed measurement. For this work, multi-threaded workloads were simulated on top a 64-bit Debian operating system (kernel 2.6.26) using the instrumented libc.

In order to reduce context switching effects from other running processes, a stripped down Debian operating system was utilized. All non-necessary daemons were killed, and the X server was removed. To ensure that comparisons would be consistent across processor configurations, each workload was executed from the same operating system checkpoint image.

In order to classify synchronization events and their effects on thread behaviors, workloads from the Splash-2 [4] and PARSEC v1.0 [3] benchmark suites were simulated. COTSon provided timestamps for pthread events in order to determine detailed timing information for each type of synchronization event. For locks and mutexes, times were recorded for resource request, acquire, and release. Additionally, wait and release times were also recorded for thread joins, barriers, and conditional variables. The thread id and internal pthread data structure addresses were recorded to isolate individual thread behaviors and interaction based upon specific synchronization instances. Event timestamps were used to calculate the time that an individual thread spent within library synchronization calls. Individual thread wait times were determined as the cumulative difference between request/acquire and acquire/release events (in the case of locks) and wait/release events for joins, barriers, and condition variables. Although code instrumentation may affect the timing behavior of any program, the instrumentation of glibc was very lightweight (an event id was added to a buffer) and should not significantly alter the actual program behavior. Since our simulation environment was based on a 64-bit linux OS, we verified measurements on real systems containing eight processors for both the AMD Opteron and Intel Xeon architectures. Collected measurements for both architectures yielded speedups consistent with those in our simulated environment. Larger input sets were also executed on real systems and exhibited performance results similar to the smaller input sets.

Data were collected using a COTSon timer that functionally executed instructions in a single time unit. The use of a functional simulator has the same effect as if a cycle-accurate model were used with a perfect cache, branch predictor, TLB, pipeline (1-issue, in-order), interconnect, and coherence. The purpose of using this configuration was to approximate a lower bound of synchronization overheads. Assuming perfect communication across the cache hierarchy and interconnect, program performance is bounded by the serial code regions, OS interaction, and synchronization overheads. Synchronization overheads are based upon the ordering imposed by the benchmark algorithm as well as costs incurred within the pthreads library. The use of functional models has been incorporated in other studies to assess inherent program parallelization [3, 4]. Previous work has shown that Splash-2 and PARSEC have the inherent parallelization necessary to scale. However, when including the operating system and threading library effects within these measurements, application speedup may differ significantly from the ideal case.

4. Synchronization Characterization

PARSEC and Splash-2 workloads discussed in this section were simulated for 1, 2, 4, 8, 16, and 32 processors. At each processor configuration, the number of threads equaled the number of processors. From these data, average synchronization wait times were calculated (excluding one thread configurations because a serial program should not wait due to synchronization). All experimental workloads implement parallelization using either pthreads or OpenMP. Because the linux version of gcc internally uses POSIX threads by default to implement OpenMP pragmas, the instrumentation of pthreads was sufficient to capture the behavior for both types of workloads. In this study, PARSEC v1.0 benchmarks were simulated using the simlarge input set. Splash-2 workloads were also simulated using the default input size. Vips was excluded because it segfaulted for the simlarge input. FFT was also excluded from Splash-2 because its execution time was too small to be reliably measured by the time command.

Figure 2 shows the average time threads spent waiting as a percentage of program execution for the studied PARSEC and Splash-2 workloads. For each of these workloads, wait times have been decomposed into their synchronization constituents. These figures show the average percentage time that threads spent waiting for condition variables, barriers, and mutexes (including read-write locks) for all tested workload/processor configurations.

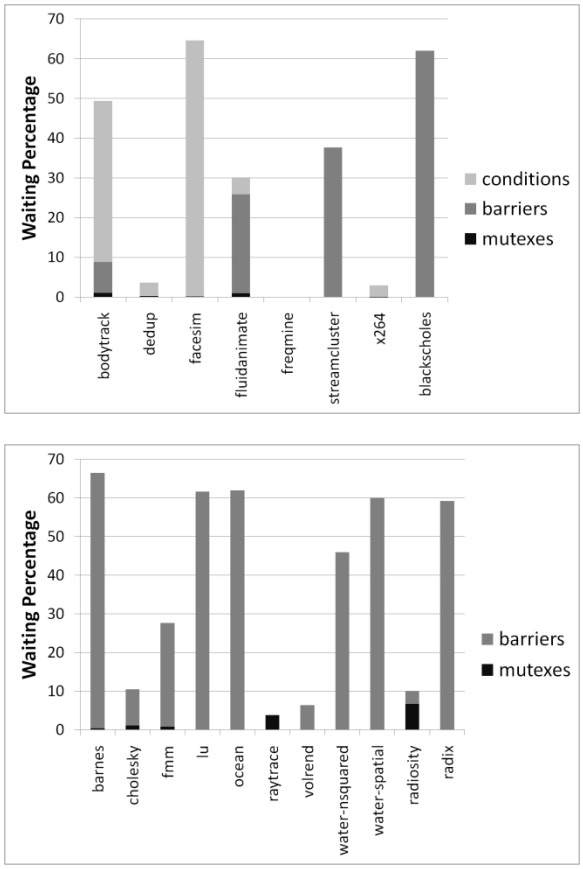

Figure 2: Thread Waiting Decomposition for PARSEC and Splash-2

For PARSEC, certain workloads spent considerable time waiting on synchronization events. Bodytrack waited for 49%, blackscholes for 62%, facesim for 65%, fluidanimate for 29%, and streamcluster for 38%. Others such as dedup, freqmine, and x264 all wait for less than 4%. Interestingly, condition variables and barriers dominate mutexes as the cause of waiting threads. Condition variables are barriers that can be selectively applied to individual threads. Once a thread reaches a condition variable, it must wait until that condition becomes true. Upon receiving the wakeup signal, thread execution will continue. For all workloads, mutexes account for less than 0.34% of thread execution. Barriers and condition variables account for 8.7% and 14.3%, respectively. On average, 17.3% of PARSEC workload execution was consumed on synchronization.

For Splash-2 workloads, the average synchronization overheads for mutexes and barriers are 1.3% and 34.2%, respectively. No condition variable activity was measured for these workloads. Similar to PARSEC, mutexes had minimal impact upon overall thread wait times. On average, Splash-2 wait times consumed 35.4% of program execution. In part, this is due to the short execution times of Splash-2 workloads. On average, all Splash-2 permutations executed in 0.68 seconds within the simulated operating system. The short execution times of Splash-2 workloads may have issues associated with constant timeslice interruption. However, these workloads are over a decade old and may be outdated for contemporary system evaluation. But, even when these workloads were relatively new, scalability issues were observed [11] for NUMA architectures.

Wait times for mutex synchronization was extremely low, implying threads can regularly acquire locks uncontested. Similar behavior for the low contention rate of mutexes in cycle-accurate simulation environments has been observed in [6, 7]. Instrumented barriers are extremely costly even for our simulated abstract machine because all system threads must wait for the slowest thread to reach a specified execution point. If threads have common algorithmic tasks and similar performance, then the slack time between the highest and lowest performing thread should be minimal. Threads with dissimilar tasks or whose performance varies greatly will incur greater barrier costs.

The scheduler may also impact associated barrier overhead because it may preempt thread execution in lieu of another system process. Assuming homogenous execution among the remaining threads, the evicted process will then become the slowest thread, and program execution cannot continue until it is both rescheduled and reaches the barrier. Here, the scheduler overheads provide a non-intuitive trade-off. If the OS time slice is too short, then threads could be preempted frequently by other system processes. Increasing the time slice interval could reduce preemption, but could also increase the penalties of preemption when it occurs. It is currently unclear which scheme would most benefit the performance of barrier execution and is left for future research.

A detailed decomposition of thread behaviors at the varying thread counts is shown in Figure 3 and Figure 4 for interesting workloads with the highest average wait times. For these workloads, the cost of synchronization increases with the number of threads. Bodytrack and facesim both result in increased wait times for conditional variables. Fluidanimate, streamcluster, barnes, lu, ocean, and water-spatial all result in greater barrier costs at higher thread counts. Wait times for bodytrack are in contrast with the other workloads (which exhibited decreased barrier wait

times as the number of threads increased) because one thread spent the majority of its time waiting for all other worker threads to complete. The addition of threads in this workload caused the overall wait times to decrease because it was averaged over more running threads.

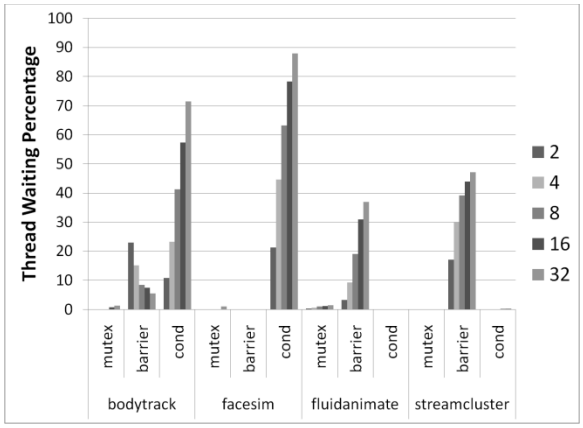

Figure 3: Thread Waiting vs. Thread Count for a subset of PARSEC

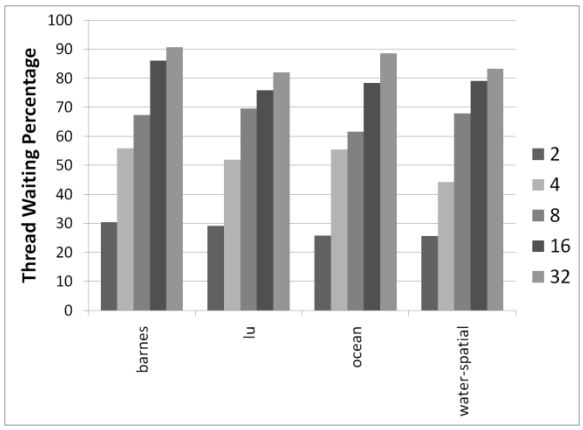

Figure 4: Barrier Overhead vs. Thread Count for a subset of Splash-2

Measured wait times were dependent upon the time spent within synchronization library functions, other miscellaneous system calls, and thread execution. As the numbers of threads are increased, highly parallel workloads that scale well have significantly less execution time. If the synchronization costs are fixed, then this would result in linear wait percentage increases. Insufficient parallel code or poor synchronization performance may cause workloads to scale poorly. In either case, the addition of threads that synchronize over the same shared structures (via condition variables or barriers) increases the overhead of the pthreads library. Such overheads, however minimal, may have dramatic impacts upon the observable scalability that can be extracted from a multi-threaded program at high thread counts, and are discussed in Section 6. In this study, many programs contain inflection points where the addition of threads will no longer help performance (or worse, hurt performance). A more detailed discussion of simulation times is discussed in Section 5.

The synchronization penalties of mutexes among the tested workloads were numerous and light. Although the specific behavior is workload dependent, the wait times associated with mutexes generally were very small and contributed little to overall thread waiting. Additionally, dynamic instances of barriers and condition variables were much less frequent than mutexes but had dramatically higher overhead.

5. Observed Speedup

Simulated speedups of each program were compared against the ideal case and were based upon the "real" execution from the perspective of the simulated OS. Execution times for each of the different processor counts were compared against the single-threaded case to derive the parallelizable fraction of execution. This resulted in five $F_{enhanced}$ ratios for each workload: one calculated by the observed speedup between 1 thread and 2 threads, between 1 thread and 4 threads, between 1 thread and 8 threads, etc. For each of the five computed fractions, the maximum was selected because it revealed the best parallelism that was observed at the system level. Measured fractions lower than the maximum indicate overheads that inhibited concurrency. While the program in isolation may inherently contain a fraction higher than the observed, it is important to include the serializing system-level effects in the measurement. Derived $F_{enhanced}$ values for the PARSEC and Splash-2 benchmark suites are shown in Table 1 and Table 2 and indicate the maximum projected speedup for the largest $F_{enhanced}$ measurement.

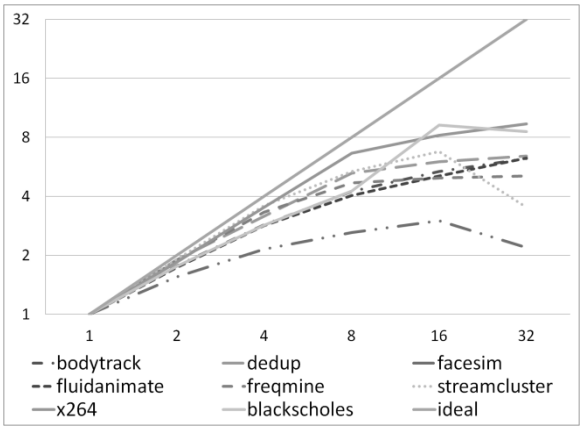

Figure 5: PARSEC Threads vs. Speedup

Figure 5 and Figure 6 show measured speedups of simulated workloads as the number of threads and processors are increased logarithmically. Surprisingly, even if a perfect processor model is incorporated, no workload is able to scale past 10x once system-level effects are incorporated. Associated speedup for the

two benchmark suites were dramatically impacted by the wait times of synchronization. Splash-2 suffered higher synchronization event wait times than PARSEC, which accounts for the lower obtained speedup values for these workloads.

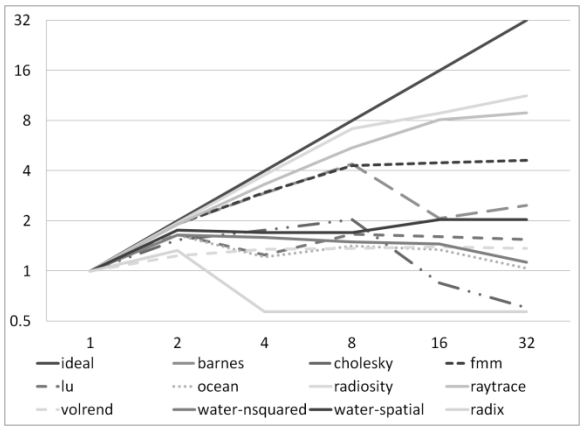

Figure 6: Splash-2 Speedup vs. Processor Count

Benchmark	$F_{enhanced}$	Projected Speedup
bodytrack	0.8740	6.52x
dedup	0.9481	12.27x
facesim	0.7127	3.23x
fluidanimate	0.8676	6.27x
freqmine	0.9331	10.41x
streamcluster	0.9615	14.59x
x264	0.9709	16.83x
blackscholes	0.9512	12.73x

Table 1: PARSEC Parallelizable Fraction

Benchmark	$F_{enhanced}$	Projected Speedup
barnes	0.9533	13.08x
cholesky	0.6949	3.06x
fmm	0.9600	14.28x
lu	0.8000	4.44x
ocean	0.7843	4.16x
radiosity	0.9835	21.20x
raytrace	0.9520	12.87x
volrend	0.3864	1.59x
water-nsquared	0.7843	4.16x
water-spatial	0.8627	6.08x
radix	0.5000	1.85x

Table 2: Splash-2 Parallelizable Fraction

At 32 processors, x264 had the highest speedup of 9.38x and facesim had the smallest speedup of 2.13x. On average, workloads with 2 processors had a speedup of 1.79. Increasing the processor counts to 4, 8, 16, and 32 had average speedups of 3.06x, 4.69x, 5.62x, and 5.58x, respectively. The performance of all workloads increased up until 16 processors. After this point, the simulation times for certain workloads actually increased. When comparing 16 and 32 processors, the program execution time of facesim increased from 13.8s to 14.2s, and streamcluster increased from 3.26s to 6.29s. Splash-2 workloads exhibit similar behavior and are included in Figure 6. Radiosity, raytrace, fmm, and barnes all scaled well relative to the ideal case up until two processors. Beyond two processors, only radiosity and raytrace scaled up to 8 before saturating. Cholesky suffered slowdowns at 16 and 32 processors, and radix suffered slowdowns beyond 2. An average speedup factor for both suites was less than 6x at 32 processors.

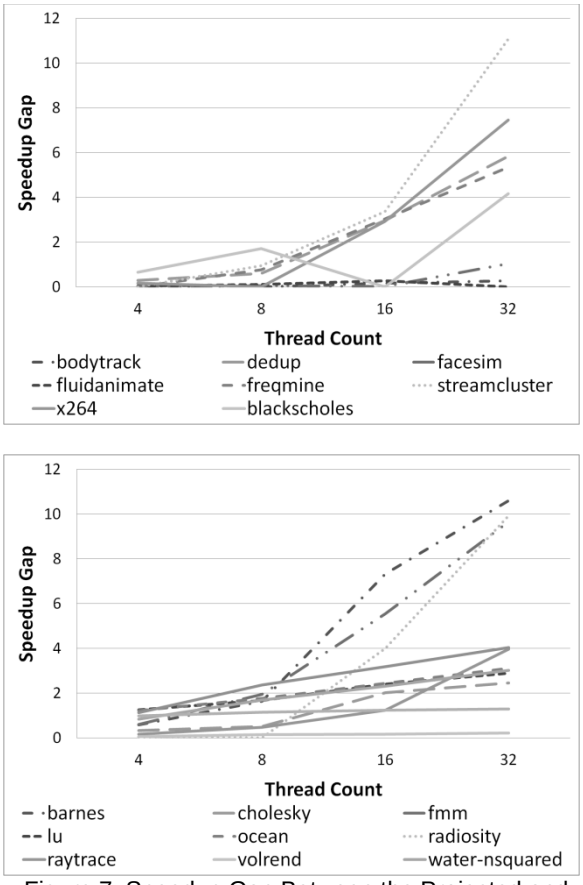

Figure 7: Speedup Gap Between the Projected and Measured

Figure 7 shows the gap between the measured speedup of benchmark performance relative to the predicted speedup from calculated $F_{enhanced}$ values. Parallelization fractions represent the maximum amount of observed parallelism from the simulated workloads, including all system overheads. As the number of processors and threads increase, the difference between the projected speedup and the simulated speedup is directly attributable to additional system overheads. The parallelization projected by the PARSEC and Splash-2 workloads did not yield sufficient scalability for even a small number of cores. Up to 8 processors PARSEC had adequate scaling, but saturated at 16. Most workloads obtained little gain

beyond 16 processors, and some exhibited degradation. Reported speedups for PARSEC workloads indicate that inherent program parallelism will scale up to 16 processors. When OS and synchronization overheads are taken into consideration, ideal scaling was not seen. Furthermore, as designs increasingly incorporate 32 processors, scalability for these workloads will become a much greater issue. For Splash-2, only two workloads were resistant to the OS and synchronization effects. The remaining workloads began to show saturation or performance degradation after two processors.

6. Putting It All Together

In order to more adequately understand the system bottlenecks associated with workload scalability, additional experiments were conducted within the simulated OS for each workload/processor combination using OProfile. On average, 33% of program execution of PARSEC workloads and 52% of Splash-2 workloads were spent within the linux kernel. This does not imply inefficiencies within the OS, but rather is an artifact of the interaction between waiting threads and the scheduler. Due to potentially high wait times associated with thread execution (shown in Section 4), threads often cannot make forward progress due to synchronization stalls. (Also, newly spawned threads that have yet to be given any useful work are started in the idle state.) If a thread is stalled, and thus not performing any useful work, it will eventually be removed from the run queue by the scheduler, and sent to a wait queue.

Decomposition of the time spent within the linux kernel shows that approximately 97% of the time is spent within the *default_idle* kernel function and 3% is spent for all other OS services (e.g. memory mapping, scheduling, I/O, filesystem bookkeeping, etc.). Outliers from this type of behavior are dedup and streamcluster from PARSEC, and water-nsquared from Splash-2. Dedup and water-nsquared spend 12% and 9% of their execution within the OS on non-idle services and can be attributed to poor paging behavior. Water-nsquared also exhibits behavior similar to dedup. Streamcluster spends approximately 31.2% of its execution with the OS on non-idle services due to load imbalancing issues. The behaviors of barriers within streamcluster indicate that approximately half of the threads never wait for barriers, while the other half spend considerable time waiting.

In general, time spent within shared libraries was small, however significant times were measured for the standard C++/C, math, and pthread shared libraries and varied between less than 1% and 5% (excluding thread wait times that were measured within the default_idle kernel function). When excluding the default_idle kernel function from the OS measurements, the OS component of workload execution varied between less than 1% and 31%.

Previous work [12] has demonstrated the impact that system calls and kernel code may have upon system performance, and advocate its inclusion within single-threaded simulation. We are advocating that synchronization, system calls and OS behavior *must* be considered when evaluating possible speedups that could be obtained from multi-threaded workloads. Future systems containing hundreds or even thousands of cores will increasingly rely upon massively parallel code in order to obtain speedup. In these systems, effective processor utilization is dependent upon input workloads that converge upon 100% parallelization. For example, if a system can execute 99.99% of a program in parallel, then the maximum attainable speedup is 10000x. Decreasing the enhanced fraction by 0.09 to 99.9% reduces the maximum obtainable speedup by 9000x!

The parallelization requirements of contemporary workload/system pairs are modest in comparison to that of the hypothetical 1000 core machine. In our experiments, measured $F_{enhanced}$ values indicate that at least three of the benchmark inputs contained the inherent parallelism necessary to scale to 16 processors, and two benchmark inputs contained the parallelism to scale above 20x. Yet, no input for any benchmark was able to obtain speedup past 11x when considering the additional system overheads included in our model.

7. Related Work

When discussing parallel benchmarking sets, various methods have been proposed to discuss the inherent parallelization characteristics. Woo et al [4] measure speedup for Splash-2 workloads using perfect caches and communication. All instructions executed in their environment complete in one cycle. The authors note that non-deterministic behaviors of programs make it difficult to compare data when architectural parameters are varied because the execution path may change. Bienia et al [3] measure the inherent program concurrency based upon the number of instructions executed in the parallel and serial regions of code. Delays due to blocking and load imbalance are not studied because they focus on the fundamental program characteristics. Our characterization differs in that we consider the operating system, shared system libraries, and detailed thread synchronization in our analysis.

8. Conclusion

As more processors are added to next generation designs, it is important to identify the capabilities of application parallelization. If a new system has a large number of cores, then programs must be able to adequately leverage its resources in order to be

effective. For the studied workloads, parallel execution was insufficient to scale along with the ideal case. This is in contrast to other work which describes the identified parallelism found within the workloads in isolation. However, discrepancies between the two can be explained by synchronization, the OS and other shared system libraries that are measured within our infrastructure. Synchronization incurs significant overheads which must be measured to obtain realistic performance projections as the number of cores scale. The effect of pthread calls causes many threads to block, thereby increasing serial sections of the multi-threaded program and decreasing $F_{enhanced}$. Furthermore, additional OS overheads increase serial code sections and limit parallelization opportunities. As more processors are added to commodity systems, the OS and shared libraries will play an increasingly important role in the available parallelism that can be achieved in a multi-threaded workload.

9. References

[1] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1-17, 2006.

[2] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero, "FAME: FAirly MEasuring Multithreaded Architectures," in *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, 2007, pp. 305-316.

[3] C. Bienia, S. Kumar, and L. Kai, "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on Chip-Multiprocessors," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, 2008, pp. 47-56.

[4] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd annual international symposium on Computer architecture* S. Margherita Ligure, Italy: ACM, 1995.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* Toronto, Ontario, Canada: ACM, 2008.[6] R. Rajwar and J. R. Goodman, "Speculative lock elision: enabling highly concurrent multithreaded execution," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture* Austin, Texas: IEEE Computer Society, 2001.

[7] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems* Washington, DC, USA: ACM, 2009.

[8] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* San Jose, California: ACM, 2002.

[9] E. Argollo, A. Falc, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 52-61, 2009.

[10] R. Bedicheck, "SimNow: Fast platform simulation purely in software," *Hot Chips 16*, 2004.

[11] A. Chauhan, B. Sheraw, and C. Ding, "Scability and Data Placement on SGI Origin," *Technical Report TR98-305*, 1998.

[12] J. A. Redstone, S. J. Eggers, and H. M. Levy, "An analysis of operating system behavior on a simultaneous multithreaded architecture," *SIGPLAN Not.*, vol. 35, pp. 245-256, 2000.