A Portable Framework for High-Speed Parallel Producer/Consumers on Real CMP, SMT and SMP Architectures

Richard T. Saunders[1], Clinton L. Jeffery[2], and Derek T. Jones[1]

[1]Rincon Research Corporation
IRAD Division
Tucson, AZ 85711 USA
{rts, dtj}@rincon.com

[2]University of Idaho
Dept. of Computer Science
Moscow, ID 83843 USA
jeffery@cs.uidaho.edu

Abstract

This paper explores generating efficient, portable High-Speed Producer Consumer (HSPC) code on current shared memory architectures: Chip Multi-Processors (CMP), Simultaneous Multi-Threading processors (SMT) and Shared Memory Processors (SMP). To build an HSPC, we use a code generation approach in two stages.

Stage One generates data structures to eliminate memory interference. This is done by adjusting and timing cache/buffer/stack placements and lengths for an idealized producer/consumer. Perfect load-balancing is achievable for CMP and SMP, but not for SMT due to simultaneous-execution interference.

In Stage Two, the codebase is refined inside its target application: profiling events sent from Python to a consumer that computes profiling information. Stage two further tests the impact of altering event sizes, synchronization primitives, container libraries, and processor affinity. Stage two achieves near perfect balancing for CMP and SMP architectures, but SMT still performs poorly.

1. Introduction

The producer/consumer relationship is a canonical pattern among event-based communicating processes/threads[2]. This work addresses the class of event-based systems in which event processing speed is a dominating factor. The example application driving this work is a monitoring and visualization facility for Python[13], a profiling system that sends events when certain frequent activities occur[15]. Such systems require the fastest producers/consumers possible, but writing a high-speed producer/consumer is surprisingly difficult (see Related Work as well as [4]).

1-4244-0910-1/07/$20.00 ©2007 IEEE.

For monitoring systems such as profilers, event processing is overwhelming: potentially millions of events per second. This is a major problem for profiling systems: detailed event processing is time-intrusive enough to obscure the behavior of the program under observation. Some profiling systems resort to statistical sampling (such as gprof[5]), while others (such as Alamo[6]) adopt numerous techniques to reduce the cost of event production/consumption in a classic uniprocessor context.

A more aggressive solution in modern systems is to offload work to another processor. This solution has been available in the high-performance community for some time, but recently has become mainstream thanks to processor trends toward hyperthreading and multi-core processors[9, 14]. The event generator does minimal work to generate the event (copying the event to a buffer), then hands the event-processing work to another CPU. That CPU processes the event concurrently, leaving the event generator to make progress and continue “real work.” In this way, the profiled program’s perturbation is minimized.

We encountered this particular problem in the context of Python profiling[13], and were motivated to investigate the general purpose High-Speed Producer/Consumer (HSPC) techniques described in this paper. In an HSPC scenario, a producer (event generator) needs to generate events as fast as possible, and the consumer (event processor) needs to consume and process the events as fast as possible so coordination between them does not degrade the performance of the system.

This paper demonstrates how to build an HSPC tool for a particular platform, and also suggests methods that might be implemented in libraries (such as the POSIX threads library) or compilers and runtime systems.

2. Methodology

In order to generate HSPC code, we experimentally generated code in two stages: Stage One attempted to produce interference-free code using an ideal producer and consumer. Stage Two used a more realistic real-world application (profiling) to tune the codebase from Stage One.

In order to generate cross-platform code, the HSPC code generator ran on seven different UNIX/Linux platforms, representing three different types of shared memory multiprocessor machines: Chip Multi-Processors (CMP), Shared Memory Multi-Processors (SMP) and Simultaneous Multi-Threaded processors (SMT) (SMT is also known as hyperthreading).

- G5 CMP: A dual-core 2 GHz PowerPC G5
- Opt64 CMP: A dual-core 1.8 GHz Opteron 64
- Opt64 SMP: A dual processor 1.8 GHz Opteron 64
- Xeon SMP: A dual processor 3.2 GHz Xeon
- Alpha SMP: A four processor 500 MHz AlphaEV6.7
- Xeon SMT: A hyperthreaded 3.2 GHz Xeon
- P4 SMT: A hyperthreaded 3 GHz Pentium 4

3. Stage One Code Generation

Stage One of the process to generate HSPC code is to generate interference-free code. *Interference-free* means a producer and consumer pair that will run in parallel as fast as the *baseline* would run by itself. In other words, the presence of a running consumer will not affect the run-time of the producer. If we can achieve this, we can claim perfect balancing for an HSPC pair. We found that we could achieve perfect balancing for all CMP and almost all SMP architectures.

We encountered three broad types of interference: Simultaneous-Execution, Shared-Buffer, and Synchronization. *Simultaneous-Execution Interference* (SEI) occurs when an orthogonal thread running in parallel with the producer slows it down. The actual presence of the running thread, even though there is no communication between the two, causes interference. *Shared-Buffer Interference* (SBI) occurs when the producer and consumer start sharing the same buffers when run together. *Synchronization Interference* (SI) occurs when the producer and consumer start synchronizing so that events are delivered reliably and in-order.

To develop interference-free code, we experimentally evolved a codebase. Starting with a baseline, we evolved the code until it reached a perfectly balancing HSPC. At each step, we either added a feature (that typically caused interference and slowed down the code) or hand-tuned some parameters (striving to eliminate interference from a recently added feature). This process was followed across multiple platforms to ensure the work was portable.

The process to develop the codebase was as follows:

1. *Determine the Baseline:* Run the producer as fast possible, generating single-byte events into a 1K buffer, and time it (similarly, independently run and time the consumer). In this case, there is absolutely no synchronization or interference because there is no parallel thread generating interference.

2. *Determine and Eliminate Simultaneous-Execution Interference:* Run the baseline consumer in parallel with the baseline producer. At this point, the consumer does not share any buffers or synchronize in any way with the producer. Hand-tune the producer and consumer code at this step to eliminate as much of the SEI effect as possible.

3. *Determine and Eliminate Shared-Buffer Interference:* With SEI eliminated, we introduce a buffer that the producer and consumer share, but with no synchronization for accessing that buffer. We now run the producer and consumer together, tuning the producer and consumer to eliminate SBI.

4. *Determine and Eliminate Synchronization Interference:* With SEI and SBI eliminated, we introduce synchronization so the consumer reads events in order and reliably. We hand-tune the codebase to eliminate SI.

This four-step process allowed us to develop interference-free code for the HSPC. Figure 1 shows the roadmap of the process, with timings and descriptions at each step. Each column of Figure 1 represents one of the seven testbed architectures. Each row represents a particular phase of the evolving codebase. The following sections detail the four steps.

All code described in this paper is written in C++ and is available for download at http://www.amalgama.us/hspc.html. The producer and consumer each run in a separate POSIX kernel thread (called "system scope" by the POSIX specification).

Most instances of slowdown of the HSPC pair occurred because of some form of false sharing: *false sharing*[4, 9] occurs when two threads access distinct memory addresses that map to the same cache line. Managing the cache from concurrent memory accesses degrades performance. The false sharing took different forms. *Local Adjacency* is when two variables are spatially next to each other so as to share the same cache line. *Modulo Adjacency* is when variables are spatially distant from each other, but because their memory locations map to the same place in the cache (the cache may be direct-mapped), they share the same cache-line.

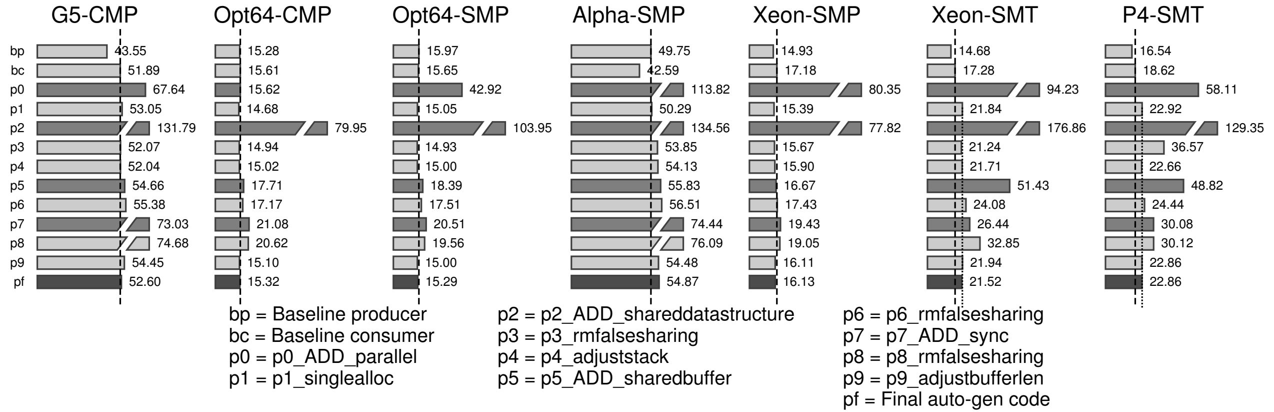

Figure 1. Roadmap of Stage One Code Generation across seven platforms. The horizontal bars represent the execution time (in seconds) of the particular codebase—the actual value is annotated on each bar. Dashed lines represent the baseline for that platform, and dotted lines represent the SMT baseline (see Section 3.2) for that platform. Darkened bars represent the addition of a feature.

3.1. Determine The Baseline

At this step, we ran the producer by itself (see the row labeled Baseline producer in Figure 1) and timed it. This is the ideal: it is the fastest a producer can run. There is no interference, as there are absolutely no threads/processes competing for resources. This time served as the baseline: HSPC aspires to run as fast as the baseline producer. Similarly, we ran the consumer by itself and timed it.

Interestingly, for most cases, the consumer ran slower than the producer (see the row labeled Baseline consumer in Figure 1). When this happened, the consumer was the bottleneck rather than the producer; thus we defined the consumer to be the baseline for those cases.

The producer/consumer code remained invariant through the Stage One process: only the HSPC buffers and data structures were changing.

3.2. Determine and Eliminate Simultaneous-Execution Interference

To start the process, the baseline producer and baseline consumer were run in parallel. This is captured in test case p0_add_parallel in Figure 1. Immediately, Simultaneous-Execution Interference appeared as the SMP and SMT cases were 2.7–5.4 times slower than the baseline (curiously, the CMP cases weren't as affected).

The first problem is buffer layout: the producer and consumer each allocate their own (non-shared) buffer separately via `malloc`. Unfortunately, this meant there was no control over where the buffers were laid-out in memory, causing potential modulo adjacency issues. To combat unpredictable buffer layouts in memory, we allocated the producer and consumer buffers in a single allocation next to each other; thus the cache-line interference of each buffer relative to the other could be controlled. The codebase p1_singlealloc contains the fix for this problem and brings us back to baseline times (except for SMT, see below).

The codebase p2_ADD_shareddatastructure is only an organizational change as the data structure for the producer and the consumer were merged: this merging introduced two instances of false sharing (we see the slowdown in p2 times). The first instance was that the read and write buffer pointers were locally adjacent causing massive interference. The second instance was that the write buffer pointer was locally adjacent to the front of one of the buffers (the entire data structure was allocated in a single allocation to avoid the buffer layout issues discussed above). Both instances of false sharing were fixed by adding some padding between the offending variables to force them into separate cache lines. The effects of adding the padding are shown in Figure 1 at p3_rmfalsesharing. On most platforms, we returned to baseline times (except SMT, see below).

The **P4-SMT** platform had an additional platform-dependent complication related to stack layout affecting its scalability. It has been noted in the literature[4, 9, 11] and is called the *64-kilobyte Aliasing Problem*: it is essentially modulo adjacency. To fix this problem, we introduced padding on the producer's stack so that the producer and consumer stack data didn't overlap in the 64-kilobyte area. This corresponds to p4_adjuststack. This step brought **P4-SMT** down back down to its baseline (except SMT, but see below).

SMT Scalability. Although we eliminated Simultaneous-Execution Interference for CMP and SMP architectures, we had difficulty doing so for SMT. This seems to be a fundamental limitation of the current hardware for HSPC. Our findings are consistent with other recent work demonstrating limitations[7, 11] of SMT parallelism.

The work in [11] suggests that memory bandwidth issues might be the problem. **Xeon-SMP** and **Xeon-SMT**, however, are the same machine (with different CPUs activated) with the same main-memory subsystem, and the **Xeon-SMP** load-balances perfectly well. Note that running the machine in **Xeon-SMT** mode splits the level 1 data cache between the two SMT processors and running the machine in **Xeon-SMP** mode gives each full CPU its own full level 1 data cache. A possible problem is that the level 1 data cache is too small on SMT platforms. Since filling those caches still relies on the same memory subsystem on both **Xeon-SMP** and **Xeon-SMT**, that problem seems less likely, especially since everything should fit in those caches at this stage (the tests use very small 1024 byte buffers until the very last step). Memory bandwidth does not seem to be the issue, as both **Xeon-SMP** and **Xeon-SMT** would suffer.

The problem is uncovered early when investigating the Simultaneous-Execution Interference. This suggests the complication with SMT is that the execution units inside the chip are in high demand by both the producer and consumer, so the execution units cannot be used completely in parallel. Attempting other mitigation techniques from Stage Two (processor affinity, different synchronization primitives, different data structures) at this step had *no effect* on the SMT scalability; there seems to be an intrinsic limit. There may be techniques to reduce this execution unit interference (looking at hardware registers on-chip, rewriting assembly code to avoid execution-unit sharing or other compiler techniques), but these are beyond the scope of this paper.

We cannot eliminate all Simultaneous-Execution Interference for the SMT case, but we do show that we *can* eliminate all shared-buffer and synchronization interference (in the steps below). Recognizing that we cannot eliminate all SEI, the codebase for p1_singlealloc becomes the baseline for SMT (note that it is about 1.25x the execution time of the original baseline).

3.3. Determine and Eliminate Shared-Buffer Interference

At this step, we introduced buffers that both the producer and consumer shared. There was no synchronization. The producer puts events to the buffer that the consumer gets (but not necessarily in order). Note that we also introduced triple buffering at this step because we know we need double/triple buffering[1] eventually (see next section). The three shared buffers were allocated with a single allocation to avoid the buffer layout problems from Section 3.2.

Consider p5_ADD_sharedbuffer: adding the shared buffers drops us about 1.2x–2.3x off the baseline. The problem is the false sharing occurred between the edges of the buffers: if the start of a buffer overlapped the cache line with the end of another buffer, false sharing could occur.

We fixed this problem by adding padding between the buffers so that the buffer edges started at different cache lines. Consider p6_rmfalsesharing: this fix made a tremendous difference on SMT platforms and a reasonable difference on most other platforms.

Strictly speaking, though, we did not eliminate all SBI at this step. We further refine the buffer sizes as part of the next step to bring us back to the baseline numbers.

3.4. Determine and Eliminate Synchronization Interference

This step introduced synchronization so that the consumer would read all events reliably and in order. For synchronization, we used portable POSIX condition variables[2].

It was impractical to use a single buffering scheme at this step; a producer and consumer would serialize behind each other as one waits for the other to finish. In order for the producer and consumer to proceed in parallel, we had to use double or triple buffering. Synchronization happens at a per-buffer level: a buffer is locked once, then multiple reads (writes) proceed until the buffer is emptied (filled), at which point the buffer is unlocked. In this way, synchronization occurred only when a complete buffer was used.

Consider p7_ADD_sync: adding the synchronization degraded performance, running about 30% slower than baseline in each case. This was partially caused by locally adjacent POSIX condition variables.

To eliminate that false sharing, we added padding to the HSPC data structure so that no condition variables were in the same cache line.[2] The results are in p8_rmfalsesharing.

The final fix was to adjust the size of the buffers. The size and number of buffers is important: if the buffers are too small, synchronization overhead becomes excessive as the producer and consumer synchronize frequently; if the buffers are too large, there are known instances (See **Xeon-SMP** at Stage Two) where we fall out of cache and degrade performance. Consider p9_adjustbufferlen: by hand-tuning the buffer sizes and number of buffers, we were able to achieve most baseline run times.

For the CMP machines, we achieved nearly perfect interference-free code. For the SMP machines the results

[1] Double, triple or n–buffering means round-robin through n buffers.

[2] This is especially important for Stage Two, when we need to make sure spinlocks stay local[10].

were almost as good. The machine **Opt64-SMP** balanced perfectly. The **Xeon-SMP** baseline consumer is actually slower than the baseline producer, so the consumer was the bottleneck. We did better than the baseline consumer, if not quite as good as the baseline producer. The **Alpha-SMP** case was slightly disappointing, but close.

Unfortunately, for the SMT machines, we were never able to eliminate all interference, specifically SEI. For **P4-SMT**, the work offloaded is $1 - \frac{22.86}{16.54+18.62} \simeq 35\%$. For **Xeon-SMT**, it is $1 - \frac{21.52}{14.68+17.28} \simeq 33\%$. Thus we were limited to offloading only about 33–35% of the work (compared to perfect balancing, offloading 50% of the work). The p1_singlealloc time became the SMT baseline as we eliminated all shared-buffer and synchronization interference.

3.5. Automating Code Generation

The Stage One process produced an interference-free HSPC codebase (or as close as could be reached on the hardware under test). All relevant parameters contributing to interference had been discovered and mitigated, but their values were tuned by hand. For portability, the ideal values for these parameters should be determined automatically. These parameters are cache line padding, stack layout offset, number of buffers, and buffer sizes. To determine the values, we take a SuperOptimizer[8] type approach where, without changing the codebase, many possible parameter values are tested and timed to determine which values produce the shortest run times.

Testing all combinations of all relevant parameters is computationally expensive, and in this case, unnecessary. The cache line padding value and stack layout offset value can be determined independently, and we know experimentally that ideal values exist. We determine a reasonable cache line padding value by running p3_rmfalsesharing codebase in isolation. With that, we have a value that we know contributes no interference, and we use it in determining the stack layout offset, by running p4_adjuststacklayout. See Figure 2. Note that we only cycle through powers of two in loops 1 and 2: the idea is that we are just looking for a value to cross the cache line or stack offset boundary. There may be slightly better values, but we are trying to limit code-generation time.

Finding the right buffer size values is more time consuming. With the ideal cache line padding and stack layout offset numbers known, we ran the p9_adjustbufferlen code baseline with all reasonable permutations of buffer size and number of buffers. The "reasonable" buffer sizes were discovered through experience when tuning the many different platforms (for example, small buffer sizes were useful on some platforms: it was hopeful that on CMP and SMT platforms, with the smaller level 1 caches, that data would fit inside the cache and avoid going off chip to perform cache coherence). This part of the process typically takes a few hours.

```
for cache_line_padding (1 2 4 8 16 32 64 128 256)
  time and run p3_rmfalsesharing    /* loop 1 */
for stack_layout_offset (32 64 128 256 512 1024)
  time and run p4_adjuststacklayout /* loop 2 */
for number_of_buffers in (2,3,4,5)
  for buffer_size in (256 512 768 1024 1200 1600
              2000 2048 2200 2500 3000 3500 4096
              4500 5000 5500 6000 7000 8192 10000
              16384 32768 65536 1000000 1500000)
    time and run p9_adjustbufferlen
```

Figure 2. Stage One Code Generation

In pseudo-code, Stage One code generation looks something like the code in Figure 2. The automatically generated codebase always found parameter values comparable to the hand-tuned values: see the Final Autogen code row in Figure 1.

4. Stage Two Code Generation

Stage Two generation is probably more accurately called "tuning for an application." By plugging in an appropriate producer and consumer that do real work, we refined the codebase from Stage One to be more appropriate for the application in question. Despite the fact that we generated interference-free code in Stage One, real applications do real work in addition to the synchronization and communication of HSPC code, and that real work caused application-specific interference. We introduce portable techniques (below) to mitigate interference by the producers and consumer.

4.1. Minor Techniques

Shrink the Event Size: This technique is very application specific and may not be applicable, but reducing the memory footprint of the event limits memory interference. Reducing the Python profiler event from 128 bytes to 24 bytes allowed us to achieve scalability much more quickly.

Code with Thread-Neutral Data Structures: Container libraries such as the C++ Standard Template Library are silent on the issue of threads. By using the OpenContainers library[12], a portable and thread-neutral library, we can proceed in confidence that *heap contention* (multiple concurrent calls to malloc being serialized) will not cause excessive interference. Replacing STL's map with an OpenContainer's HardHashT improved the Python profiler runtime performance by 20%. Issues related to this are discussed in [9] and [12].

Use Processor Affinity: Although we didn't need processor affinity in Stage One (because we could generate

interference-free code without it), locking a producer to one CPU and locking the consumer to another CPU does seem to mitigate interference from the OS (as processes/threads migrate). In general, this seemed to give about a 5% speedup overall.

4.2. Alternate POSIX Synchronization Primitives

In Stage One, we used POSIX condition variables for synchronization. Switching to other primitives significantly improved scalability on certain platforms. By staying with POSIX synchronization primitives, we also can have a large degree of portability. We used three different types of POSIX synchronization: condition variables, mutexes[2] and spinlocks[1, 10] with three different variations on the spinlock:

Standard: Use as provided by POSIX implementation

Local spinlock: The local spinlock does a POSIX trylock and if it cannot get the lock, it does exponential backoff spinning on the local variable to stay off the bus. This local spinlock is completely portable.

Hyperthread-aware spinlock: Similar to the local spinlock. Instead of spinning on the local variable, it executes the Intel pause instruction, while continuing the exponential backoff strategy. The Intel documentation[4] hints that a pause instruction is SMT friendly and will get better performance for spinlocks on SMT machines. This is a very non-portable construct, but is provided (hopefully) to mitigate severe SMT interference.

The previous minor techniques helped us mitigate interference, but tuning the buffer sizes (because event sizes have changed from Stage One) and varying synchronization primitives were by far the most useful techniques.

Spinlocks added considerable overhead, taking a full CPU when running, but allowed most tests in Stage Two to reach varying degrees of scalability with smaller buffer sizes. Our local spinlocks and hyperthread-aware spinlocks made little difference for the HSPC. Condition variables and mutexes were much less intrusive to a system overall, but typically needed large buffers to balance well. Spinlocks thus appear to be more suited to low-latency applications, while condition variables and mutexes are more suited to applications that must conserve processing resources.

Figures 3–8 show the results of varying the buffer size and synchronization primitives across the different architectures. In all of these test cases, the producer computes (in Python) a recursive Fibonacci function and generates events (each function call enter and return) that a consumer (in another thread) consumes, computing profiling information. We are sending on the order of half a million events per second. In all cases, the producer runs taking a full CPU,

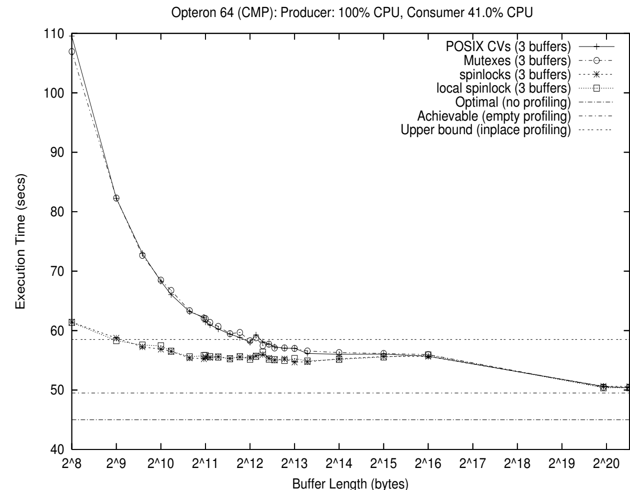

Figure 3. CMP: Dual Core Opteron

usually 100%. In most cases (except where noted), the consumer takes about 41% of a CPU consuming events and computing profiling information. This is an estimate from observing CPU usage as reported by top. [3]

In each figure, we show three limit lines:

The *Optimal* line is how fast the test case (Python computing recursive Fibonacci numbers) can compute without profiling. It is the execution time of the best we can do.

The *Achievable* line shows the cost of generating the minimal profiling information and putting the event into an empty buffer. We aspire to the *Achievable* line, and we claim perfect balancing if we reach that.

The *Upper bound* line is the execution time of the test case where all computing (both the recursive fibonacci work and the profiling work) is done in a single thread (in other words, no consumer and producer threads running). If we cannot do better than *Upper bound*, there is no reason to even try to offload profiling work to another processor: it is faster just to do the profiling work inplace.

Chip Multi-Processors. Consider Figure 3 for the CMP machine: **Opt64-CMP**. As buffers got larger, the timings for all the synchronization primitives tended to converge. The dual-core Opteron machine performed well with large buffer sizes. Although very large buffers gave the best results and asymptotically approached *Achievable*, the spinlocks with much smaller buffers (2048 and 4096 bytes) gave acceptable performance with much better latency.

The **Opt64-CMP** machine load-balanced well. Unfortunately, we only had access to one CMP machine (**G5-CMP** was not available at this stage), but it seemed to perform well in all of our tests. But this is to be expected, as shared

[3] In the spinlock cases, the consumer appears to take 100% of a CPU, but is only actually doing 41% of a CPU of work (the remaining 59% is the spinlock time).

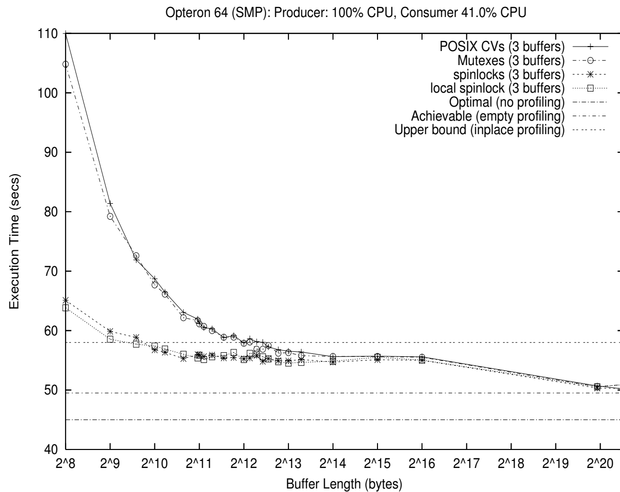

Figure 4. SMP: Two Processor Opteron

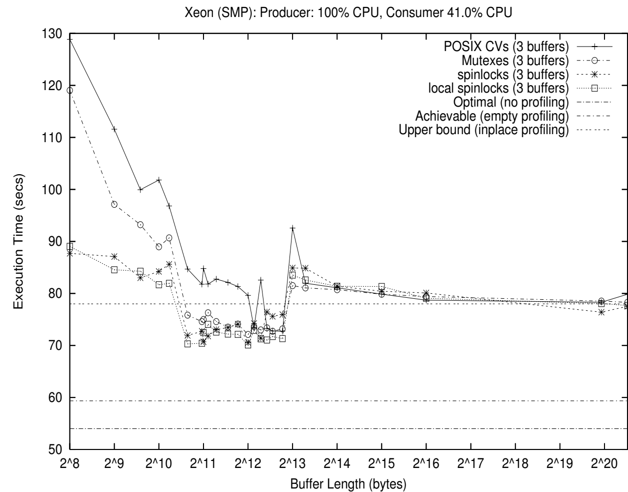

Figure 6. SMP: Two processor Xeon

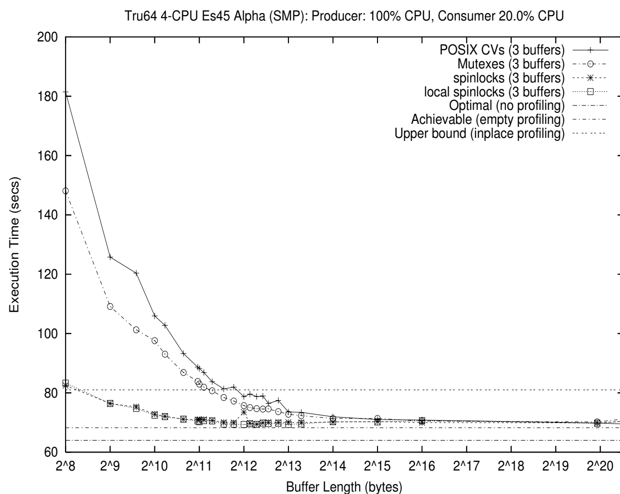

Figure 5. SMP: Four processor Alpha EV6.7

caches are very tightly integrated on-chip, and there are no shared execution units (unlike the SMT case).

Shared Memory Multi-Processors. As hinted at by Stage One, SMP load-balanced reasonably well. Both **Opt64-SMP** (Figure 4) and **Alpha-SMP** (Figure 5) were able to get nearly perfect balancing. The **Alpha-SMP** did best with small buffers and local spinlocks, but still very well at large buffers. The **Opt64-SMP** does best with large buffers (and in fact, looks very similar to **Opt64-CMP**). Both **Opt64-SMP** and **Alpha-SMP** were able to offload all of the producer's work, balancing almost perfectly.

Although **Xeon-SMP** balanced well in Stage One code generation, it performed poorly here (see Figure 6). **Xeon-SMP** still achieved some parallelism, but by no means approached perfect balancing. We believe the problem here is the memory subsystem is overloaded (similar to [11]).

Simultaneous Multi-Threaded Processors. Our results showed little parallelism. On both the **Xeon-SMT**(Figure 7) and the **P4-SMT**(Figure 8) machines, there was almost no work offloaded, even with the hyperthread-aware spinlock.

We expected because SMTs have tightly integrated caches that HSPCs would perform well for small buffer sizes. That was not the case, but some recent work corroborates our findings for SMT machines. [11] discusses how poorly SMT works on real network servers: they found (for their context) that the best you can expect is about 30%-50% speedup, but a slowdown is possible as well.

5. Related Work

FFTW[3] is the inspiration for much of the HSPC work. The codelets FFTW generates are similar to Stage One codebase samples. HSPC's idea of parameterizing, running and timing code came from [3], but similar work has been done by others[8, 16]. The ATLAS[16] work discusses optimizing buffer sizes in real software. The SuperOptimizer work [8] explores the entire state space of a problem, although it tends to be looking for surprising possibilities in code.

The work in [9] is directly related: many of the issues discussed there we revisit in the context of HSPC, especially the issues we have with SMT architectures.

The [7, 11] papers were an important sanity check, corroborating difficulties of SMT scaling on real hardware.

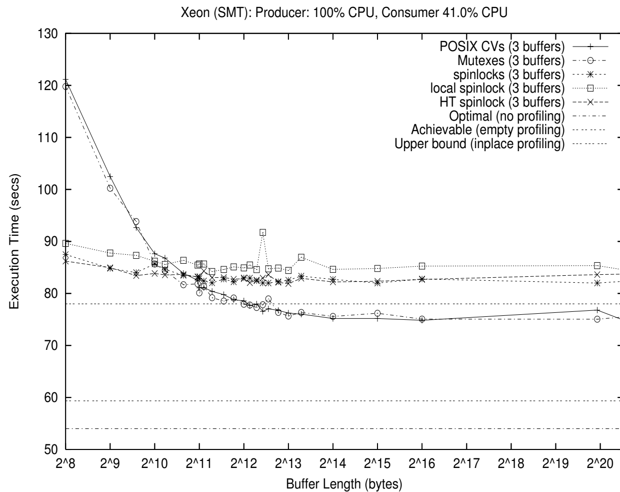

Figure 7. SMT: Hyperthreaded Xeon

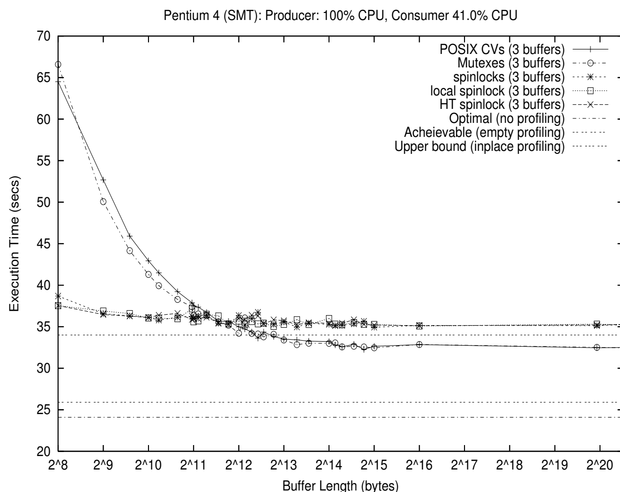

Figure 8. SMT: Hyperthreaded P4

6. Conclusion

This paper described the steps taken to optimize HSPC code on CMP, SMP and SMT architectures and showed that doing so can dramatically reduce the overhead associated with synchronization and communication. Demonstrating further limits of SMT systems[11], we discovered that limited availability of functional units in SMT systems prevents perfect speedup when adding threads; the CMP/SMP results imply that this failure is specific to SMT architecture and not due to the nature of the HSPC problem.

Although the HSPC results were derived from a Python profiling system, they can easily be applied to other systems software because the producer/consumer relationship is so fundamental. By keeping the HSPC codebase portable, other systems can use and tune an HSPC for its particular needs (optimizing event latency, reducing memory footprint) without having to rediscover all issues presented here.

References

[1] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990.

[2] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley, Boston, MA, 1997.

[3] M. Frigo. A fast fourier transform compiler. In *Conf. on Programm. Lang. Design and Impl.*, pages 169–180, 1999.

[4] R. Gerber and A. Binstock. *Programming with Hyper-Threading Technology*. Intel Press, 2004.

[5] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, New York, NY, USA, 1982. ACM Press.

[6] C. L. Jeffery. *Program Monitoring and Visualization: An Exploratory Approach*. Springer-Verlag, NY, 1999.

[7] D. Kim, S. S. wei Liao, P. H. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In *4th Symposium on Code generation and optimization*, page 27, Washington, DC, USA, 2004. IEEE.

[8] H. Massalin. Superoptimizer: a look at the smallest program. In *2nd Conference on Architectual support for programming languages and operating systems*, pages 122–126, Los Alamitos, CA, USA, 1987. IEEE.

[9] L. K. McDowell, S. J. Eggers, and S. D. Gribble. Improving server software support for simultaneous multithreaded processors. In *9th Symposium on Principles & Practices of Parallel Programm.*, pages 37–48, 2003.

[10] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[11] Y. Ruan, V. S. Pai, E. Nahum, and J. M. Tracey. Evaluating the impact of simultaneous multithreading on network servers using real hardware. In *ACM SIGMETRICS Conference on Measurement and modeling of computer systems*, pages 315–326, 2005.

[12] R. T. Saunders. Opencontainers: A portable, thread-neutral library. *www.amalgama.us/oc.html*.

[13] R. T. Saunders, C. L. Jeffery, and M. Wilder. Python profiling and visualization. In *PyCon DC 2005*, 2005.

[14] H. Sutter. A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3), Mar 2005.

[15] G. van Rossum and F. L. Drake. Extending and embedding the python interpreter, release 2.3.4. *python.org*, May 2004.

[16] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Comput.*, 27(1-2):3–35, 2001.