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

Richard T. Saunders
Rincon Research Corporation
101 N. Wilmot, Tucson AZ 85711
rts@rincon.com

Clinton L. Jeffery
Department of Computer Science
University of New Mexico, Las Cruces NM
jeffery@cs.nmsu.edu

Derek T. Jones
Rincon Research Corporation
101 N. Wilmot, Tucson AZ 85711
dtj@rincon.com

Abstract

The producer/consumer relationship is fundamental in parallel systems software, but writing an efficient, portable High-Speed Producer Consumer (HSPC) is surprisingly difficult. When offloading work from one processor (producer) to another processor (consumer) at a high rate, naively written code can easily perform suboptimally, taking more time than if all the work were done by a single CPU. This paper explores generating 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 simultaneous-execution, memory, and synchronization 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, even when using machine-specific synchronization primitives.

1. Introduction

The producer/consumer relationship is a canonical pattern among event-based commu-
nicipating processes/threads[2][3]. 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[17], a profiling system that sends events when certain frequent activities occur[8]. Such systems require the fastest producers/consumers possible, but writing a high-speed producer/consumer is surprisingly difficult (see Related Work as well as [6]).

For monitoring systems such as profilers, event processing is overwhelming. Consider the events that profiling systems typically generate: function entry and exit, instruction counts, etc. Such systems can easily generate hundreds of thousands or even millions of profiling events per second. For example, computing a recursive Fibonacci(35) generates a combined total of about 18 million function enter/exit events in 25 seconds on a 3 GHz Pentium 4, about 7.3e5 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[7]), while others (such as Alamo[9]) 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 multicore processors[14][20]. 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[17], 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.

For such a simple goal, a number of obstacles arose during the course of development, requiring trial-and-error and research for which platform-specific[6] information was required. 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 multi-processor 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

For HSPC, portability means code that not only works, but also scales across multiple platforms.

For this investigation, we used the percentage of work offloaded to the consumer as our metric. We measured the total amount of work done by summing the times of the ideal baseline producer and ideal baseline consumer. For example, for Opt64-CMP, the baseline producer ran for 15.28 seconds, and the baseline consumer ran for 15.61 seconds, for a total of 30.89 seconds of work. Ideally, with no interference or overhead, the producer and consumer would divide the work equally, corresponding to a metric of 50% work offloaded, for perfect balancing. The baseline was either the baseline consumer or producer, depending on which was slower (as the slower one became the bottleneck when the two ran together).

The percentage of work we can offload is:

$$1.0 - \left( \frac{\text{time of the baseline}}{\text{total amount of time}} \right)$$

If the final run time of the producer remains unchanged, we are offloading 50% of the total work to the consumer. Thus, for Opt64-CMP, we were indeed able to achieve very nearly ideal results, offloading $1 - \frac{15.61}{30.89}$, or about 50%, of the work to the consumer—we achieved nearly perfect balancing.

### 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).

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 consumer and baseline producer were run in parallel (test case p0_ADD_parallel) with nothing deliberately shared. At row p0_ADD_parallel, note the times in Figure 1: The CMP cases, G5-CMP and Opt64-CMP, showed some to no interference compared to the massive 2.7x–5.4x slowdown (over the baseline) of the SMP and SMT cases. Some platforms, like P4-SMT, Opt64-SMP, and Xeon-SMT varied widely at this step, running 5x slower than baseline on some runs, balancing ideally at other times, demonstrating unpredictable per-run interference.

3.2.1. Buffer Layout

At this step, the producer and consumer each had their own (non-shared) buffers allocated separately via malloc. Unfortunately, this meant we had no control over where the buffers were laid out in memory; we were dependent on the allocator. On some runs, there seemed to be no interference, because malloc allocated buffers that didn’t happen to overlap cache lines, whereas other runs, on the exact same architecture, had buffers that completely overlapped cache lines.

To combat unpredictable buffer layouts in memory, we ensured that the produce and consume buffers were allocated in a single allocation; thus, the interference of each buffer relative to the other could be controlled. The codebase p1_singlealloc in Figure 1 contains the fixes for this problem. Note that these fixes brought us back to baseline times (except for SMT, see section 3.2.4). Strictly speaking, p1_singlealloc should be the new baseline (since on many platforms, performance improved slightly during the code reorganization), but we wish to emphasize the evolutionary nature of the HSPC and keep the original
### 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 3.2.4) for that platform. Darkened bars represent the addition of a feature.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Baseline Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>G5-CMP</td>
<td></td>
</tr>
<tr>
<td>Opt64-CMP</td>
<td></td>
</tr>
<tr>
<td>Opt64-SMP</td>
<td></td>
</tr>
<tr>
<td>Alpha-SMP</td>
<td></td>
</tr>
<tr>
<td>Xeon-SMP</td>
<td></td>
</tr>
<tr>
<td>Xeon-SMT</td>
<td></td>
</tr>
<tr>
<td>P4-SMT</td>
<td></td>
</tr>
</tbody>
</table>

#### 3.2.2. False Sharing

The next step was to merge the data structures for the consumer and producer, since at some point, this must happen for them to communicate. Note that this was only an organizational step that should not have affected the run time. There was no sharing of buffers. There was no intentional communication between the producer and consumer, as they accessed the shared HSPC data structure disjointly, as if there were two separate data structures. The producer only accessed the read buffer portion, and the consumer only accessed the write buffer portion. In other words, any interference at this step would have been SEI.

The numbers for introducing data structure sharing are in `p2_ADD_shareddatastructure`. This addition caused a major slowdown (2.1x–8x) on all platforms. The fix for the slowdown was to eliminate two instances of false sharing found in the shared HSPC data structure: False sharing [6][14] occurs when two threads access distinct memory addresses that map to the same cache line—in this case, it was devastating, as the bus is overwhelmed by memory traffic when synchronizing cache lines.

Instance 1: The read buffer pointer and write buffer pointer were right next to each other in the HSPC data structure so as to share a cache line. This was a huge problem, because every get and put interfered. By adding padding between the read portions and the write portions of the data structure, we eliminated most of the interference.

Instance 2: Since we allocated the HSPC data structure and the buffers with a single allocation (to control buffer layout), the write buffer pointer happened to be right next to the buffers. We needed extra padding between the two so that accessing the front of the buffer didn’t interfere with accesses to the write buffer pointer.

The effects of adding the extra padding are shown in Figure 1 at `p3_rmfalsesharing`. On most platforms, we returned to baseline times (except for SMT, see section 3.2.4).

#### 3.2.3. Stack Layout

The P4-SMT platform had an additional complication affecting its scalability. The problem was a very platform-dependent problem related to stack layout. It has been noted in...
the literature[6][14][18] and is called the 64-kilobyte Aliasing Problem: “When Intel processors based on the Intel Netburst microarchitecture load caches, they keep track of cache data as offsets from a 64-kilobyte boundary. That is, if two items start 512 bytes in from a 64-kilobyte address, they will collide in cache, and the oldest one will be expelled by the new memory reference. ... The problem is that when threads are started up, their stacks always start on a 64-kilobyte boundary, so the items allocated on the thread’s stack will frequently collide in cache with those of another active thread.[6]” We saw exactly this from [6]: The highly accessed stack variables in the producer/consumer threads overlapped, and we saw a false sharing effect. 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, (but see section 3.2.4), and had a slight effect on the other platforms.

At this step, we had eliminated almost all SEI for SMP and CMP architectures—their timings at p4_adjuststack are in line with the baseline timings.

3.2.4. SMT Scalability

Although we eliminated Simultaneous-Execution Interference (SEI) 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[11][18] for SMT parallelism.

The work in [18] 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 from it.

The problem is uncovered early investigating the Simultaneous-Execution Interference—that 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[4], rewriting assembly code to avoid execution-unit sharing[10] or other compiler techniques[12]), 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\textsuperscript{1} eventually (see next section). The three shared buffers were allocated with a single allocation to avoid the buffer layout problems from Section 3.2.1.

Consider p5_ADD_sharedbuffer: Adding the shared buffers drops us about 1.2x–2.3x off the baseline.

3.3.1. False Sharing

In this case, the false sharing occurred between the buffers: at the starts and the ends. 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\textsuperscript{2}[2][3]

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.

3.4.1. False Sharing

Since we were using triple buffering, synchronization is required at every buffer boundary. The HSPC data structure contained a C array of POSIX condition variables. False sharing can occur if two condition variables are in the same cache-line, or if some other element of the HSPC data structure shares a cache-line with the condition variables.\textsuperscript{3}

To eliminate possible false sharing, we added padding to the HSPC data structure so that no condition variables were in the same cache line. The results are in p8_rmfalsesharing, and made a difference for Opt64-CMP and Xeon-SMP, even though Alpha-SMP and Xeon-SMT slowed down. The real effect is that this codebase produced more predictable run times.

3.4.2. Buffer Issues

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 machine Opt64-CMP, we achieved interference-free code. The final run times seen in p9_adjustbufferlen are almost

\textsuperscript{1}Double, triple or $n$–buffering means round-robin through $n$ buffers.

\textsuperscript{2}We will consider the effects of other POSIX synchronization primitives in Stage Two of the HSPC. We also considered POSIX semaphores instead of condition variables, and found that the implementations of the two are extremely comparable for HSPC.

\textsuperscript{3}This is especially important for Stage Two, when we need to make sure spinlocks stay local[15].
perfect (Opt64-CMP is even slightly better than the baseline producer or consumer). We achieved very nearly perfect balancing for the G5-CMP.

For the SMP machines (namely Opt64-SMP, Alpha-SMP and Xeon-SMP), the results 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 is slightly disappointing: early numbers showed Alpha-SMP balancing perfectly, but subtle changes to compiler flags dramatically changed the run time of the producer relative to the consumer: For example, the baseline consumer would run at twice the speed of the baseline producer. The current run times in Figure 1 seem to give the most stable and experimentally repeatable results.

Unfortunately, for the SMT machines, we were never able to eliminate all interference, specifically SEI. For P4-SMT, the work offloaded is \(1 - \frac{2286}{16.54 + 18.62} = 35\%\). For Xeon-SMT, it is \(1 - \frac{21.32}{14.08 + 17.28} = 33\%\). Thus we were limited to offloading only about 33–35% of the work (compared to perfect balancing, offloading 50% of the work). The \(p_1\_singlealloc\) 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\[13\] 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 \(p_3\_rmfalsesharing\) 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 \(p_4\_adjuststacklayout\). 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 \(p_9\_adjustbufferlen\) 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.

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).
Figure 2. Stage One Code Generation

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 code-base 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 consumers without resorting to non-portable compiler techniques (mentioned earlier) such as [4][10][12]. We showed that we can load-balance very well on CMP and SMP architectures, but SMT architectures seem to fare even worse than at Stage One.

In the next sections, we explore these techniques.

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 thread-safety and thread-neutrality. By using the OpenContainers library[19], 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 Hard-HashT improved the Python profiler runtime performance by 20%. Issues related to this are discussed in [14] and [19].

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). Affinity also limits cache and TLB (translation lookaside buffer) interference as local memories stay local. 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][3] and spinlocks[1][15] 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: The hyperThread-aware spinlock is similar to the local spinlock, but instead of spinning on...
Figure 3. CMP: Dual Core Opteron

Figure 4. SMP: Two Processor Opteron
Figure 5. SMP: Four processor Alpha EV6.7

Figure 6. SMP: Two processor Xeon
the local variable, it executes the Intel `pause` instruction, while continuing the exponential backoff strategy. The Intel documentation[6] 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 to mitigate (hopefully) 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 3–10 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, 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`. 5

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.

### 4.2.1. 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 the buffer size. Although very large buffers (1 MB and 1.5 MB) 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 the best in all of our tests. But this is to be expected, as shared caches are very tightly integrated on-chip, and there are no shared execution units (unlike the SMT case).

### 4.2.2. 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

---

5In 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).
Opt64-CMP). Both Opt64-SMP and Alpha-SMP were able to offload all of the producer’s work, balancing almost perfectly.

Although Xeon-SMP load-balanced perfectly 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 [18]).

4.2.3. Simultaneous Multi-Threaded Processors

Our first results showed little parallelism. On both the Xeon-SMT (Figure 7) and the P4-SMT (Figure 8) machines, there was almost no work offloaded.

Since local spinlocks gave the needed performance for some SMP machines (as seen in the previous section), we decided to try the hyperthread-aware spinlock. The difference, unfortunately, was negligible.

One important observation is that the amount of work we offloaded to the consumer is small for most of our tests: The consumer appears to only do about an extra 41% of work (as seen in our top displays). What if the consumer happens to do more intense computation? Can we offload any more work? One early version of the Python profiler was written so that each of the producer and consumer took an entire CPU: we will call this the “heavy” profiler (whereas the profiler used everywhere else in this paper
5. Related Work

FFTW[5] 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 [5], but similar work has been done by others[13][16]. The ATLAS[16] work discusses optimizing buffer sizes in real software. The SuperOptimizer work [13] explores the entire state space of a problem, although it tends to be looking for surprising possibilities in code (In HSPC work, we have already factored out the surprising possibilities and are simply looking for the best-behaved code).

The work in [14] 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 [18] paper was an extremely important sanity check on our work, corroborating the difficulties in SMT scaling for real hardware.

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[18], 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
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 the issues presented here.

References