Proceedings of 12th Intl Conference on Parallel Architectures and Compilation Techniques, September 2003.

Initial Observations of the Simultaneous Multithreading Pentium 4 Processor

Nathan Tuck Dean M. Tullsen

Department of Computer Science and Engineering
University of California, San Diego
{ntuck,tullsen}@cs.ucsd.edu

Abstract

This paper analyzes an Intel Pentium 4 hyper-threading processor. The focus is to understand its performance and the underlying reasons behind that performance. Particular attention is paid to putting the processor in context with prior published research in simultaneous multithreading – validating and re-evaluating, where appropriate, how this processor performs relative to expectations. Results include multiprogrammed speedup, parallel speedup, as well as synchronization and communication throughput.

The processor is also evaluated in the context of prior work on the interaction of multithreading with the operating system and compilation.

1 Introduction

The Intel® Pentium® 4 Hyper-Threading processor [13] is the first general-purpose microprocessor to ship with features of simultaneous multithreading (SMT). Simultaneous multithreading appeared in research publications in the early to middle 1990s [8, 6, 17, 9, 29, 31, 28] and a large body of work has followed in the years since. It remains an extremely active area of research. And yet, the introduction of the Pentium 4 processor represents the first opportunity to validate and compare any of those results against real hardware targeting the high performance, general purpose computing market. Simultaneous multithreading is a specific instance of hardware multithreading, which encompasses an even larger body of work, but has likewise not previously appeared in a mainstream general-purpose microprocessor.

This paper seeks to measure various aspects of the performance of the simultaneous multithreaded Pentium 4 and to understand the reasons behind the performance. Particular emphasis is placed on putting this implementation into context with respect to the SMT research. We want to know how well it meets expectations raised by the research, where it differs, where it doesn't differ. We will examine where it diverges quantitatively from the research models, and whether it diverges qualitatively with the expected performance characteristics.

This work uses a mixture of single-threaded, multiprogrammed, and parallel workloads, as well as component benchmarks to evaluate this processor architecture. We report multithreaded speedups and low-level performance counters to attempt to better understand the machine. We examine multiprogrammed speedup, parallel speedup, synchronization and communication latencies, compiler interaction, and the potential for operating system interaction.

This paper is organized as follows. Section 2 discusses related work. Section 3 describes the Intel Pentium 4 hyper-threading architecture, and Section 4 describes our measurement and reporting methodology. Section 5 presents a wide variety of performance results, and Section 6 concludes.

2 Related Work

The focus of this paper is the Pentium 4 Hyper-Threading processor architecture [7], and particularly the implementation and effectiveness of simultaneous multithreading, or hyper-threading, described in some detail in Marr, et al. [13].

The concept of hardware multithreading has been around since the 1960's, starting with the I/O controller of the CDC 6600 [24]. Other important machines include the Denelcor HEP [21] and the Cray/Tera MTA [3], both projects led by Burton Smith. Other early publications include Flynn, et al.'s [5] description in 1970 of a multithreaded processor with distinct front ends but a time shared execution unit, and Shar and Davidson's [19] analysis in 1974 of a pipelined processor that allowed each stage to be executing in a different context. Other important work includes Laudon, et al.'s pipelined multithreaded model [10], and the coarse-grain multithreaded processor for Alewife [2]. All but the last describe implementations of the fine-grain (context switch every cycle) multithreaded execution model.

Simultaneous multithreading is a natural extension to the earlier multithreading models. It differs from those models in the following way. On a superscalar processor, the SMT execution model assumes the ability of the architecture to issue instructions from multiple threads to the functional units each cycle. All previous models of multithreading incorporate the notion of a low-level context switch, even if it happens every cycle. SMT loses the notion of a context switch, as all hardware contexts are active simultaneously, competing actively for all processor resources. This competition maximizes the utilization of the processor resources.

Early work incorporating at least some aspects of what became known as simultaneous multithreading include [8, 6, 17, 9, 29, 31, 28, 20]. The focus of this research will be to compare the Pentium 4 implementation in particular with the SMT execution model proposed by Tullsen, et al. [29], and the architecture first fully described in [28]. That execution model and architecture has since re-appeared in a plethora of papers from many research groups.

The Pentium 4 architecture is the first general-purpose processor delivered with simultaneous multithreading, but other implementations include the announced, but never delivered Alpha 21464 [4] and a network processor intended for routing applications from Clearwater Networks. Sun's MAJC processor [25] is a fine-grain multithreaded architecture which also features support for speculative multithreading.

This paper compares the Intel Pentium 4 architecture and performance with the published SMT research architecture model. It also examines quantitative and qualitative results from research examining parallel performance [11], synchronization and communication performance [30], compiler interaction [12], and operating system interaction [22, 18, 23].

3 Pentium 4 Hyper-threading Architecture

The Intel Pentium 4 processor [7] is the first commercially available general-purpose processor to implement a simultaneously multithreading core.

The Pentium 4 microarchitecture is that of a deeply pipelined out-of-order processor. It has a 20 cycle branch misprediction penalty. The core can retrieve 3 micro-operations (μops) per cycle from the trace cache, execute up to six per cycle and graduate up to three per cycle. There is a 128-entry renamed register file and up to 126 instructions can be in flight at a time. Of those instructions, 48 can be loads and 24 can be stores. It has an 8KB 4-way set associative direct mapped L1 data cache with a 2 cycle load-use latency and a 256 KB 8-way set associative unified L2 cache. The L1 instruction cache is a 12K entry trace cache with a 6 μop line size.

The Pentium 4 supports a dual-threaded implementation of simultaneous multithreading called Hyper-Threading Technology. The Linux operating system treats the system as a dual-processor system, each processor maintaining a separate run queue. This approach to multithreading minimizes the changes to a conventional multiprocessor-aware operating system necessary to manage the SMT hardware.

The primary difference between the Hyper-Threading implementation and the architecture proposed in the SMT research [28] is the mode of sharing of hardware structures. While the SMT research indicates that virtually all structures are more efficient when shared dynamically rather than partitioned statically, some structures in the Hyper-Threading implementation such as the ROB entries and load and store buffers are statically divided in half when both threads are active. The reasons for this difference are at least two-fold. First of all, static partitioning isolates a thread from a poorly behaving co-scheduled thread more effectively – see [27] for an example of two-thread thrashing in a shared instruction queue, as well as alternative solutions to static partitioning. Second, any negative effects of the partitioned approach are minimized with only two hardware contexts. The technique employed in the Pentium 4 does not scale well to a larger number of hardware contexts, such as the eight threads that were typically modelled in the SMT research. However, with two threads, a scheme that uses the entire instruction queue in single-thread mode, and divides it in half for dual-thread mode provides a reasonable approximation of the performance of a dynamically shared queue. With more threads, the static partitioning would likely hamper performance excessively.

Accesses to several non-duplicated, non-partitioned parts of the Pentium 4 pipeline, such as the trace cache and main instruction decoder are handled in a round-robin manner. Similarly, Tullsen, *et al.* [28] proposed an architecture that time-shares the front end of the processor (i.e., the front end is fine-grain multithreaded), but mixes instructions (simultaneous multithreading) beginning with the queue stage. That paper also examines architectures with limited multithreading (e.g., two threads rather than eight) in the front end.

Using the tool LMbench [15] we measure the L1, L2, and main memory latencies of the processor as 0.794ns, 7.296ns and 143.9ns. This corresponds to 2, 18 and 361 cycles of latency, which in the case of the caches correlates well with Intel's claims about the microarchitecture. We were able to measure main memory bandwidth for our setup as between 1.24 GB/s and 1.43 GB/s using the Stream benchmark [14]. Neither aggregate memory bandwidth nor latency were observed to degrade measurably when two threads ran simultaneous copies of LMbench or Stream.

4 Methodology

All experiments were run on a single processor Hyper-Threading 2.5 GHz Pentium 4 processor with 512 MB of DRDRAM running RedHat 7.3 Linux kernel version 2.4.18smp.

For these experiments, we use the SPEC CPU2000 benchmarks, a subset of version 3.0 of the NAS parallel benchmarks, and expanded versions of the Stanford SPLASH2 benchmarks. The SPEC 2000 and NAS benchmarks were compiled with Intel's 6.0 Linux compilers. All optimizations for SPEC were as close as possible to the base optimization level in Intel's SPEC submissions with the exception that no profiling data was used. The SPLASH2 benchmarks are compiled with gcc version 3.2 with full optimization enabled.

For the SPLASH benchmarks, we did not use the default inputs, rather we first sized up the inputs so as to increase the amount of execution time and working set size. Each benchmark was sized up so that total runtime was between 30 seconds and 2 minutes. This was necessary to get accurate timing measurements. Actual parameters for each benchmark are given in Table 1. Every component was run in three different configurations. First we ran a single threaded version of the benchmark so as to determine a baseline expected time for benchmark completion. Then we ran a dual threaded version of the benchmark so as to determine parallel speedup. Then we ran a single threaded version of each benchmark against every other benchmark (including itself) to obtain a heterogenous workload speedup.

For SPEC we first ran a fully instrumented SPEC run to be sure that the binaries we were generating for benchmarks were close to the baseline SPEC submission. We then ran every SPEC benchmark against every other SPEC benchmark with the full inputs in order to determine heterogeneous speedups.

For heterogeneous workloads in both SPLASH and SPEC, we used the following methodology for runs so as to avoid overmeasuring simultaneous job startup and early completion artifacts. We launched both jobs simultaneously from a parent process which then waited for either job to complete. When either job completed, it would be immediately relaunched by the parent, until at least 12 total jobs had been launched and completed. The time required for the last job to complete was thrown away so as to eliminate any jobs that were not run completely in parallel with another thread. We ensured that we always had at least three instances of even the longest benchmarks that completed before the other threads finished. The completion times for each of the remaining jobs was then averaged. This methodology has several advantages. It eliminates end effects (which otherwise potentially insert significant single-thread execution into our multithreaded results) and it takes the average of several staggered relative starting times. We found that in general staggered relative starting times was not an important factor in measurement – the average variance in program run times for all SPEC benchmarks as a factor of start time was less than 0.5% and the average maximum variance in run times was 1%.

Benchmark	Altered Parameters
barnes	nbody=65536 tstop=0.3
fmm	65536 particles
ocean - contiguous	1026×1026 grid
ocean - noncontiguous	1026×1026 grid
radiosity	large room dataset
raytrace	-a32 -m64
water - nsquared	5832 molecules
water-spatial	5832 molecules

Table 1. Splash Application Expanded Run Parameters

For parallel benchmarks we report the speedup used in the reporting methodology of the benchmark. In the case of SPLASH this can exclude some sequential parts of the code. For the heterogeneous workloads, we report the sum of the two speedups. That is, if A and B run together, and A runs at 85% of its single-thread execution time, and B runs at 75%, the reported speedup is 1.6. This gives the expected average response time improvement over a workload where the same combination of the two jobs time-share a single-threaded processor, assuming approximately equal run times, and no context-switch overhead.

We attempted to determine the performance of synchronization on the hyper-threading Pentium 4, with particular attention to whether hyper-threading makes synchronization profitable at a finer granularity than a traditional SMP. In order to do this, we modified the code used in [30] to implement x86 spin locks with the *pause* instruction in them as is recommended by Intel [1]. The *pause* instruction will de-pipeline a thread's execution, causing it to not contend for resources such as the trace cache until the *pause* has committed. However, unlike *halt*, *pause* does not release statically partitioned resources in the pipeline. Further details of that benchmark are presented in Section 5.4.

5 Results

This section examines a variety of aspects of the performance of the multithreaded Pentium 4 processor. It covers multiprogrammed speedups, parallel program speedups, the performance of synchronization and communication, the effects of workload scheduling, architecture-compiler interactions, and simulation results.

5.1 Impact of Static Partitioning of the Pipeline

The Pentium 4 implementation dynamically shares very few resources. The functional units are shared, as are the caches and branch prediction logic. However, when multiple threads are active, the following structures are statically split in half, each half being made available to a single thread: the reorder buffer, instruction queues, and load/store queues. Thus, in multithreading mode, there are now two effects that prevent the workload from achieving ideal speedup (i.e., two times the single-thread throughput). One is the effect of competition for statically partitioned resources, and the second is the effect of competition for the dynamically shared resources.

We discern between these two effects by measuring the impact of static partitioning on a single thread, when there is no dynamic sharing. Because the Pentium 4 only partitions these structures when there are actually two threads in execution, we use a dummy thread that uses virtually no resources (it makes repeated calls to the *pause* instruction) to force the processor into partitioned mode.

We found that on average, the SPECINT benchmarks achieve 83% of their non-partitioned single-thread performance, and the SPECFP benchmarks achieve 85% of their single-thread performance. Those benchmarks hit worst are *eon*, which is limited to 71% of peak, and *wupwise*, which is limited to 72% of peak. The benchmarks that perform best in this environment are *mcf*, *art* and *swim* at 93%, 97% and 98% of peak respectively. *eon* and *wupwise* have relatively high instruction throughput of 0.9 and 1.2 IPC respectively, while *mcf* and *swim* have relatively low IPCs of .08, .2 and .4 (all IPCs measured in μops). Not unexpectedly, then, those applications with low instruction throughput demands due to poor memory performance are less affected by the statically partitioned execution resources. See Figure 1 for a summary of results from these runs.

5.2 Multiprogrammed Multithreaded Speedup

In an SMT processor with only two threads, there are exactly two ways to achieve multithreaded speedup — running two independent threads, or running two cooperating threads. We examine the former case in this section, the latter in the following section. The multiprogrammed (independent threads) performance, in some ways, gives the clearest picture of the multithreaded pipeline implementation, because performance is unaffected by synchronization, load-balance, and inefficient parallelization issues.

We ran the cross-product of all 26 SPEC benchmarks, as well as the cross-product of all single-threaded SPLASH2 benchmarks. Those results are too voluminous to present here, so we show a box and whisker plot of the SPEC data which illustrates the median, middle 50% and outliers in

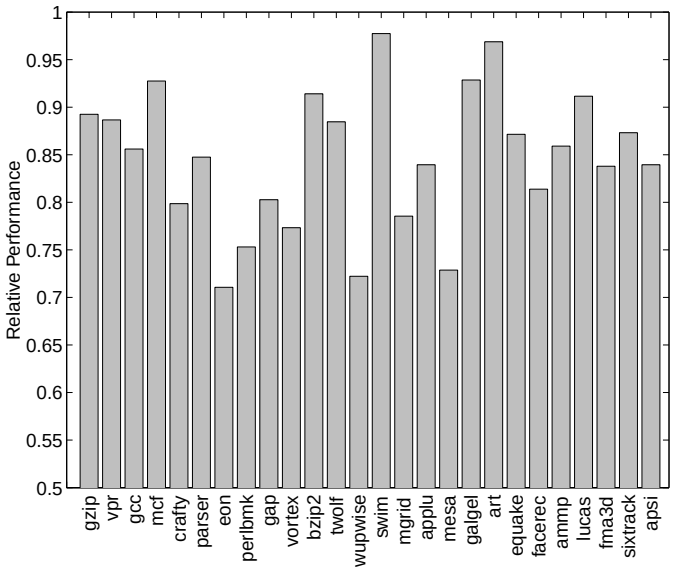

Figure 1. Performance of SPEC benchmarks run against the pause thread relative to their peak performance.

speedups for each test in Figure 2. For each benchmark, this figure shows the range of combined speedups for that benchmark paired with all other benchmarks.

The way to read these plots is that for each datapoint, the box outlines the middle 50% of the data (also known as the inter-quartile range or IQR), the line inside the box is the median datapoint, and the "whiskers" or dotted lines extend to the most extreme datapoint within 1.5*IQR of the box. Points which fall outside of 1.5*IQR are classified as outliers and appear as crosses on the plot. We do not attempt to claim any statistical significance to presenting the data in this manner, but we feel that it provides a quicker understanding of its nature than a scatter plot or averaged bar graph. This data is also summarized in numerical form in Table 2.

The average multithreaded speedup (Table 2) for SPEC is 1.20, and ranges from 0.90 to 1.58, with a sample standard deviation of 0.11.

From the data, we can see several things. First of all, we can pick out particular tests that seem to have more or less dynamic conflicts with other threads. For instance, *sixtrack* and *eon* seem to have fairly little impact on most other tests, achieving high combined speedups regardless of which threads they are paired with. In fact, the high outliers in Figure 2 are all caused by runs of benchmarks with *sixtrack*. However, *swim* and *art* conflict severely with most other threads. What isn't clear from these graphs (because we report combined speedup) is whether the low speedups for *swim* and *art* are due to them running slowly when competing for dynamically allocated resources, or whether they

Benchmark	Best Speedup	Worst Speedup	Avg Speedup
gzip	1.48	1.14	1.24
vpr	1.43	1.04	1.17
gcc	1.44	1.00	1.11
mcf	1.57	1.01	1.21
crafty	1.40	0.99	1.17
parser	1.44	1.09	1.18
eon	1.42	1.07	1.25
perlbmk	1.40	1.07	1.20
gap	1.43	1.17	1.25
vortex	1.41	1.01	1.13
bzip2	1.47	1.15	1.24
twolf	1.48	1.02	1.16
wupwise	1.33	1.12	1.24
swim	1.58	0.90	1.13
mgrid	1.28	0.94	1.10
applu	1.37	1.02	1.16
mesa	1.39	1.11	1.22
galgel	1.47	1.05	1.25
art	1.55	0.90	1.13
equake	1.48	1.02	1.21
facerec	1.39	1.16	1.25
ammp	1.40	1.09	1.21
lucas	1.36	0.97	1.13
fma3d	1.34	1.13	1.20
sixtrack	1.58	1.28	1.42
apsi	1.40	1.14	1.23
Overall	1.58	0.90	1.20

Table 2. Multiprogrammed SPEC Speedups

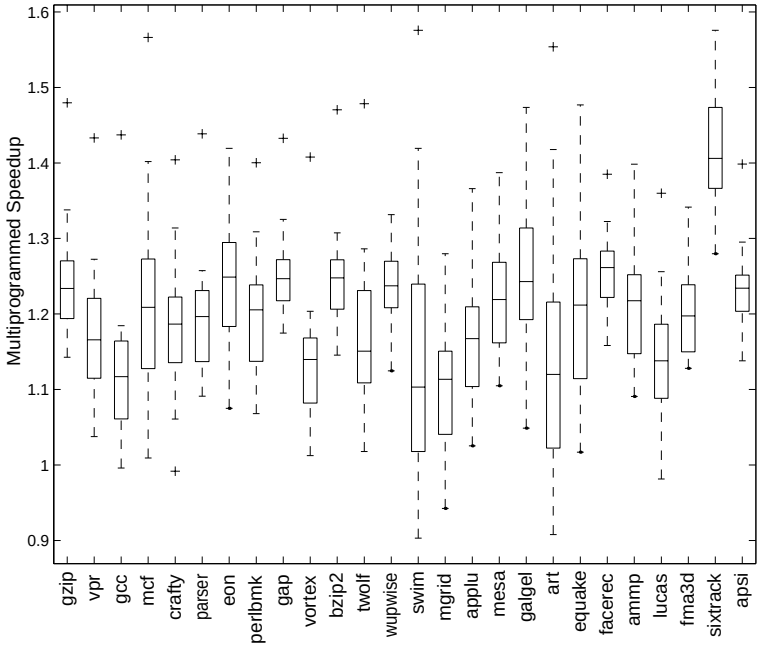

Figure 2. Multiprogrammed speedup of all SPEC benchmarks

tend to cause their partner to run slowly. In fact, it is the latter. We see that *swim* on average sees 63% of its original throughput, while the coscheduled thread achieves only 49% of it's original throughput. Likewise, *art* on average achieves 71% of its original throughput, whereas coscheduled threads achieve only 42%.

The worst speedup is achieved by *swim* running with *art*. The best speedup is achieved by *swim* running with *sixtrack* (although *art* also has the third highest speedup in conjunction with *sixtrack*). Thus, *swim* is part of both the best and worst pairs! Analysis of these cases with VTune reveal that *swim* and *art* both have low IPCs due to relatively high cache miss rate. When run with *swim*, *art*'s poor cache behavior increases the L2 miss rate of *swim* by a factor of almost 40. In the other case, *swim* and *art*'s low IPCs interfere only minimally with *sixtrack*.

The lowest speedup is below 1.0. However, the slowdowns are uncommon, and no lower than a 0.9 speedup. In all, only 8 of the 351 combinations experience slowdowns. None of the slowdowns approach the worst case results predicted by Tullsen and Brown [27]. That paper identifies scenarios, particularly with two threads, where a shared instruction queue could result in speedups significantly below one for certain combinations of threads if the queue is not designed carefully. That work also shows that a partitioned queue, while limiting performance in the best case, mitigates the problem. These results for the Pentium 4 partitioned queue seem to confirm that result. This architecture usually achieves its stated goal [13] of providing high isolation between threads.

5.3 Parallel Multithreaded Speedup

Parallel applications exercise aspects of the multithreaded implementation that the multiprogrammed workload does not. Parallel speedup will be sensitive to the speed and efficiency of synchronization and communication mechanisms. In addition, parallel applications have the potential to compete more heavily for shared resources than a multiprogrammed workload – if the threads of the parallel program each have similar characteristics, they will likely each put pressure on the same resources in the same way, and a single resource will become a bottleneck more quickly [11].

Parallel speedups for the SPLASH2 benchmarks are shown in Figure 3. For the sake of comparison, that graph also shows the speedup when two single-threaded copies of that benchmark are run. The multithreaded speedup ranges from 1.02 to 1.67, and thus is a positive speedup in all cases, although the speedups for *ocean contiguous* and *water spatial* are not significant. The multiprogrammed speedups range from 1.06 to 1.49, with no slowdowns. It is interesting to note that in three out of the eight cases, the speedup from the parallel version is actually greater than the average speedup from running two copies.

Figure 4 shows the results of running the NAS parallel benchmarks compiled with OpenMP directives for the class

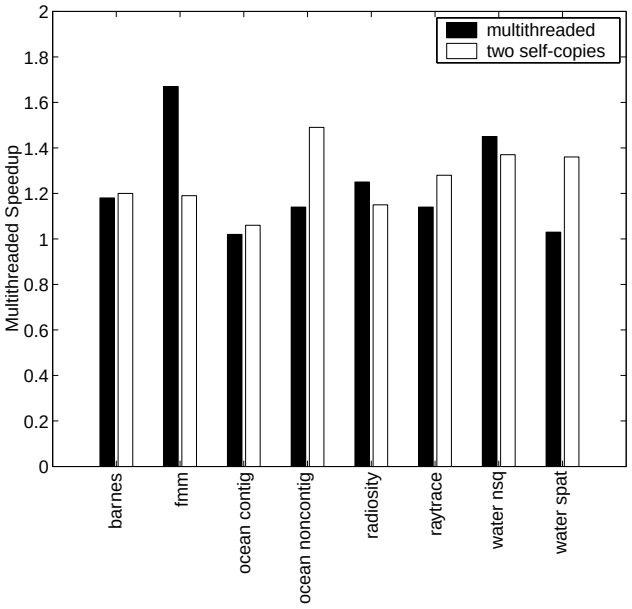

Figure 3. SPLASH2 Multithreaded and Dual-Copy Speedups

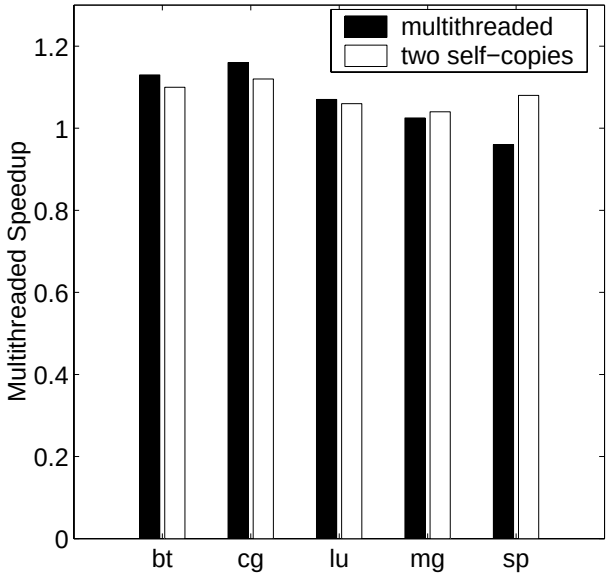

Figure 4. NAS Multithreaded and Dual-Copy Speedups

A problem size (using the Intel compiler). The parallel speedups for these benchmarks were more modest, ranging from 0.96 to 1.16. Multiprogrammed speedups were higher in three out of five cases than the corresponding multithreaded speedups.

5.4 Synchronization and Communication Speed

While the parallel applications indirectly measure synchronization and communication speed, this section measures those quantities more directly. To measure communication speed, we construct a loop that does nothing but alternate between threads reading a value that is protected by a lock. We can pass the value between threads 37 million times per second, implying that this lock and read takes approximately 68 cycles.

A second communication component test creates two threads that update (increment) a value repeatedly, again protected by synchronization. On this benchmark, the combined threads can update the value 14.6 million times per second, or an update every 171 cycles. This is significantly more than the latency of either of the caches, but less than main memory latency.

A third test of communication is taken from Tullsen, et al. [30], in which a loop is parallelized in interleaved fashion among the threads. This loop contains a loop-carried dependence, as well as a variable amount of independent work. The independent work is a sequence of N serial floating point computations. The metric of communication and synchronization speed is the value of N at which the parallelism of the floating point computation allows the parallel version of the benchmark to outperform the single-thread version. In that paper, it is shown that an SMT processor with the proposed synchronization mechanism (not the same as that used for the Pentium 4) can effectively parallelize a loop with more than an order of magnitude less parallelism than necessary for a traditional parallel machine. A similar graph is shown in Figure 5, with different synchronization mechanisms. We coded a test-and-test-and-set spin lock with a *pause* instruction inside the loop. Using this locking mechanism to synchronize between two threads running the same code as [30], we found a break-even point of approximately 20000 independent computations for a hyper-threaded processor and 200000 independent computations for a traditional SMP machine. Though the SMT processor breaks even with orders of magnitude less parallel computation than the multiprocessor (as research would lead us to expect), the actual break-even points are far different (much higher) than those shown in [30]. We also tried this experiment with standard pthreads based mutex locks and unlocks, but found that the performance was much lower that our custom coded locking routine.

5.5 Heterogeneous vs. Homogeneous Workloads

Several papers have demonstrated that an SMT processor benefits from diversity. Speedups are not as high with traditional loop-level parallelism, because all threads put pressure on the same resources [11, 12]. Even in a multiprogrammed workload, threads with similar resource needs tend to achieve lower combined speedups [22].

In running the cross-product of all SPEC benchmarks, we also run each benchmark against a version of itself (with

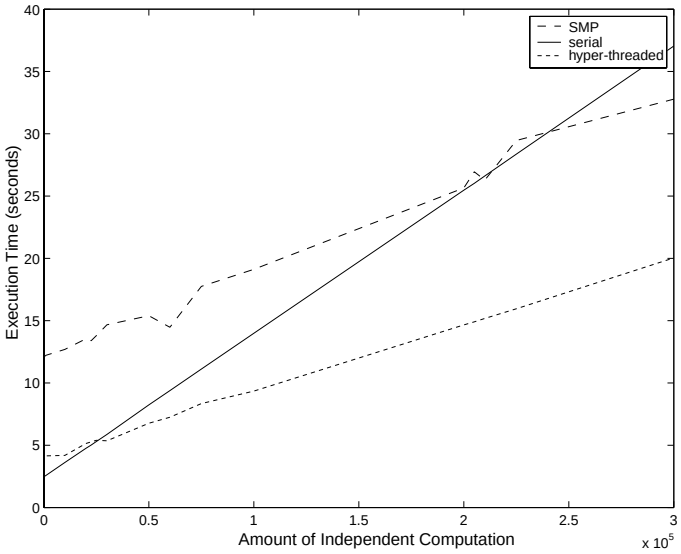

Figure 5. Parallel Execution Profitability

the exact same inputs). In that configuration, the threads do benefit from some constructive interference, in the branch predictor and the page cache, but we still expect the low-level conflicts to dominate those effects. In fact, the average speedup of that case was 1.11, lower than the average speedup for all combinations of 1.20.

This phenomenon even persists for a more general definition of "similar" threads. We found that the average speedup when integer benchmarks are run with other integer benchmarks was 1.17, the average speedup for two floating point threads was 1.20, while the average speedup when an integer and floating point thread are run together was 1.21. This is a slight advantage for heterogeneity, but not nearly as much as has been suggested by prior research. It is likely that static resource partitioning ameliorates the effect.

5.6 Symbiosis

The term symbiosis was used to describe the range of behaviors different groups of threads experience, some threads cooperating well and others not [22, 23]. This phenomenon was exploited to demonstrate speedups of 17% with a symbiosis-aware OS scheduler, by creating a schedule that pairs together threads that have high symbiosis when coscheduled, in the case where more threads than hardware contexts are time-sharing the available contexts.

We achieved speedups on SPEC ranging from 0.90 to 1.58, but it is not enough for speedups to vary for symbiotic job scheduling to work – speedups must vary with characteristics of the pair of scheduled jobs rather than the characteristics of the individual threads. The expected speedup from a random schedule of the 26 SPEC threads is just 1.20, the average speedup of all pairs. However, we can manually create a "best" schedule, using the assumptions of [22], where each job will be coscheduled with two others as the system moves through a fair circular schedule, each timeslice swapping out one of the two jobs. The expected speedup of our best schedule, using the measurements we already have, is 1.27, or about a 6% improvement. Thus there is still some potential for symbiotic scheduling on this architecture, even though there is less dynamic sharing of low-level resources than the research SMT processor.

Despite this validation that the OS can benefit from making a coordinated decision about job schedules, the Linux kernel is currently not well positioned to exploit symbiosis. On the hyper-threading CPU it employs two instantiations of the kernel scheduler, each hardware context making its own scheduling decisions without coordination with the other. In this configuration, exploiting symbiosis requires communication between the two schedulers. However, if it used a single version of the scheduler that made a single decision about which two jobs to run each time slice, it would be able to exploit symbiosis more naturally.

5.7 Compiler Interaction

Expected SMT performance has been shown to vary with the optimizations performed by the compiler, and the effectiveness of particular compiler optimizations are affected by the level of multithreading [16, 12]. The general trend, however, is that multithreading makes the processor somewhat tolerant of the quality of the compiler – the expected speedup from simultaneous multithreading is highest when the compiler is least effective at exploiting ILP. In this section we examine this thesis as follows. We selected a subset of the SPEC benchmarks (*gcc, mcf, crafty, eon, gap, swim, mgrid, art, lucas, and sixtrack*), to use as a representative subset of SPEC. Our subset has a slightly lower overall speedup than our original subset (1.18 vs 1.20) but it includes all the cases of highest and lowest speedups from our original exhaustive results. We then ran the crossproduct of these benchmarks compiled with the following optimization levels: **-O3**, **-O1 -xW -O1**, and **-O0** and compare the results against the same subset of our baseline run, which was compiled with **-O3 -xW -ipo**. The meaning of these compiler switches is as follows:

- **-O0** Disable all compiler optimizations
- **-O1** Enable global register allocation, instruction scheduling, register variable detection, common subexpression elimination, dead-code elimination, variable renaming, copy propagation, constant propagation, strength reduction-induction, tail recursion elimination and software pipelining.

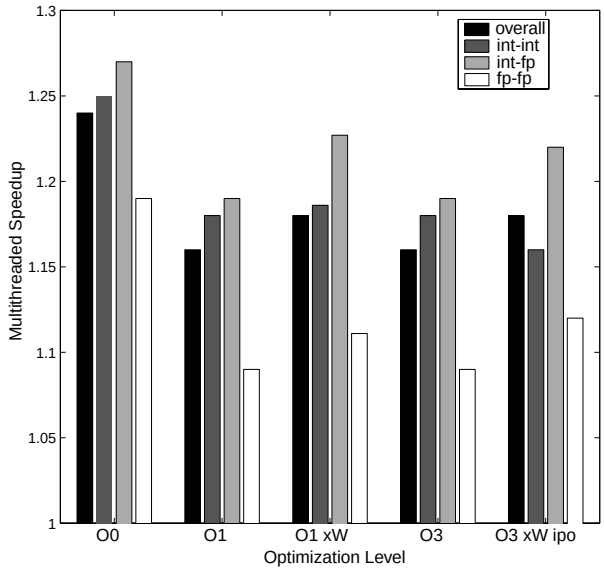

Figure 6. Average multithreaded speedup for given optimization levels, relative to single-thread performance for the same optimization level.

- **-O3** Performs the same optimizations as **-O1** and adds prefetching, scalar replacement, and loop tranformations such as unrolling.
- **-xW** Enables generation of all instructions that are available on the Pentium 4 that are not core x86 instructions. These include MMX, SSE and SSE2.
- **-ipo** Enables interprocedural optimizations within and across multiple files.

In general, we find (Figure 6) that decreasing amounts of optimization improves multithreaded speedup slightly relative to the degraded baseline. However, this is not enough to make up for the performance lost. For example, **-O0** has an average multithreaded speedup of 1.24, or 3.3% better than our fully optimized multithreaded speedup, however the average speedup relative to the fully optimized version of the code is only 0.67. As might be expected, floating point programs suffer more in absolute terms at reduced optimization levels, but their normalized multithreaded speedup is only slightly lower than that of the integer programs. These results are in line with previous research results.

5.8 Simulation Validation

Prior SMT research demonstrated a potential speedup of about 2.5 on a dynamically scheduled superscalar processor [28]. However, that speedup was predicted for a very different implementation than the Pentium 4. That processor had eight hardware contexts and was an 8-issue processor with a shorter pipeline. We reconfigured the SMTSIM simulator [26] to model a configuation more like the Pentium 4 hyper-threading implementation. Then we ran it on the same reduced subset of SPEC benchmarks that were used in the previous section.

Changes to the simulator include modifications to the cache subsystem, accurate modeling of cache and memory latencies, changes to fetch and commit policies, etc. The SMTSIM simulator, however, is still emulating the Alpha ISA, so validation is very approximate, at best.

Our results showed ranges in performance that were similar to those we show here, and if anything predict lower overall performance gains. Those results indicate that while the measured overall speedup for this processor is low relative to previously published expectations, that difference is primarily a function of the issue width of the processor and the number of threads. When those things are taken into account, the Hyper-Threading Pentium 4 is indeed achieving a level of performance that meets or even exceeds expected results from the research.

6 Conclusions

This paper presents performance results for a simultaneous multithreaded architecture, the hyper-threading Pentium 4 processor. This architecture achieves an average multithreaded speedup of 1.20 on a multiprogrammed workload and 1.24 on a parallel workload. This processor delivers on the promise of the published SMT research, in light of the limitations of a dual-thread 3-wide superscalar processor.

We find that although the processor has been designed to minimize conflicts between threads, it still shows symbiotic behavior due to cache and other resource conflicts. The processor delivers an order of magnitude faster synchronization and communication than traditional SMP systems, however this implementation of the architecture seems to fall short of the potential to exploit the very tight coupling of an SMT processor to provide very fast synchronization and enable new opportunities for parallelism.

Acknowledgments

The authors would like to thank Doug Carmean and Eric Sprangle, as well as the anonymous reviewers, for helpful comments on this research. We would also like to thank Doug Carmean and Intel for early access to the equipment used in this research. This work was funded in part by NSF Grant CCR-0105743 and a gift from Intel Corporation.

References

[1] Using spin-loops on intel pentium 4 processor and intel xeon processor. Application note, Intel Corporation, May 2001.

[2] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, June 1993.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[4] J. Emer. Simultaneous multithreading: Multiplying alpha's performance. In *Microprocessor Forum*, Oct. 1999.

[5] M. J. Flynn, A. Podvin, and K. Shimizu. *Parallel Processing Systems, Technologies, and Applications*, chapter A multiple instruction stream processor with shared resources, pages 251–286. Spartan Books, 1970.

[6] J. G.E. Daddis and H. Torng. The concurrent execution of multiple instruction streams on superscalar processors. In *International Conference on Parallel Processing*, pages I:76–83, Aug. 1991.

[7] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium4 processor. *Intel Technology Journal*, 2001.

[8] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *19th Annual International Symposium on Computer Architecture*, pages 136–145, May 1992.

[9] S. Keckler and W. Dally. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *19th Annual International Symposium on Computer Architecture*, pages 202–213, May 1992.

[10] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, Oct. 1994.

[11] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen. Converting thread-level parallelism into instruction level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, Aug. 1997.

[12] J. Lo, S. Eggers, H. Levy, S. Parekh, and D. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *30th International Symposium on Microarchitecture*, Dec. 1997.

[13] D. Marr, F. Binns, D. Hill, G. Hinton, K. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture: a hypertext history. *Intel Technical Journal*, 1(1), Feb. 2002.

[14] J. D. McCalpin. Sustainable memory bandwidth in current high performance computers. Technical report, Silicon Graphics, Oct. 1995.

[15] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

[16] N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen. Ilp versus tlp on smt. In *Supercomputing '99*, Nov. 1999.

[17] R. Prasadh and C.-L. Wu. A benchmark evaluation of a multi-threaded RISC processor architecture. In *International Conference on Parallel Processing*, pages I:84–91, Aug. 1991.

[18] J. Redstone, S. Eggers, and H. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[19] L. E. Shar and E. S. Davidson. A multiminiprocessor system implemented through pipelining. *IEEE Computer*, 7(2), Feb. 1974.

[20] U. Sigmund and T. Ungerer. *Lecture Notes in Computer Science 1123:797-800*, chapter Identifying Bottlenecks in Multithreaded Superscalar Multiprocessors. springer-verlag, 1996.

[21] B. Smith. Architecture and implications of the HEP multiprocessor computer system. In *SPIE Real Time Signal Processing IV*, pages 241–248, 1981.

[22] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading architecture. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[23] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *2002 International Conference on Measurement and Modeling of Computer Systems (Sigmetrics 02)*, June 2002.

[24] J. Thornton. Parallel operations in control data 6600. In *Proceedings of the AFIPS Fall Joint Computer Conference*, 1964.

[25] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse. The MAJC architecture: A synthesis of parallelism and scalability. *IEEE Micro*, 20(6), Nov. 2000.

[26] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[27] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *34th International Symposium on Microarchitecture*, Dec. 2001.

[28] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, May 1996.

[29] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[30] D. Tullsen, J. Lo, S. Eggers, and H. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, Jan. 1999.

[31] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Conference on Parallel Architectures and Compilation Techniques*, pages 49–58, June 1995.