Multiprogramming Performance of the Pentium 4 with Hyper-Threading

James R. Bulpin*and Ian A. Pratt

University of Cambridge Computer Laboratory
J J Thomson Avenue, Cambridge, UK, CB3 0FD.
Tel: +44 1223 331859.
james.bulpin@cl.cam.ac.uk

Abstract

Simultaneous multithreading (SMT) is a very fine grained form of hardware multithreading that allows simultaneous execution of more than one thread without the notion of an internal context switch. The fine grained sharing of processor resources means that threads can impact each others' performance.

Tuck and Tullsen first published measurements of the performance of the SMT Pentium 4 processor with Hyper-Threading [12]. Of particular interest is their evaluation of the multiprogrammed performance of the processor by concurrently running pairs of single-threaded benchmarks. In this paper we present experiments and results obtained independently that confirm their observations. We extend the measurements to consider the mutual fairness of simultaneously executing threads (an area hinted at but not covered in detail by Tuck and Tullsen) and compare the multiprogramming performance of pairs of benchmarks running on the Hyper-Threaded SMT system and on a comparable SMP system.

We show that there can be considerable bias in the performance of simultaneously executing pairs and investigate the reasons for this. We show that the performance gap between SMP and Hyper-Threaded SMT for multiprogrammed workloads is often lower than might be expected, an interesting result given the obvious economic and energy consumption advantages of the latter.

*James Bulpin is funded by a CASE award from Marconi Corporation plc. and EPSRC

1 Introduction

Intel Corporation's "Hyper-Threading" technology [6] introduced into the Pentium 4 [3] line of processors is the first commercial implementation of simultaneous multithreading (SMT). SMT is a form of hardware multithreading building on dynamic issue superscalar processor cores [15, 14, 1, 5]. The main advantage of SMT is its ability to better utilise processor resources and to hide memory hierarchy latency by being able to provide more independent work to keep the processor busy. Other architectures for simultaneous multithreading and hardware multithreading in general are described elsewhere [16].

Hyper-Threading currently supports two heavy weight threads (processes) per processor, presenting the abstraction of two independent logical processors. The physical processor contains a mixture of duplicated (per-thread) resources such as the instruction queue; shared resources tagged by thread number such as the DTLB and trace cache; and dynamically shared resources such as the execution units. The resource partitioning is summarised in table 1. The scheduling of instructions to execution units is process independent although there are limits on how many instructions each process can have queued to try to maintain fairness.

Whilst the logical processors are functionally independent, contention for resources will affect the progress of the processes. Compute-bound processes will suffer contention for execution units while processes making more use of memory will contend for use of the cache with the possible result of increased capacity and conflict misses. With cooperating processes the sharing of the cache may be useful but for two arbitrary processes the contention may have a

	Duplicated	Shared	Tagged/Partitioned
Fetch	ITLB Streaming buffers	Microcode ROM	Trace cache
Branch prediction	Return stack buffer Branch history buffer		Global history array
Decode	State	Logic	uOp queue (partitioned)
Execute	Register rename	Instruction schedulers	Retirement Reorder buffer (up to 50% use per thread)
Memory		Caches	DTLB

Table 1: Resource division on Hyper-Threaded P4 processors.

negative effect.

In general, multi-threaded processors are best exploited by running cooperating threads such as a true multi-threaded program. In reality however many workloads will be single threaded. With Intel now incorporating Hyper-Threading into the Pentium 4 line of processors aimed at desktop PCs it is likely that many common workloads will be single-threaded or at least have a dominant main thread.

In this paper we present measurements of the performance of a real SMT system. This is preferable to simulation as using an actual system is the only way to guarantee that all contributory factors are captured in the measurements. Of particular interest was the effect of the operating system and the memory hierarchy, features sometimes simplified or ignored in simulation studies. Furthermore most simulation studies are based on SMTSIM [13] which differs from Intel's Hyper-Threading in a number of major ways including the number of threads available, the degree of dynamic sharing and the instruction set architecture.

A study of this nature is useful as an aide to understanding the benefits and limitations of Hyper-Threading. This work is part of a larger study of practical operating system support for Hyper-Threading. The observations from the experiments described here are helping to drive the design of a Hyper-Threading-aware process scheduler.

It is of interest to compare the performance of pairs of processes executing with different degrees of resource sharing. The scenarios are:

- Shared-memory symmetric multiprocessing (SMP) where the memory and its bus are the main shared resources.
- SMT with its fine grain sharing of all resources.
- Round-robin context switching with the non-simultaneous sharing of the caches.

The obvious result is that in general the per-thread and aggregate performance will be the highest on the SMP system. However at a practical level, particularly for the mass desktop market, one must consider the economic advantages of a single physical processor SMT system. It is therefore useful to know how much better the SMP performance is.

2 Related Work

Much of the early simultaneous multithreading (SMT) work studied the performance of various benchmarks in order to demonstrate the effectiveness of the architecture [15, 14, 5]. Necessarily these studies were simulation based and considered the application code rather than the entire system including the operating system effects. Whilst useful, the results from these studies do not directly apply to current hardware as the microarchitecture and implementation can drastically change the behaviour.

Snavely *et al.* use the term "symbiosis" to describe how concurrently running threads can improve the throughput of each other [8]. They demonstrated the effect on the Tera MTA and a simulated SMT processor.

Redstone *et al.* investigated the performance of workloads running on a simulated SMT system with a full operating system [7]. They concluded that the time spent executing in the kernel can have a large impact on the speedup measurements compared to a user-mode only study. They report that the inclusion of OS effects on a SPECInt95 study has less impact on SMT performance measurements that it does on non-SMT superscalar results due to the better latency hiding of SMT being able to mask the poorer

IPC of the kernel parts of the execution. This result is important as it means that a comparison of SMT to superscalar without taking the OS into account would not do the SMT architecture justice.

In a study of database performance on SMT processors, Lo *et al.* noted that the large working set of this type of workload can reduce the benefit of using SMT unless page placement policy is used to keep the important "critical" working set in the cache [4].

Grunwald and Ghiasi used synthetic workloads running on a Hyper-Threaded Pentium 4 to show that a malicious thread can cause a huge performance impact to a concurrently running thread through careful targeting of shared resources [2]. Our work has a few features in common with that of Grunwald and Ghiasi but we are more interested in the mutual effects of non-malicious applications that may not have been designed or compiled with Hyper-Threading in mind. Some interesting results from this study were the large impact of a trace-cache flush caused by self-modifying code, and of a pipeline flush caused by floating-point underflow.

Vianney assessed the performance of Linux under Hyper-Threading using a number of microbenchmarks and compared the performance of some multithreaded workloads running on a single processor with and without Hyper-Threading enabled [17]. The result was that most microbenchmarks had the same performance with Hyper-Threading both enabled and disabled and that the multithreaded workloads exhibited speedups of 20 to 50% with Hyper-Threading enabled depending on the workload and kernel version.

More recently Tuck and Tullsen [12] have made measurements of thread interactions on the Intel Pentium 4 with Hyper-Threading; these measurements parallel our own upon which this work is based. All of the studies show that the range of behaviour is wide.

3 Experimental Method

The experiments were conducted on an Intel Pentium 4 Xeon based system running the Linux 2.4.19 kernel. This version of the kernel contains support for Hyper-Threading at a low level, including the detection of the logical processors and the avoidance of timing-loops. The kernel was modified with a variation of the `cpus_allowed` patch[1]. This patch provides an interface to the Linux `cpus_allowed` task attribute and allows the specification of which processor(s) a process can be executed on. This is particularly important as the scheduler in Linux 2.4.19 is not aware of Hyper-Threading. Use of this patch prevented threads being migrated to another processor (logical or physical). A `/proc` file was added to allow lightweight access to the processor performance counters.

Details of the experimental machine are given in table 2. The machine contained two physical processors each having two logical (Hyper-Threaded) processors. Also included are details given by Tuck and Tullsen for their experimental machine [12] to which Intel gave them early access which would explain the non-standard clock speed and L2 cache combination.

For each pair of processes studied the following procedure was used. Both processes were given a staggered start and each run continuously in a loop. The timings of runs were ignored until both processes had completed at least one run. The experiment continued until both processes had accumulated 3 timed runs. Note that the process with the shorter runtime will have completed more than 3 runs. This method guaranteed that there were always two active processes and allowed the caches, including the operating system buffer cache, to be warmed. Note that successive runs of the one process would start at different points within the other process' execution due to the differing run times for both.

The complete cross-product of benchmarks was run on Hyper-Threading, SMP and single-processor context switching configurations. The Hyper-Threading experiments were conducted using the two logical processors on the second physical processor and the SMP experiments used the first logical processor on each physical processor with the other processor idle (equivalent to disabling Hyper-Threading). The context switching experiments were run on the second physical processor and used the round-robin feature of the Linux scheduler with a modification to allow the quantum to be specified. In all cases the machine was configured to minimise background system activity.

A set of base run times and performance counter values were measured by running benchmarks alone on a single physical processor. A dummy run of each benchmark was completed before the timed runs to

[1]The cpus_allowed/launch_policy patch was posted to the linux-kernel mailing list by Matthew Dobson in December 2001

	Our machine	Tuck and Tullsen
Model	Intel SE7501 based	
CPU	2 x P4 Xeon 2.4GHz HT	1 x P4 2.5GHz HT
L1 cache	8kB 4 way D, 12k-uops trace I	
L2 cache	8 way 512kB	8 way 256kB
Memory	1GB DDR DRAM	512MB DRDRAM
OS	RedHat 7.3	RedHat 7.3
Kernel	Linux 2.4.19	Linux 2.4.18smp

Table 2: Experimental machine details.

warm the caches. A total of 9 timed runs were made and the median run time was recorded. This procedure was performed twice; once using a single logical processor with the second logical processor idle (but still with Hyper-Threading enabled), and once with Hyper-Threading disabled. The run times for both configurations were almost identical. This behaviour is expected because the processor recombines partitioned resources when one of the logical processors is idle through using the `HALT` instruction [6].

The pairs of processes came from the SPEC CPU2000 benchmark suite [11]. The runs were complete and used the reference data sets. The executables were compiled with GCC 2.96 using a fairly benign set of optimisation flags. The Fortran-90 benchmarks, *178.galgel*, *187.facerec*, *189.lucas* and *191.fma3d* were not used due to GCC not supporting this language.

In order to ascertain the effect of the compiler on the process' interaction a subset of experiments was run using GCC 3.3 with a more aggressive set of optimisation flags. While the newer compiler produced executables with reduced run times, we observed no significant difference in speedup. We hope to further explore this area in the future using the Intel C Compiler.

4 Results

For the purposes of the analysis, one process was considered to be the *subject* process and the other the *background*. The experiments were symmetric therefore only one experiment was required for each pair but the data from each experiment was analysed twice with the two processes taking the roles of subject and background in turn (except where a benchmark competed against a copy of itself).

The performance of an individual benchmark running in a simultaneously executing pair is described by its execution time when running alone divided by its execution time when running in the pair. If a non-SMT processor is being timeshared in a theoretic perfect (no context switch penalty) round-robin fashion with no cache pollution then a performance of 0.5 would be expected as the benchmark is getting half of the CPU time. A perfect SMP system with each processor running one of the pair of benchmarks with no performance interactions would give a performance of 1 for each benchmark. It would be expected that benchmarks running under SMT would fall somewhere between 0.5 and 1, anything less than 0.5 being a unfortunate loss.

The total *system speedup* for the pair of benchmarks is the sum of the two performance values. This speedup is compared to zero-cost context switching with a single processor so a perfect SMP system should have a system speedup of 2 while a single Intel Hyper-Threaded processor should come in somewhere between 1 and 2. Intel suggest that Hyper-Threading provides a 30% speedup which would correspond to a system speedup of 1.3 in our analysis.

In the following sections we present a summary of results from the Hyper-Threading and SMP experiments. The single-processor context switching experiments using a quantum of 10ms generally resulted in a performance of no worse than 0.48 for each thread, a 4% drop from the theoretic zero-cost case. As well as the explicit cost of performing the context switch the cache pollution contributes to the slowdown. The relatively long quantum means that the processes have time to build up and benefit from cached information. We do not present detailed results from these experiments.

4.1 Hyper-Threading

In figure 1 we show our results for benchmark pairs on the Hyper-Threaded Pentium 4 using the same format as Tuck and Tullsen [12] to allow a direct

comparison[2]. For each subject benchmark a box and whisker plot shows the range of system speedups obtained when running the benchmark simultaneously with each other benchmark. The box shows the interquartile range (IQR) of these speedups with the median speedup shown by a line within the box. The whiskers extend to the most extreme speedup within 1.5 IQR of the 25th and 75th percentile (i.e. the edges of the box) respectively. Individual speedups outside of this range are shown as crosses. The gaps on the horizontal axis are where the Fortan-90 benchmarks would fit.

Our experimental conditions differ from Tuck and Tullsen's in a few ways, mainly the size of the L2 cache (our 512kB vs. 256kB), the speed of the memory (our 266MHz DDR vs. 800MHz RAMBUS) and the compiler (our GCC 2.96 vs. the Intel Reference Compiler). The similarities in the results given these differences show that the effect of the processor microarchitecture is important and that the lessons that can be learned can be applied to more than just the particular configuration under test.

For the integer benchmarks our results match those of Tuck and Tullsen almost exactly. However we do see a slightly larger IQR with many of the integer benchmarks which is one reason we see fewer outliers than Tuck and Tullsen. Of the floating point results we match closely on *wupwise*, *mgrid*, *applu*, *art* and *equake*, and fairly closely on *swim* and *apsi*. We show notable differences on *mesa*, our experiments having greater speedups, and *sixtrack*, our experiments having smaller speedups. The *sixtrack* difference is believed to be due to the different L2 cache sizes; this is further described in the discussion below. We did not use the Fortran-90 benchmarks.

We measure an average system speedup across all the benchmarks of 1.20, the same figure as reported by Tuck and Tullsen. We measure slightly less desirable best and worst case speedups of 1.50 (*mcf* vs. *mesa*) and 0.86 (*swim* vs. *mgrid*) compared to Tuck and Tullsen's 1.58 (*swim* vs. *sixtrack*) and 0.90 (*swim* vs. *art*).

Figure 2 shows the individual performance of each benchmark in a multiprogrammed pair. The figure is organised such that a square describes the performance of the row benchmark when sharing the processor with the column benchmark. The performance is considered bad when it is less than 0.5, i.e. worse than perfect context switching, and good when above 0.5. The colour of the square ranges from white for bad to black for good with a range of shades inbetween. The first point to note is the lack of reflective symmetry about the top-left to bottom-right diagonal. In other words, when two benchmarks are simultaneously executing, the performance of each individual benchmark (compared to it running alone) is different. This shows that the performance of pairs of simultaneously executing SPEC2000 benchmarks is not fairly shared. Inspection of the rows shows that benchmarks such as *mesa* and *apsi* always seem to do well regardless of what they simultaneously execute with. Benchmarks such as *mgrid* and *vortex* suffer when running against almost anything else. Looking at the columns suggests that benchmarks such as *sixtrack* and *mesa* rarely harm the benchmark they share the processor with while *swim*, *art* and *mcf* usually hurt the performance of the other benchmark.

The results show that a benchmark executing with another copy of itself (using a staggered start) usually has a lower than average performance demonstrating the processor's preference for heterogeneous workloads which is not overcome by benefits in shared text segments.

The performance counter values recorded from the base runs of each benchmarks allow an insight into the observed behaviour:

mcf has a notably low IPC which can be attributed, at least in part, to its high L2 and L1-D miss rates. An explanation for why this benchmark rarely suffers when simultaneously executing with other benchmarks is that it is already performing so poorly that it is difficult to do much further damage (except with *art* and *swim* which have very high L2 miss rates). It might be expected that a benchmark simultaneously executing with *mcf* would itself perform well so long as it made relatively few cache accesses. *eon* and *mesa* fall into this category and the latter does perform well (28% speedup compared to sequential execution) but the former has only a moderate performance (12% speedup) probably due its very high trace cache miss rate causing many accesses to the (already busy) L2 cache.

gzip is one of the benchmarks that generally does not detriment the performance of other benchmarks. It makes a large number of cache accesses and has a moderately high L1 D-cache miss rate of approximately 10%. It does however have a small L2 cache and D-TLB miss rate due to its small memory footprint.

vortex suffers a reduced performance when running with most other benchmarks. There is nothing of par-

[2]the figure is physically sized to match Tuck and Tullsen's

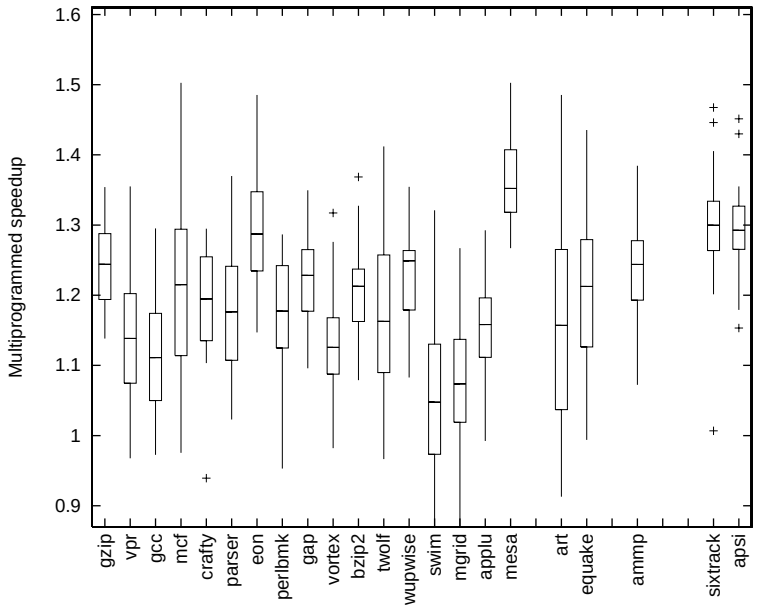

Figure 1: Multiprogrammed speedup of pairs of SPEC CPU2000 benchmarks running on a Hyper-Threaded processor.

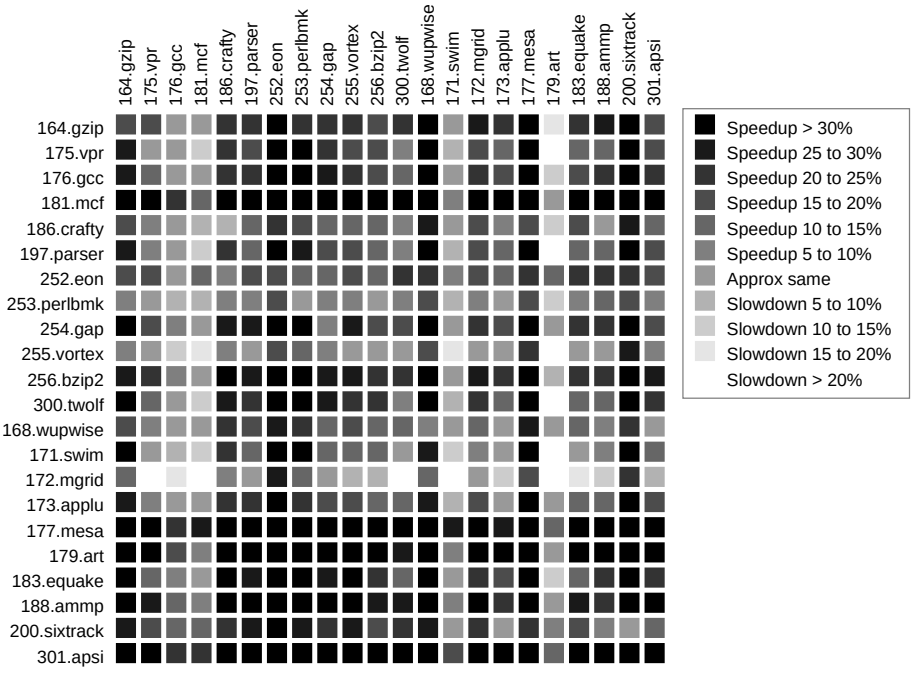

Figure 2: Effect on each SPEC CPU2000 benchmark in a multiprogrammed pair running on a Hyper-Threaded processor. A black square represents a good performance for the subject benchmark and a white square denotes a bad performance.

ticular note in it performance counter metrics other than a moderately high number of I-TLB misses and a reasonable number of trace cache misses (although both figures are well below the highest of each metric).

mcf, *swim* and *art* have high L1-D and L2 miss rates when running alone and have a low average IPC. They tend to cause a detriment to the performance of other benchmarks when simultaneously executing. *art* and *mcf* generally only suffer a performance loss themselves when the other benchmark also has a high L2 miss rate, *swim* suffers most when sharing with these benchmarks but is also more vulnerable to those with moderate miss rates.

mgrid is the benchmark that suffers the most when running under SMT whilst the simultaneously executing benchmark generally takes only a small performance hit. *mgrid* is notable in that it executes more loads per unit time than any other SPEC CPU2000 benchmark and has the highest L1 D-cache miss rate (per unit time). It has only a moderately high L2 miss rate and a low D-TLB miss rate. The only benchmarks that do not cause a performance loss to *mgrid* are those with low L2 miss rates (per unit time). *mgrid*'s baseline performance is good (an IPC of 1.44) given its high L1-D miss rate. The benchmark relies on a good L2 hit rate which makes it vulnerable to any simultaneously executing thread that pollutes the L2 cache.

sixtrack has a high baseline IPC (with a large part of that being floating point operations) and a low L1-D miss rate but a fairly high rate of issue of loads. The only benchmark it causes any significant performance degradation to is another copy of itself; this is most likely due to competition for floating point execution units. It suffers a moderate performance degradation when simultaneously running with benchmarks with moderate to high cache miss rates such as *art* and *swim*. The competitor benchmark will increase contention in the caches and harm *sixtrack*'s good cache hit rate. Tuck and Tullsen report that *sixtrack* suffers only minimal interference from *swim* and *art*. We believe the reason for this difference is that our larger L2 cache gives *sixtrack* a better baseline performance which makes it more vulnerable to performance degradation from benchmarks with high cache miss rates giving it lower relative speedups.

The best pairing observed in terms of system throughput was *mcf* vs. *mesa* (50% system speedup). Although *mcf* gets the better share of the performance gains, *mesa* does fairly well too. The performance counter metrics shown qualitatively in table 3 for this pair show that heterogeneity is good.

Tuck and Tullsen note that *swim* appears in both the best and worse pairs. The reason for this is mainly down to the competitor. *mgrid* with its high L1-D miss rate is bad; *sixtrack* and *mesa* are good as they only have low L1-D miss rates so do little harm to the other thread.

4.2 Hyper-Threading vs. SMP

Figure 3 shows the speedups for the benchmark pairs running in a traditional SMP configuration. Also shown for comparison is the Hyper-Threading data as shown above. An interesting observation is that benchmarks that have a large variation in performance under Hyper-Threading also have a large variation under SMP. It might be imagined that the performance of a given benchmark would be more stable under SMP than under Hyper-Threading since there is much less interaction between the two processes. The correspondence in variation suggest that competition for off-chip resources such as the memory bus are as important as on-chip interaction.

Unlike Hyper-Threading, SMP does not show any notable unfairness between the concurrently executing threads. This is clearly due to the vast reduction in resource sharing with the main remaining resource being the memory and its bus and controller. This means that the benchmarks that reduce the performance of the other running benchmarks are also the ones that suffer themselves. The benchmarks in this category include *mcf*, *swim*, *mgrid*, *art* and *equake*, all ones that exhibit a high L2 miss rate which further identifies the memory bus as the point of contention.

The mean speedup for all pairs was 1.20 under Hyper-Threading and 1.77 under SMP. This means that the performance of an SMP system is 48% better than a corresponding Hyper-Threading system for SPEC CPU2000.

A full table of results is not shown here but some interesting cases are described:

An example of expected behaviour is *equake* vs. *mesa*. This pair exhibits a system performance of just under 1 for context switching on a single processor, just under 2 for traditional SMP and a figure in the middle, 1.42, for Hyper-Threading. As *mesa* has a low cache miss rate it does not make much use of the memory bus so it not slowed by *equake*'s high L2 miss rate when running under SMP. Similarly for round robin

Best HT system throughput (1.50)	181.mcf	177.mesa
Int/FP	Int	FP
L1-D/L2 miss rates	high	low
D-TLB miss rate	high	low
Trace cache miss rate	low	high
IPC	very low	moderate
Worst HT system throughput (0.86)	**171.swim**	**172.mgrid**
Int/FP	FP	FP
L1-D miss rate	moderate	moderate
L2 miss rate	high	low
D-TLB miss rate	high	low
Trace cache miss rate	low	low
IPC	fairly low	fairly high
Stereotypical SMP vs HT performance	**183.equake**	**177.mesa**
Int/FP	FP	FP (less FP than equake)
L1-D/L2 miss rate	moderate	high
Trace cache miss rate	low	high
IPC	moderate	moderate

Table 3: Performance counter metrics for some interesting benchmark pairs. Metrics are for the benchmark running alone.

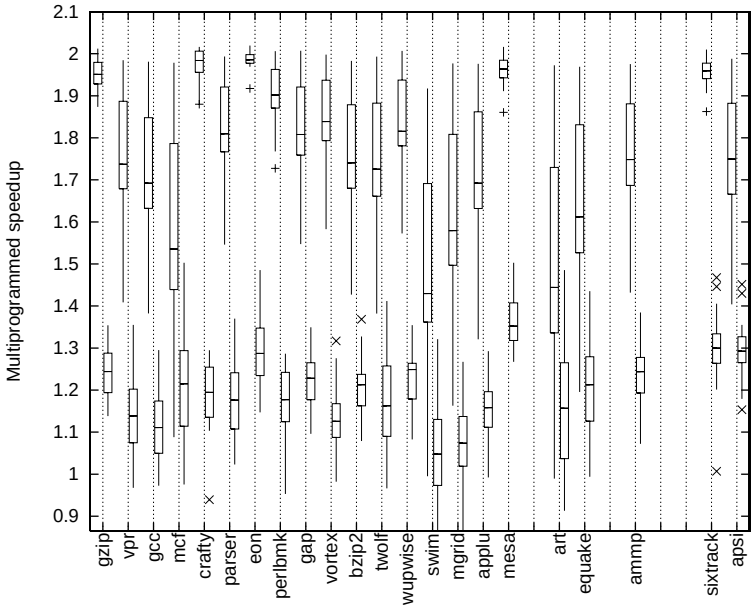

Figure 3: Multiprogrammed speedup of pairs of SPEC2000 benchmarks running on a Hyper-Threaded processor and non-Hyper-Threaded SMP. The right of each pair of box and whiskers is Hyper-Threading and the left is SMP.

context switching the small data footprint of *mesa* does not cause any significant eviction of data belonging to *equake*. Under Hyper-Threading there is little contention for the caches and the smaller fraction of floating-point instructions in *mesa* means that the workloads are heterogeneous and therefore can better utilise the processor's execution units.

art and *mcf* perform similarly under SMP, Hyper-Threading and round robin context switching. This is almost certainly due to the very high L1 and L2 cache miss rates and the corresponding low IPC they both achieve.

When executing under Hyper-Threading, *art* does better to the detriment of *mgrid* however under SMP the roles are reversed. Both have a high L1 miss rate but *art*'s is the highest of the pair. *art* has a high, and *mgrid* a fairly low L2 miss rate. Under Hyper-Threading *art* benefits most from the latency hiding offered by Hyper-Threading and causes harm to *mgrid* by polluting the L1-D cache. Under SMP there is no L1 interference so the *mgrid* outperforms *art* due to the lower L2 miss rate of the former.

When running against another copy of itself *vortex* has virtually no speedup running under Hyper-Threading compared to context switching. Under SMP there is almost no penalty which is due to the fairly low memory bus utilisation. As mentioned above, there is nothing particularly special about this benchmark's performance counter metrics to explain the low performance under Hyper-Threading.

vortex and *mcf* running under SMP take a notable (20% and 15% respectively) performance hit compared to running alone. This is due to a moderate L2 miss rates causing increased bus utilisation. Performance under Hyper-Threading shows *vortex* suffering a large performance loss (20% lower than if it only had half the CPU time) while *mcf* does particularly well. The latter has a low IPC due to its high cache miss rates so benefits from latency hiding. *vortex* has a fairly low L1-D miss rate which is harmed by the competing thread.

gzip with its very low L2 and trace cache miss rates, moderate L1-D miss rate and large number of memory accesses always does well under SMP due to the lack of bus contention but has a moderate and mixed performance under Hyper-Threading. *gzip* is vulnerable under Hyper-Threading due to its high IPC and low L2 miss rate meaning it is already making very good use of the processor's resources. Any other thread will take away resource and slow *gzip* down.

5 Conclusions

We have measured the mutual effect of processes simultaneously executing on the Intel Pentium 4 processor with Hyper-Threading. We have independently confirmed similar measurements made by Tuck and Tullsen [12] showing speedups for individual benchmarks of up to 30 to 40% (with a high variance) compared to sequential execution. We have expanded on these results to consider the bias between the simultaneously executing processes and shown that some pairings can exhibit a performance bias of up to 70:30. Using performance counters we have shown that many results can be explained by considering cache miss rates and resource requirement heterogeneity in general. Whilst the interactions are too complex to be able to give a simple formula for predicting performance, a general rule of thumb is that threads with high cache miss rates can have a detrimental effect on simultaneously executing threads. Those with high L1 miss rates tend to benefit from the latency hiding provided by Hyper-Threading.

We have compared the multiprogrammed performance of Hyper-Threading with traditional symmetric multiprocessing (SMP) and shown that although the throughput is always higher with SMP as would be expected, the performance gap between Hyper-Threading and SMP is not as large as may be expected. This is important given the economic and power consumption benefits of having a single physical processor package.

These measurements are part of a larger study of operating system support for SMT processors. Of relevance to this paper is the development of a process scheduler that is able to best exploit the processor while avoiding coscheduling poorly performing pairs of processes. We are using data from processor performance counters to influence the scheduling decisions and avoiding the need to have *a priori* knowledge of the process' characteristics. We believe that dynamic, feedback-directed scheduling is important as it can deal with complex thread interactions which may differ between microarchitecture versions. We have goals similar to Snavely *et al.* [9, 10] but our scheduler is designed to constantly adapt to changing workloads and phases of execution without having to go through a sampling phase.

6 Acknowledgements

The authors would like to thank Tim Harris, Keir Fraser, Steve Hand and Andrew Warfield of the University of Cambridge Computer Laboratory for helpful discussions and feedback on earlier drafts of this paper. The authors would also like to thank the reviewers for their constructive and helpful comments.

References

[1] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, Oct. 1997.

[2] D. Grunwald and S. Ghiasi. Microarchitectural denial of service: Insuring microarchitectural fairness. In *Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO-35)*, pages 409–418. IEEE Computer Society, Nov. 2002.

[3] G. Hinton, D. Sager, M. Upton, D. Boggs D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 5(1):1–13, Feb. 2001.

[4] J. L. Lo, L. A. Barroso, S. J. Eggers K. Gharachorloo, H. M. Levy, and S. S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA '98)*, pages 39–50. ACM Press, June 1998.

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

[6] D. T. Marr, F. Binns, D. L. Hill, G. Hinton D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(2):1–12, Feb. 2002.

[7] J. A. Redstone, S. J. Eggers, and H. M. Levy. An analysis of operating system behaviour on a simultaneous multithreaded architecture. In *Proceedings of the 9th International Conference on Architectural Support for Programming Langauages and Operating Systems (ASPLOS '00)*, pages 245–256. ACM Press, Nov. 2000.

[8] A. Snavely, N. Mitchell, L. Carter, J. Ferrante and D. M. Tullsen. Explorations in symbiosis on two multithreaded architectures. In *Workshop on Multi-Threaded Execution, Architectures and Compilers*, Jan. 1999.

[9] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Langauages and Operating Systems (ASPLOS '00)*, pages 234–244. ACM Press, Nov. 2000.

[10] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the 2002 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '02)*, pages 66–76. ACM Press, June 2002.

[11] The Standard Performance Evaluation Corporation, http://www.spec.org/.

[12] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '2003)*, pages 26–34. IEEE Computer Society, Sept. 2003.

[13] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, pages 819–828. Computer Measurement Group, Dec. 1996.

[14] D. M. Tullsen, S. J. Eggers, J. S. Emer, and H. M. Levy. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23th International Symposium on Computer Architecture (ISCA '96)*, pages 191–202. ACM Press, May 1996.

[15] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th International Symposium on Computer Architecture (ISCA '95)*, pages 392–403. ACM Press, June 1995.

[16] T. Ungerer, B. Robič, and J. Šilc. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35(1):29–63, Mar. 2003.

[17] D. Vianney. Hyper-Threading speeds Linux. *IBM developerWorks*, Jan. 2003.