Hyper-Threading Aware Process Scheduling Heuristics

James R. Bulpin and Ian A. Pratt
University of Cambridge Computer Laboratory
James.Bulpin@cl.cam.ac.uk, http://www.cl.cam.ac.uk/netos

Abstract

Intel Corporation's "Hyper-Threading Technology" is the first commercial implementation of simultaneous multithreading. Hyper-Threading allows a single physical processor to execute two heavyweight threads (processes) at the same time, dynamically sharing processor resources. This dynamic sharing of resources, particularly caches, causes a wide variety of inter-thread behaviour. Threads competing for the same resource can experience a low combined throughput.

Hyper-Threads are abstracted by the hardware as logical processors. Current generation operating systems are aware of the logical-physical processor hierarchy and are able to perform simple load-balancing. However, the particular resource requirements of the individual threads are not taken into account and sub-optimal schedules can arise and remain undetected.

We present a method to incorporate knowledge of per-thread Hyper-Threading performance into a commodity scheduler through the use of hardware performance counters and the modification of dynamic priority.

1 Introduction

Simultaneous multithreaded (SMT) processors allow multiple threads to execute in parallel, with instructions from multiple threads able to be executed during the same cycle [10]. The availability of a large number of instructions increases the utilization of the processor because of the increased instruction-level parallelism.

Intel Corporation introduced the first commercially available implementation of SMT to the Pentium 4 [2] processor as "Hyper-Threading Technology" [3, 5, 4]. Hyper-Threading is now common in server and desktop Pentium 4 processors and becoming available in the mobile version. The individual Hyper-Threads of a physical processor are presented to the operating system as logical processors. Each logical processor can execute a heavyweight thread (process); the OS and applications need not be aware that the logical processors are sharing physical resources. However, some OS awareness of the processor hierarchy is desirable in order to avoid circumstances such as a two physical processor system having two runnable processes scheduled on the two logical processors of one package (and therefore sharing resources) while the other package remains idle. Current generation OSs such as Linux (version 2.6 and later versions of 2.4) and Windows XP have this awareness.

When processes share a physical processor the sharing of resources, including the fetch and issue bandwidth, means that they both run slower than they would do if they had exclusive use of the processor. In most cases the combined throughput of the processes is greater than the throughput of either one of them running exclusively — the system provides increased system-throughput at the expense of individual processes' throughput. The system-throughput "speedup" of running tasks using Hyper-Threading compared to running them sequentially is of the order of 20% [1, 9]. We have shown previously that there are a number of pathological combinations of workloads that can give a poor system-throughput speedup or give a biased per-process throughput [1]. We argue that the operating system process scheduler can improve throughput by trying to schedule processes simultaneously that have a good combined throughput. We use measurement of the processor to inform the scheduler of the realized performance.

2 Performance Estimation

In order to provide throughput-aware scheduling the OS needs to be able to quantify the current per-thread and system-wide throughput. It is not sufficient to measure throughput as instructions per cycle (IPC) because processes with natively low IPC would be misrepresented. We choose instead to express the throughput of a process as a *performance ratio* specified as its rate of execution under Hyper-Threading versus its rate of execution

when given exclusive use of the processor. An application that takes 60 seconds to execute in the exclusive mode and 100 seconds when running with another application under Hyper-Threading has a performance ratio of 0.6. The *system speedup* of a pair of simultaneously executing processes is defined as the sum of their performance ratios. Two instances of the previous example running together would have a system speedup of 1.2 — the 20% Hyper-Threading speedup described above.

The performance ratio and system speedup metrics both require knowledge of a process' exclusive mode execution time and are based on the complete execution of the process. In a running system the former is not known and the latter can only be known once the process has terminated by which time the knowledge is of little use. It is desirable to be able to estimate the performance ratio of a process while it is running. We want to be able to do this online using data from the processor hardware performance counters. A possible method is to look for a correlation between performance counter values and calculated performance; work on this estimation technique is ongoing, however, we present here a method used to derive a model for online performance ratio estimation using an analysis of a training workload set.

Using a similar technique to our previous measurement work [1] we executed pairs of SPEC CPU2000 benchmark applications on the two logical processors of a Hyper-Threaded processor; each application running in an infinite loop on its logical processor. Performance counter samples were taken at 100ms intervals with the counters configured to record for each logical processor the cache miss rates for the L1 data, trace- and L2 caches, instructions retired and floating-point operations. A stand-alone base dataset was generated by executing the benchmark applications with exclusive use of the processor, recording the number of instructions retired over time. Execution runs were split into 100 windows of equal instruction count. For each window the number of processor cycles taken to execute that window under both Hyper-Threading and exclusive mode were used to compute a performance ratio for that window. The performance counter samples from the Hyper-Threaded runs were interpolated and divided by the cycle counts to give events-per-cycle (EPC) data for each window. A set of 8 benchmark applications (integer and floating-point, covering a range of behaviours) were run in a cross-product with 3 runs of each pair, leading to a total of 16,800 windows each with EPC data for the events for both the application's own, and the "background" logical processor. A multiple linear regression analysis was performed using the EPC data as the explanatory variables and the application's performance ratio as the dependent variable. The coefficient of determination (the R^2 value), an indication of how much of the variability of the dependent variable can be explained by the explanatory variables, was 66.5%, a reasonably good correlation considering this estimate is only to be used as a heuristic. The coefficients for the explanatory variables are shown in Table 1 along with the mean EPC for each variable (shown to put the magnitudes of the variables into context). The fourth column of the table indicates the importance of each counter in the model by multiplying the standard deviation of that metric by the coefficient; a higher absolute value here shows a counter that has a greater effect on the predicted performance. Calculation of the *p-values* showed the L2 miss rate of the background process to be statistically insignificant (in practical terms this metric is covered largely by the IPC of the background process).

The experiment was repeated with a different subset of benchmark applications and the MLR model was used to predict the performance ratio for each window. The coefficient of correlation between the estimated and measured values was 0.853, a reasonably strong correlation.

We are investigating refinements to this model by considering other performance counters and input data.

3 Scheduler Modifications

Rather than design a Hyper-Threading aware scheduler from the ground up, we argue that gains can be made by making modifications to existing scheduler designs. We wish to keep existing functionality such as starvation avoidance, static priorities and (physical) processor affinity. A suitable location for Hyper-Threading awareness would be in the calculation of dynamic priority; a candidate runnable process could be given a higher dynamic priority if it is likely to perform well with the process currently executing on the other logical processor.

Our implementation uses the Linux 2.4.19 kernel.

Counter	**Coefficient** (to 3 S.F.)	**Mean events per 1000 Cycles** (to 3 S.F.)	**Importance** (coeff. x st.dev.
(Constant)	0.4010		
TC-subj	29.7000	0.554	26.2
L2-subj	55.7000	1.440	87.2
FP-subj	0.3520	52.0	29.8
Insts-subj	-0.0220	258	-4.3
L1-subj	2.1900	10.7	15.4
TC-back	32.7000	0.561	29.0
L2-back	1.5200	1.43	2.3
FP-back	-0.4180	52.6	-35.3
Insts-back	0.5060	256	99.7
L1-back	-3.5400	10.6	-25.3

Table 1: Multiple linear regression coefficients for estimating the performance ratio of the subject process. The performance counters for the logical processor executing the subject process are suffixed "subj" and those for the background process's logical processor, "back".

This kernel has basic Hyper-Threading support in areas other than the scheduler. We modify the `goodness()` function which is used to calculate the dynamic priority for each runnable task when a scheduling decision is being made; the task with the highest goodness is executed. We present two algorithms: "*tryhard*" which biases the goodness of a candidate process by how well it has performed previously when running with the process on the other logical processor, and "*plan*" which uses a user-space tool to process performance ratio data and produce a scheduling plan to be implemented (as closely as possible) in a gang-scheduling manner.

For both algorithms the kernel keeps a record of the estimated system-speedups of pairs of processes. The current *tryhard* implementation uses a small hash table based on the process identifiers (PIDs). The performance counter model described above is used for the estimates. The goodness modification is to lookup the recorded estimate for the pair of PIDs of the candidate process and the process currently running on the other logical processor.

For each process *p*, *plan* records the three other processes that have given the highest estimated system-speedups when running simultaneously with *p*. Periodically a user-space tool reads this data for all processes and greedily selects pairs with the highest estimated system-speedup. The tool feeds this plan back to the scheduler which heavily biases goodness in order to approximate gang-scheduling of the planned pairs. Any processes not in the plan, or those created after the planning cycle, will still run when processes in the plan block or exhaust their time-slice. For both algorithms the process time-slices are respected so starvation avoidance and static priorities are still available.

4 Evaluation

The evaluation machine was an Intel SE7501 based 2.4GHz Pentium 4 Xeon system with 1GB of DDR memory. RedHat 7.3 was used, the benchmarks were compiled with gcc 2.96. A single physical processor with Hyper-Threading enabled was used for the experiments. The scheduling algorithms were evaluated with sets of benchmark applications from the SPEC CPU2000 suite:

Set A: *164.gzip*, *186.crafty*, *171.swim*, *179.art*.

Set B: *164.gzip*, *181.mcf*, *252.eon*, *179.art*, *177.mesa*.

Set C: *164.gzip*, *186.crafty*, *197.parser*, *255.vortex*, *300.twolf*, *172.mgrid*, *173.applu*, *179.art*, *183.equake*, *200.sixtrack*.

Each benchmark application was run in an infinite loop. Each experiment was run for a length of time sufficient to allow each application to run at least once. The experiment was repeated three times for each benchmark set. The individual application execution times were compared to exclusive mode times to get a performance ratio similar to that described above. The sum of the performance ratios for the benchmarks in the set gives a system-speedup. The execution time for the entire application, rather than a finer granularity, was used in order to assess the over all effect of the scheduling algorithm adapting to the effects of Hyper-Threading.

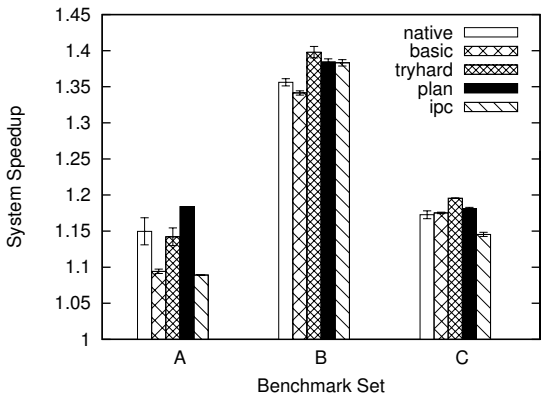

Figure 1: System-speedups for benchmark sets running under the different scheduling algorithms.

Each benchmark set was run with both *tryhard* and *plan*; the stock Linux 2.4.19 scheduler "*native*"; the same modified to provide physical, rather than logical, processor affinity "*basic*"; and an IPC-maximizing scheme using rolling-average IPC for each process and a goodness modification to greedily select tasks with the highest IPC (inspired by Parekh *et al*'s "G_IPC" algorithm [7]).

Figure 1 shows the system-speedups (relative to running the tasks sequentially) for each benchmark set with each scheduler. Improvements over *native* of up to 3.2% are seen; this figure is comparable with other work in the field and is a reasonable fraction of the 20% mean speedup provided by Hyper-Threading itself. The *tryhard* scheduler does reasonably well on benchmark sets B and C but results in a small slowdown on A: the four applications execute in a lock-step, round-robin fashion which *tryhard* is unable to break. It results in the same schedule as *native* but suffers the overhead of estimating performance. This is an example of worst-case behaviour that would probably be mitigated with a real, changing workload. *plan* provides a speedup on all sets.

The fairness of the schedulers was tested by considering the variance in the individual performance ratios of the benchmarks within a set. *tryhard*, *plan* and *basic* were as fair as, or fairer than *native*. The per-process time-slices were retained which meant that applications with low estimated performances were able to run once the better pairings had exhausted their time-slices. As would be expected, *ipc* was biased towards high-IPC tasks, however, the use of process time-slices meant that complete starvation was avoided. The algorithms were also tested for their respect of static priorities ("nice" values); both *plan* and *tryhard* behaved correctly. This

behaviour is a result of retaining the time-slices; a higher priority process is given a larger time-slice. Again, *ipc* penalized low-IPC processes but this was partially corrected by the retention of the time-slice mechanism.

5 Related Work

Parekh *et al* introduced the idea of thread-sensitive scheduling [7]. They evaluated scheduling algorithms based on maximizing a particular metric, such as IPC or cache miss rates, for the set of jobs chosen in each quantum. The algorithm greedily selected jobs with the highest metric; there was no mechanism to prevent starvation. They found that maximizing the IPC was the best performing algorithm over all their tests. Snavely *et al*'s "SOS" (sample, optimize, symbios) "symbiotic" scheduler sampled different combinations of jobs and recorded a selection of performance metrics for each *jobmix* [8]. The scheduler then optimized the schedule based on this data executed the selected jobmixes during the "symbios" phase. Nakajima and Pallipadi used a user-space tool that read data from processor performance counters and changed the package affinity of processes in a two package, each of two Hyper-Threads, system [6]. They aimed to balance load, mainly in terms of floating point and level 2 cache requirements, between the two packages. They measured speedups over a standard scheduler of approximately 3% and 6% on two test sets chosen to exhibit uneven demands for resources. They only investigated workloads with four active processes, the same number as the system had logical processors. The technique could extend to scenarios with more processes than processors however the infrequent performance counter sampling can hide a particularly high or low load of one of the processes sharing time on a logical processor.

6 Conclusion and Further Work

We have introduced a practical technique for introducing awareness of the performance effects of Hyper-Threading into a production process scheduler. We have demonstrated that throughput gains are possible and of a similar magnitude to alternative user-space based schemes. Our algorithms respect static priorities and starvation avoidance. The work on these algorithms is ongoing. We are investigating better performance estimation methods and looking at the sensitivity of the algorithms to the accuracy of the estimation. We are considering implementation modifications to allow learned data to be inherited by child processes or through subsequent instantiations of the same application.

The scheduling heuristics were demonstrated using the standard Linux 2.4 scheduler – a single-queue dynamic priority based scheduler where priority is calculated for each runnable task at each rescheduling point. Linux 2.6 introduced the "O(1)" scheduler which maintains a run queue per processor and does not perform goodness calculations for each process at each reschedule point. The independence of scheduling between the processors complicates coordination of pairs of tasks. We plan to investigate further how our heuristics could be applied to Linux 2.6.

Acknowledgements

We would like to thank our shepherd, Vivek Pai, and the anonymous reviewers. James Bulpin was funded by a CASE award from EPSRC and Marconi Corp. plc.

References

[1] J. R. Bulpin and I. A. Pratt. Multiprogramming performance of the Pentium 4 with Hyper-Threading. In *Third Annual Workshop on Duplicating, Deconstruction and Debunking (at ISCA'04)*, pages 53–62, June 2004.

[2] 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.

[3] Intel Corporation. *Introduction to Hyper-Threading Technology*, 2001.

[4] D. Koufaty and D. T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–64, 2003.

[5] 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.

[6] J. Nakajima and V. Pallipadi. Enhancements for Hyper-Threading technology in the operating system — seeking the optimal scheduling. In *Proceedings of the 2nd Workshop on Industrial Experiences with Systems Software*. The USENIX Association, Dec. 2002.

[7] S. S. Parekh, S. J. Eggers, H. M. Levy, and J. L. Lo. Thread-sensitive scheduling for SMT processors. Technical Report 2000-04-02, University of Washington, June 2000.

[8] 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 Languages and Operating Systems (ASPLOS '00)*, pages 234–244. ACM Press, Nov. 2000.

[9] 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.

[10] 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. IEEE Computer Society, June 1995.