Increasing Throughput of Multiprogram HPC Workloads: Evaluating a SMT Co-scheduling Approach

Extended Abstract

Elias Lundmark
Computer Engineering
University West, Sweden
elias.lundmark@outlook.com

Chris Persson
Computer Engineering
University West, Sweden
chris.s.persson@gmail.com

Andreas de Blanche
Department of Engineering Science
University West, Sweden
andreas.de-blanche@hv.se

Thomas Lundqvist
Department of Engineering Science
University West, Sweden
thomas.lundqvist@hv.se

ABSTRACT

Simultaneous Multithreading (SMT) is a technique that allows for more efficient processor utilization by scheduling multiple threads on a single physical core. Previous research have shown an average throughput increase of around 20% with an SMT level of two, e.g. two threads per core. However, a bad combination of threads can actually result in decreased performance. To be conservative, many HPC-systems have SMT disabled, thus, limiting the number of scheduling slots in the system to one per core. However, for SMT to not hurt performance, we need to determine which threads should share a core. In this poster, we use 30 random SPEC CPU job mixed on a twelve-core Broadwell based node, to study the impact of enabling SMT using two different co-scheduling strategies. The results show that SMT can increase performance especially when using no-same-program co-scheduling.

1 INTRODUCTION

Simultaneous Multithreading (SMT) is a technique where multiple co-executing threads share the functional units, buffers, and caches of a processing core. This allows for more efficient processor utilization since scheduling multiple threads on a single processor core will better exploit the instruction level parallelism as it mixes instructions from two or more threads.

A bit surprising, many HPC-systems have SMT turned off, thus, limiting the scheduling slots in the system to one per core. Enabling hyperthreading (HT), Intels SMT technology, would double the number of available slots [6]. However, more slots do not automatically lead to higher performance. In [9], Tuck and Tullsen observed system throughput increases ranging from 0.90 to 1.58 when enabling SMT, and an average speedup of 1.20, when running the SPEC benchmarks. Thus, some of the benchmarks actually showed a slowdown of up to 10%. To be safe, we should keep SMT disabled to avoid possible slowdowns. However, this neglects possible speedups and according to Tuck and Tullsen [9], the problem with obtaining high throughput lies in determining which programs should be co-scheduled, i.e. share a physical core.

If the system is exclusively used by a few programs the co-scheduling effects can be determined experimentally, but as the number of programs increases, the experimental approach soon becomes unfeasible. Several co-scheduling schemes aimed at achieving high throughput have been suggested. Snavely and Tullsen [7] achieved an average throughput increase of 17% using; Sample, Optimize, Symbiosis (SOS) which samples hardware counters and uses heuristics to determine which processes are suitable to execute together. Eeckhout et al. [5] introduced a probabilistic model for co-scheduling on SMT processors. A simple approach needing no pre-runtime analysis is suggested in [2, 3] that identify instances of the same program as being more likely to heavily utilize the same resources in a compute node, like caches, memory buses, or disks, than a random set of programs. In [2–4] it was shown that disallowing several instances of the NAS benchmarks to be co-scheduled on the same compute node increases the throughput compared to random co-scheduling.

The aim of this pre-study is to evaluate the job co-scheduling principle from [3]: Will disallowing the co-scheduling of several instances of the same program affect the throughput positively when applied to a SMT enabled core instead of a compute node? Furthermore, we assess if enabling SMT, thus practically doubling the number of available slots, will, in general, increase throughput. Our evaluation assumes an HPC system that is used for many different programs.

The evaluation was performed on a Xeon E5-2650v4 node with 32 GB ram and Centos 7.3. The E5-2650 has 12 physical cores and supports Intel's Hyperthreading [8] with an SMT level of two. All experiments rely on the SPEC CPU benchmarks [1]. Furthermore, all frequency scaling techniques were turned off. In the next section, we first do a full factor evaluation, based on the SPEC CPU benchmark, to examine the effect of co-scheduling several instances of the same program. Then we evaluate the SMT performance of different job mixes of up to 24 threads.

2 FULL FACTOR EVALUATION

First, we evaluate the effect of disallowing the execution of two instances of the same SPEC benchmark on an SMT enabled core. This is done by executing all possible combinations of the benchmarks and calculating the speedup each same program pair gained compared to executing the program serially. The line in Fig. 2 shows the average speedup for all possible SMT combinations excluding two instances of the program on the X-axis. The X shows the

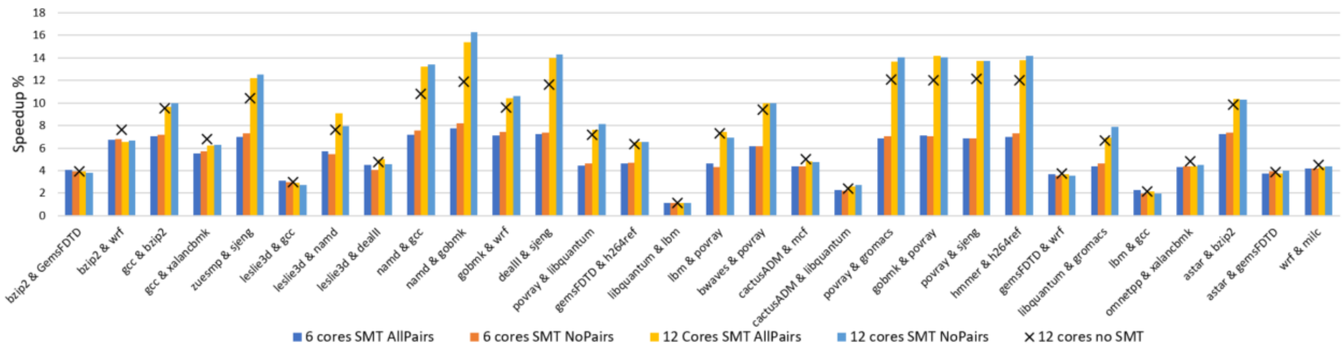

Figure 1: Average speedup for all SMT combinations excluding two instances of the same program. X marks the speedup of two co-scheduled instances of the same program.

speedup gained when two instances of the same benchmark are co-scheduled.

Interestingly, in the entire test-set, involving 784 combinations, only one has a negative speedup and that is when two instances of libquantum share a core. Furthermore, there are only two cases, out of 28, where co-scheduling two instances of the same program has a positive effect on the average speedup.

3 FULL SYSTEM EVALUATION

To evaluate a full system, we created 30 job cases, each consisting of multiple instances of two random SPEC CPU benchmarks. To establish a baseline six instances of each benchmark were executed on one core each with SMT turned off (12 jobs on 12 cores), marked by X in Fig. 1. The job mixes were then executed using six and twelve cores with SMT enabled using two co-scheduling algorithms. AllPairs: two identical processes are placed on each core and NoPairs: one process from each benchmark is placed on each core.

Compared to the twelve-core baseline, six cores with SMT enabled had a lower throughput in 27 cases using both AllPairs and NoPairs. Six SMT cores were, on average 39% (AllPairs) and 38% (NoPairs) slower then twelve cores with SMT disabled.

When enabling SMT on twelve cores (24 threads), 20 cases perform better then the baseline using AllPairs and 18 performs better using NoPairs. Nonetheless, the average twelve-core NoPairs throughput is 10% higher then the twelve-core baseline while the AllPairs throughput is only 8% higher.

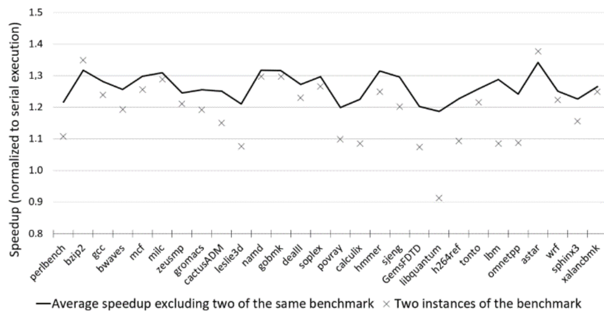

Figure 2: Average speedup for all SMT combinations excluding two instances of the same program. X marks the speedup of two co-scheduled instances of the same program.

4 CONCLUSIONS

We performed an experimental study comparing the throughput of 30 random SPEC CPU job mixed on a Broadwell based node using two different co-scheduling strategies. In conclusion, a six-core processor with SMT enabled has 38% lower throughput than a twelve-core processor without SMT but when enabling SMT the throughput increases by 10%. Hence, downgrading the processor and enabling SMT will decrease the throughput but enabling SMT will increase it.

Also, disallowing several instances of the same program to be co-scheduled on the same SMT enabled core increased the throughput by, on average, a few percentage points.

REFERENCES

[1] SPEC cpu2006. [n. d.]. Standard Performance Evaluation Corporation. ([n. d.]). Retrieved August 6, 2017 from https://www.spec.org/cpu2006/

[2] Andreas de Blanche and Thomas Lundqvist. 2015. Addressing characterization methods for memory contention aware co-scheduling. *Springer Journal of Supercomputing* 71, 4 (2015), 1451–1483.

[3] Andreas de Blanche and Thomas Lundqvist. 2016. Terrible twins: A simple scheme to avoid bad co-schedule. In *Workshop on Co-Scheduling of HPC Applications (COSH), HIPEAC 2016.* 1–6.

[4] Andreas de Blanche and Thomas. Lundqvist. 2017. Disallowing same-program co-schedules to improve efficiency in quad-core servers. In *Workshop on Co-Scheduling of HPC Applications (COSH), HIPEAC.* 13–18.

[5] Stijn Eyerman and Lieven Eeckhout. 2012. Probabilistic modeling for job symbiosis scheduling on smt processors. *ACM Transactions on Architecture and Code Optimization* 9, 2 (2012).

[6] Leo Porter, Adam Jundt William A. Ward Jr Roy Campbell Michael A. Laurenzano, Ananta Tiwari, and Laura Carrington. 2015. Making the Most of SMT in HPC. *ACM Transactions on Architecture and Code Optimization* 11, 4 (2015), 1–26.

[7] Allan Snavely and Dean M. Tullsen. 2000. Symbiotic jobscheduling for a simultaneous multithreading processor. *ACM SIGPLAN Notices* 35, 11 (2000), 234–244.

[8] Robert Hood David Barker Piyush Mehrotra Subhash Saini, Haoqiang Jin and Rupak Biswas. 2011. The impact of hyper-threading on processor resource utilization in production applications. In *International Conference on High Performance Computing (HiPC).* IEEE, 1–10.

[9] Nathan Tuck and Dean M. Tullsen. 2003. Initial observations of the simultaneous multithreading pentium 4 processor. In *International Conference on Parallel Architectures and Compilation Techniques (PACT).* IEEE.