Micro-op Fission: Hyper-threading Without the Hyper-headache

Robert Koutsoyannis, Anthony Cartolano and Daniel S. McFarlin
Carnegie Mellon University, Dept. of Electrical and Computer Engineering
{rjkousto, acartola, dmcfarli}@ece.cmu.edu

Introduction

The proliferation of Simultaneous Multithreading (SMT), also called Hyper-threading, in commercial processors such as the IBM Power7 and the Intel Core i7, offers the possibility of efficient fine-grain loop parallelization due to low HW thread overhead. At the same time, traditional, nonparallelizing loop optimizations are experiencing diminishing returns as they increase pressure on the i-cache, decoder and register file; these dense, monolithic CAM/RAM structures are scaling poorly in modern deep sub-micron processes compared to the clustered CAM/RAM structures used to implement SMT[1] .

SMT's performance potential however is undermined by two major factors: heavyweight software threading interfaces and the limited fetch/decode bandwidth of all commercial SMT CPUs. First, the popular threading APIs like OpenMP are designed for general purpose threading which results in high overhead as software must mediate the entire thread lifetime. Second, all commercial SMT CPUs use a time-division multiplexing fetch/decode scheme for SMT threads; the frontend is only fetching/decoding from one thread per cycle. This design can starve the backend of the CPU especially when SMT is used for numerical kernels. Our recent attempt to parallelize a numerical kernel using OpenMP on SMT cores resulted in at least a 10% slowdown on the Intel Core i7 and is consistent with the slowdowns observed in [2].

Approach

This paper proposes the use of SMT-based microthreads for fine-grain loop parallelization. Microthreads were first proposed in [3] and are lightweight, ephemeral, HW helper threads spawned by an explicit ISA instruction or HW events. These threads then run in tandem or ahead of a general purpose HW thread. Microthreading enhances the performance of **single-threaded** code by increasing ILP mostly through runahead prefetching and branch resolution. Unlike our proposed SMT-based microthreads, "classic" microthreads are micro-coded programs that are extracted by the compiler and execute in their own, private HW contexts.

In contrast, our approach exploits **thread-level parallelism** (TLP) by statically "fusing" the iterations of a parallelized loop into a single sequential instruction stream at compile time and then "fissioning" this sequential stream into parallel streams at runtime. The fissioned instruction streams are steered by the decoder to separate SMT HW contexts. Steering occurs when the decoder converts the macro instructions into micro(μ)-ops. Our approach overcomes the traditional SMT fetch/decode bottleneck, keeps the wider backend fully fed and reduces fetch and decode power consumption.

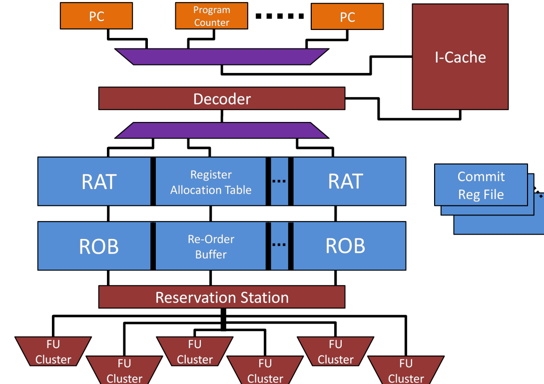

Figure 1: N-way Out-of-Order SMT Core

This paper makes two key contributions:

- We present a HW/SW co-optimization framework that leverages existing OpenMP constructs and existing SMT CPU HW to implement a microthreading infrastructure
- We demonstrate how this approach can be readily used to speedup (6% average on 2-Way SMT and 19% average on 4-Way SMT) a range of numerical kernels without programmer intervention on current and near term simulated SMT CPUs

Motivation

To help explain our approach, we describe the execution of a parallelized loop containing a body with four statements: A, B, C, D on two SMT cores: a conventional core and one that implements μ-op fission. First, we briefly describe a generic N-way out-of-order SMT core shown in Figure 1. This core has standard OOO architectural features with a few notable exceptions: there are N program counters, Register Alias Tables (RATs), Reorder Buffers (ROBs). Also, there is a multiplexer between the i-cache and a decoder and a demultiplexer between the decoder and the RATs. This arrangement is common to all SMT CPUs and highlights the chief SMT bottleneck; the narrow frontend.

We now describe the execution of the SMT parallelized loop on a 2-way SMT core possessing a fetch/decode and per RAT rename width of four instructions (shown in Figure 2). The core is executing the loop described above. We see the first loop iteration en route to the RAT stage of the pipeline. This iteration is colored orange to represent its SMT thread assignment. Note that the the second iteration has just been fetched by the second SMT thread's PC (shown in blue) and is being decoded. In the next cycle it will be dispatched to the rename stage and the third loop iteration,

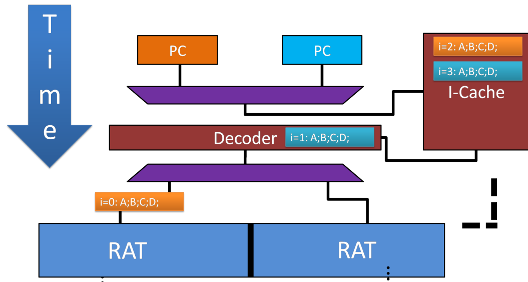

Figure 2: SMT Parallelized Loop Execution

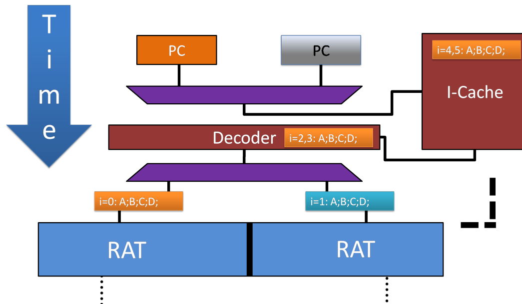

Figure 3: Micro-op Fission Parallelized Loop Execution

again assigned to the first (orange) SMT thread in cyclic fashion, will be fetched and deocded. The point is that even though we have an aggregate rename width of eight instrucitons (four per RAT times two RATs) we can only dispatch four due to the fetch/decode width limitations.

Now consider Figure 3, showing the same 2-way SMT core but this time one that implements μ-op fission. One major change is that the compiler has fused two adjacent iterations into a single iteration. In other words, we still fetch four instructions but we fission them after the decode stage into eight instructions, four instructions to each RAT. Note also that the program counter for the second thread has been grayed out signifying that the second SMT thread is idle. The main takeaway is that we only fetch from a single SMT thread but broadcast those decoded μ-ops marked as fused, by the compiler, to all subsequent backend SMT structures. It is this compiler directed broadcasting that is the essence of μ-op fission.

Experimental Setup and Results

Architectural Parameter(s)	Value(s)
Fetch/Decode/Dispatch/Issue/Commit Width	4
Load/Store Queue Size	48/32
ROB Size	256
Physical Register File Size	3 x 256
Load/Store/Arithmetic/FP units	2/2/2/2
L1/L2/L3 Cache Size, Latency (cycles)	16 KB/256 KB/4 MB, 1/5/8
Main Memory Latency	140 cycles

We simulated three OOO SMT cores on a cycle accurate x86 simulator, PTLSim, using the parameters described in the table above. The cores differed only in the number of SMT threads they support: one, two or four. We used five different already highly tuned numerical kernels written in C

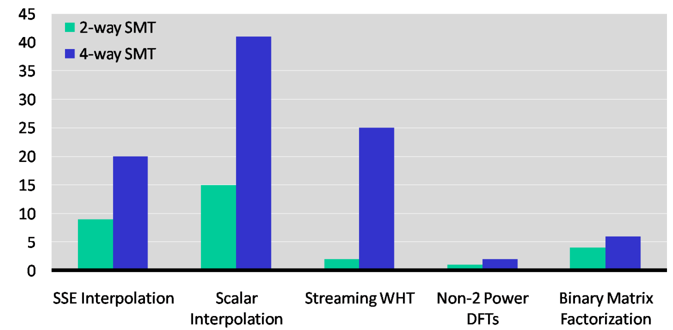

Figure 4: Performance Improvement of Micro-op Fission

and compiled using Intel ICC 11.1 with the -03 optimization flag. Space constraints prevent us for describing the kernels in detail but the first two are linear and radial interpolation kernels from a high resolution SAR image formation kernel. The WHT/DFT kernels are well known and the final kernel is a heuristic driven, brute force Binary Matrix Factorization kernel used in a super-optimizer. All kernels were simulated up to 10^7instructions.

Figure 4 shows the results of our experiments. Overall performance gains are attained even for issue-width (compute bound) kernels like the DFT due to improved functional unit utilization. The main performance driver for the interpolation and streaming kernels is the effective prefetching performed by cyclically distributing iterations amongst SMT threads [4]. The cache-lines fetched by the nth SMT thread are often subsequently used by the $0..n-1$ SMT threads in their later iterations. This implicit software prefetching is vital for interpolation kernels which often exhibit strides that are non-integral and/or non-linear and therefore difficult for most HW prefetchers to satisfy.

Summary and Future Work

We have demonstrated a lightweight compile-time/runtime optimization infrastructure for 2-way and 4-way OOO SMT cores that overcomes the traditional SMT bottlenecks resulting in performance improvements for a variety of numerical kernels. Our initial results have been promising with estimated power savings in the 10% realm as suggested by [5]. We hope to extend our approach to wider issue-width architectures and hybrid CMP/SMT designs.

References

[1] Z. Chishti and T. N. Vijaykumar, "Optimal power/performance pipeline depth for smt in scaled technologies," *IEEE Trans. Comput.*, vol. 57, no. 1, pp. 69–81, 2008.

[2] M. Curtis-Maury and T. Wang, "Integrating multiple forms of multithreaded execution on multi-smt systems: A study with scientific applications," in *QEST '05*, 2005, p. 199.

[3] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (ssmt)," in *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, 1999, pp. 186–195.

[4] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parekh, and D. M. Tullsen, "Tuning compiler optimizations for simultaneous multithreading," in *MICRO 30*, 1997, pp. 114–124.

[5] J. González, Q. Cai, P. Chaparro, G. Magklis, R. Rakvic, and A. González, "Thread fusion," in *ISLPED '08*, 2008, pp. 363–368.