

Thread Fusion

José González¹, Qiong Cai¹, Pedro Chaparro¹, Grigorios Magklis¹, Ryan Rakvic² and Antonio González¹

¹UPC-Intel Lab Barcelona
Barcelona, Spain

{pepe.gonzalez,qiongcai,pedro.chaparro.monferrer,
grigorios.magklis,antonio.gonzalez}@intel.com

²United States Naval Academy
Annapolis, Maryland, USA
rakvic@usna.edu

ABSTRACT

This work proposes Thread Fusion as an effective way of reducing power consumption when a Simultaneous Multi-Threaded (SMT) core is executing two threads from a homogeneous parallel application. Two dynamic instances of the same static instruction, each from a different thread are merged (fused) into a single instruction, consuming half of the resources of front-end pipeline stages. When the fused instruction is executed, it is cloned and it proceeds at full bandwidth. Our simulation results show average energy reduction of 10% with less than 1% impact on performance.

Categories and Subject Descriptors

C.1.2 [Computer Systems Organization]: Multiple Data Stream Architectures – *Parallel processors*.

General Terms

Performance, Design

Keywords

Thread Fusion, Multi-threaded Application, Chip Multi-processor, In-order pipeline, Microarchitecture, Computer Architecture, Low-power, Energy-aware

1. INTRODUCTION

In recent years, chip multiprocessors (CMPs) [19] have emerged as the next step in computer evolution. It is believed that CMPs are one of the most promising ways to address the ever increasing power and power density challenges faced by today's designs, while also sustaining the current performance growth trend.

The Niagara architecture [14], Sun's Throughput Computing [22] and Intel's Tera-scale computing [2] are steps in this direction. The main design goal behind these initiatives is a CMP with a large number of cores and even greater number of threads. One way to achieve this is by using simple, small cores, such as in-order cores, capable of SMT execution [23]. Niagara follows such a design [14].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. *ISLPED'08*, August 11–13, 2008, Bangalore, India. Copyright 2008 ACM 978-1-60558-109-5/08/08...\$5.00.

Despite their simplicity, integrating dozens of cores on a single chip presents a challenge from the energy point of view [12]. The cost of cooling, the limitations on total and maximum power dissipation, and the existence of multiple hot spots [5] in such architectures are still open research topics.

Such future processors are envisioned to execute mainly parallel applications [11][15]. These parallel applications usually consist of several parallel sections (*e.g.*, parallel loops) that end at a barrier which synchronizes all parallel threads before resuming execution. Parallel sections can be classified into either balanced or unbalanced [1][18].

In a balanced section, all parallel threads reach the barriers almost at the same time. Usually, these parallel sections are *data parallel* or *loop parallel*, which means that all threads run pseudo-identical code over different data. In addition, the cache miss rate of these threads is very similar. In an unbalanced section, some threads reach the synchronization points much earlier than others, which causes faster threads to wait for the slower ones.

This work focuses on parallel and balanced applications. We have observed that threads from these applications most of the time run the same code simultaneously. When two of these parallel threads are running on the same SMT core, much of the activity is unnecessarily repeated since both instruction streams are similar.

In this paper, we devise a scheme to reduce redundant activity when two parallel threads from the same application run together on a two-way SMT in-order core. The fact that both parallel threads execute pseudo-identical code is exploited by forcing them to execute the same instructions at the same time, referred to as *lock-step mode*. This way, the activity of the processor front-end is reduced, since only half of the resources are used to execute the two threads.

We propose *Thread Fusion* to implement the aforementioned idea. Two dynamic instances of the same static instruction from two parallel threads are merged (*fused*) dynamically into a single instruction. This fused instruction behaves as if it were a single instruction until it is issued. Therefore, it uses a single decoder, occupies only one entry in the instruction buffer, accesses only one scoreboard table, etc. At issue, the fused instruction is cloned into two instructions, which execute in parallel using the resources of the SMT in-order core. Total activity, and thus energy, is reduced since fused execution utilizes roughly half of front-end resources compared to non-fused execution.

Our experiments show energy savings of 10% on average with less than 1% slowdown, and up to 18% energy savings with 5% speedup for some applications.

The rest of this paper is organized as follows. In Section 2, we describe the architecture of our baseline SMT in-order core. Section 3 presents a motivating example for Thread Fusion. Section 4 explains how Thread Fusion works and our proposed

implementation. In Section 5, we discuss our experimental results and in Section 6 we conclude the paper.

2. BASELINE MICROARCHITECTURE

As stated before, we believe that, in order to build a CMP processor with dozens of cores, each core must be simple and efficient. An in-order core utilizing SMT to run multiple threads satisfies these requirements. The main drawback of an in-order core is the performance penalty of cache misses. Instructions are issued in program order, and a cache miss causes a full stall in the pipeline. In order to address this issue, the core is enhanced with SMT capabilities. In an SMT, two threads share the core resources. In particular, issue slots that cannot be used by one thread because of a cache miss are available to the other thread. This way high execution bandwidth can be maintained. The pipeline stages of our proposed processor are the following:

Fetch: Intel[®] 64 instructions are fetched from the instruction cache and stored in an instruction buffer waiting to be decoded. A *gshare* branch predictor is implemented in conjunction with a BTB.

```
#omp_parallel                // r2 is i * word size
for (i=0; i < limit; i++)    (1)  r3 = load r2(r4);
{                            (2)  r6 = load r2(r10)
    p=q[i];                  while:
    while (p!=NULL)         (3)  beq r3,0, outwhile
    {                        (4)  r5 = load 0(r3)
        b[i] = a[i] + p->elem; (5)  r11 = r5 + r6
        p = p->next;         (6)  r3 = load 4(r3)
    } ...                    (7)  jmp while
}                            (8)  store r11 r2(r12)
                             (9)  jmp while:
```

Figure 1 Example of a parallel loop and assembly.

Decode: Intel[®] 64 instructions are decoded into “RISC-like” instructions and inserted into a queue, where they remain until issued. There is one such queue per thread. These RISC-like instructions are also known as micro-ops. However, we will call them instructions since our mechanism works with any ISA.

Issue: instructions are issued in-order to the execution units as soon as their inputs are available. A scoreboard table keeps tracks of data dependences.

Register file: both integer and floating-point register files are accessed to read the source operands which can be obtained from the bypass as well, to allow back to back execution.

Address generation (AGU): both loads and stores compute their effective address in this stage.

Cache: the data cache is accessed. In case of a miss, the offending thread is stalled until the data arrives from the higher memory levels. The architecture includes a private L2 unified cache and a shared, on-chip L3 cache.

Execution: integer, floating-point and SIMD arithmetic instructions are executed in the respective functional units.

Write Back: the execution results are written back to the register files.

We assume this core to be part of a CMP microarchitecture, which consists of a number of cores each having a private first level instruction cache, a first level data cache, a second level unified cache, and a shared L3 cache connected to all cores through a bus. Caches are kept coherent through a MESI protocol.

Nevertheless, this paper focuses on the power/performance of one of these cores when running two parallel threads.

3. MOTIVATION

In this section, we utilize a simple example to further motivate our work. Figure 1 shows a parallel loop containing an innermost loop and the associated pseudo-assembly version of this code. Note that two parallel threads executing this loop on an SMT core will be eventually running the same piece of code several times. Assume the core synchronizes these threads and enters Thread Fused mode at the beginning of some iteration. Instructions 4 to 8 from both threads will be executed in lock-step mode. We will refer to the instruction that results from merging two identical instructions from different parallel threads as *fused instruction*.

Therefore two dynamic instances of instruction 4, each from a different thread, are fetched together using one fetch port as if they were a single instruction, creating a fused instruction. Then, the fused instruction is decoded and inserted in the instruction queue. Later on, the fused instruction is issued, utilizing the resources of a single instruction even though it represents two. Therefore it uses half of the resources that the two threads would use if they ran independently.

At this point, the fused instruction is cloned in order to compute the corresponding outputs. After this point, the instruction will utilize the same resources as two independent instructions would.

Figure 2 shows the utilization of the pipeline when two parallel threads run together. Figure 2 (a) shows the occupancy of the different stages when two threads execute independently in normal mode. All the resources are available for both threads. Figure 2 (b) shows the pipeline utilization when the two threads are fused and they run in lockstep mode.

Fetch, decode and issue stages have the occupancy of their resources halved, since a single fused instruction represents the same instruction for both threads. After issuing the fused instruction, execution occurs in parallel. This means that, theoretically, we should not expect any performance penalty with thread fusion since the processor bandwidth is not modified.

4. THREAD FUSION

This section describes our Thread Fusion implementation and the extensions needed in our in-order core. The proposed core has two execution modes:

Normal mode: instructions are processed as usual; two threads running together utilize their private resources including instructions queues, cache ports and architectural register file as well as shared blocks including functional units, and bypasses. We will refer to these threads as thread *A* and thread *B*.

Fused mode: both threads execute the same code; they are forced to run in lockstep mode. Energy is reduced by *fusing* their instructions and using half of the resources in the front end of the machine. Fused instructions utilize the instruction queue and the scoreboard of thread *A*.

4.1. Triggering Thread Fusion

The core is initially in normal mode. A *Synchronization Point (SP)* is used to switch to fused mode. A Synchronization Point is the first PC of an instruction that is visited frequently by the threads executing a parallel section. *SPs* are stored in a small table in the processor front-end.

If thread *A* fetches a PC that is a *SP*, it stops fetching instructions and waits for the other thread to reach the same PC. If thread *B*

fetches that PC, both are executing different iterations of the same parallel loop. The processor enters fused mode. If thread *B* does not reach the *SP* after a specified time interval (implemented through a watchdog timer), then thread *A* resumes execution. When the processor enters fused mode, the pipeline of thread *B* is drained in order to have all registers of that thread available. Then, the scoreboard of thread *A* can be used safely as the unique scoreboard for fused instructions.

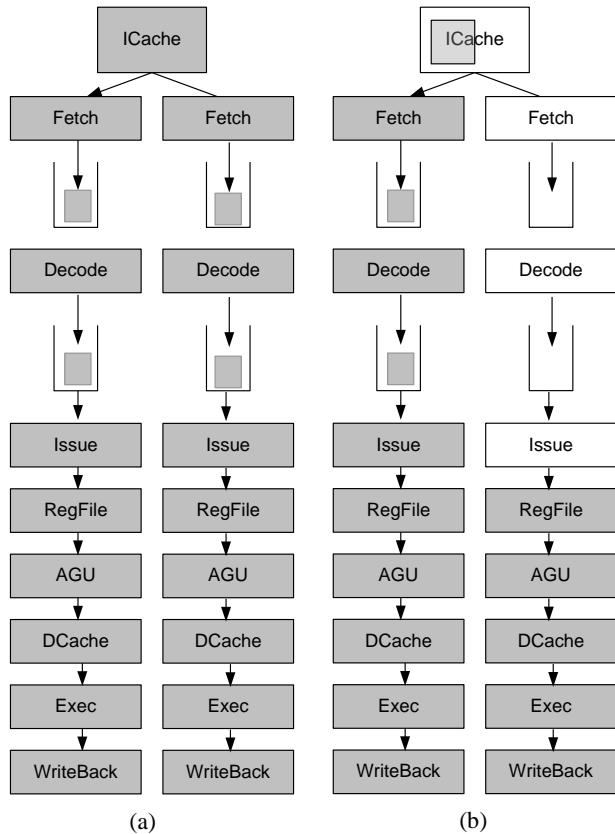


Figure 2. Pipeline utilization (a) 2 threads in normal mode (b) 2 threads in fused mode. In gray, the blocks used in the execution of instructions in both cases.

Synchronization Points can be inserted by the compiler via marking the instruction with a special hint, by the user with an OMP directive, or can be created dynamically by the microarchitecture (*e.g.*, by detecting backward branches, control-independent points in an *if-then-else* structure, etc.) Synchronization Points are stored in a small table located in the fetch (8 entries suffice).

4.2. Executing Fused Threads

Once the processor enters fused mode the pipeline behaves as follows:

Fetch

After synchronizing and draining the pipeline, fetching resumes by reading just one instruction per cycle, which is inserted in the buffer associated with thread *A*. Branches are predicted using

their respective history registers. If the prediction outcome is different for both threads, the processor switches to normal mode.

Decode

A fused instruction is read from the buffer of thread *A*. This instruction has the same encoding as an un-fused instruction, and is decoded following the normal procedure. This is possible because the fused instruction represents two *identical* instructions, so decoding has not changed. The only change is that we have less decode activity.

Issue

Fused instructions check the availability of their operands in the scoreboard of thread *A*. If all sources are ready, and this is the oldest pending instruction, then the fused instruction is issued for execution. Then, the scoreboard is accessed again using the destination register identifier in order to set the cycle when the result will be available for consumers.

The two instructions represented by the fused instruction may compute different values, but since they are using the same type of functional unit, these values will be produced at the same time. This explains why the system works with only one scoreboard.

After issued, the fused instruction is cloned and it acts as if two identical but independent instructions are flowing through the pipeline in parallel.

Register File

The fused instruction accesses the register files of both threads in order to read the sources. Note that although the instructions forming the fused one are identical, they belong to different iterations of a parallel loop, so the values of their sources may be different.

AGU

In this stage the fused instruction has already been cloned and it executes as two instructions. In case of a load or store fused instruction, both address generation units are engaged to compute two addresses that are then passed to the data cache.

Data Cache

Memory instructions access the data cache in this stage. In the issue stage, instructions dependent on loads are scheduled assuming a load hit. Fused loads access the data cache in parallel using both read ports. If both of them hit, the execution continues normally. If both of them miss, dependent instructions that have issued are flushed and are sent back to the instruction queue, waiting for the miss to resolve.

If one of the two loads represented in the fused instruction hits and the other misses we are presented with an interesting situation. In order to preserve correctness and to continue benefiting from fused mode, it is assumed that both load operations of the fused instruction have missed, fused instructions already issued are flushed and the fused thread is stalled until the miss is resolved. Contrary to normal mode execution, when two threads are fused, if one of them suffers a load miss, the other thread is also stalled. As we will see, this can be a reason for performance loss for Fused Threads.

Execution & Write Back

Integer, Floating Point and SIMD instructions execute in their respective functional units. If the processor has two functional units for a particular operation, the fused instruction executes in

both units simultaneously. Otherwise, the cloned instructions are executed serially in a pipelined fashion. In this case the observed latency of that particular operation is increased since both executions have to provide their results simultaneously to their dependent fused instructions. This happens only for some SIMD operations, and for integer multiplication and division.

After execution, the result is forwarded to the different bypass levels in order to allow back-to-back execution between consecutive fused instructions. The result is also written back in the corresponding register file of the two threads.

4.3. Enhancements

Failing or false synchronizations (when one of the threads arrives to a synchronization point and waits until the watchdog timer expires) constitute a source of performance loss. In order to reduce the number of failing synchronizations, a mechanism has been implemented to keep track of the combinations of PCs that constantly result to failing synchronizations.

Every time thread *A* fetches a *SP*, the current PC of thread *B* is stored in an additional column of the synchronization point table, in the same entry as the *SP*. If the synchronization fails, a two-bit saturating counter (also associated with that PC) is increased. When this counter exceeds a particular threshold, the next time thread *A* fetches that *SP* and thread *B* is fetching the associated PC, the synchronization is not initiated.

We have found experimentally that storing a “potentially” offending PC along with each *SP* is enough to capture most failing synchronizations. Usually this offending PC corresponds to a load that misses on cache. This means that, whereas one thread is stalled in the fetch stage waiting for the other thread to reach the synchronization point, the other thread is stalled on a cache miss.

In order to realize important energy savings, the processor must run in fused mode as much as possible. When the synchronization points are decided statically by either the user or the compiler, it may happen that they are not effective enough to get a high rate of fused instructions. Here we propose a mechanism to add new synchronization points dynamically.

Our algorithm is triggered every time the core exits fused mode due to a mismatch in the destination of two branches. The target PC is set as “candidate” for becoming a new synchronization point. The next time this candidate is fetched by any thread, the system counts the number of cycles it takes the other thread to reach that point.

If the number of cycles is below a given threshold, then a two-bit saturating counter associated to the candidate is increased. This counter represents the confidence of that PC as a real synchronization point. When the saturating counter is greater than 1, the candidate PC becomes a synchronization point. On the other hand, if the other thread does not reach the candidate before the threshold expires, the associated counter, as well as our confidence to this candidate, is decreased.

Regarding the size of the different structures, we propose to have a table with 8 synchronization points (either already established or being evaluated). Each entry will consist of the following fields:

- PC-SP: PC of the synchronization point. It is Either established by the user/compiler or chosen dynamically.
- SP-conf: two bit up-down saturated counter that assigns confidence to a candidate dynamic *SP* when it is being evaluated.

- PC-miss-SP: PC that usually causes a synchronization miss when one thread is waiting in the *SP*.
- PC-miss-SP-conf: two bit, up down saturated counter that is used to decide whether a particular (SP, miss-SP) pair is causing synchronization misses.

5. RESULTS

In this section, we present our experimental methodology followed by a discussion of the results we obtained for our proposed technique.

5.1. Simulation Methodology

Our experiments were carried out using an execution-driven simulator. The functional model of our simulator is using SoftSDV [24]. The simulator also incorporates a power model based on the activity factors and energy per access, similar to Wattch [4].

We simulate a multi-core architecture consisting of a number of SMT in-order cores connected through a bus. Each core has a private first level instruction (32 KB) and data cache (32KB) and a private and unified second level cache (512KB). The processor also includes a third level unified cache (8MB) shared among all cores. A MESI protocol has been implemented to maintain coherency among the caches.

The results presented in this work consider only one core, along with all the memory subsystem and interconnect. The simulated core is an in-order, 2-way, SMT microarchitecture. It fetches and decodes one Intel® 64 instruction per thread, per cycle. The core issues up to two instructions (micro-ops) per cycle, either from the same or from different threads. If both threads have ready instructions, each of them issues one. If one thread does not have ready instructions, both issue slots are assigned to the other thread.

5.2. Benchmarks

To evaluate our techniques, we use a combination of benchmarks from the SPECComp [1] and from the Recognition, Mining and Synthesis (RMS) Benchmark suite [6][10]. Each benchmark is parallelized for two threads and we focus on the loop that represents most of the execution time. The functional simulator is able to feed the performance model with both user and OS code. In our baseline configuration, each program is simulated until each one of the threads executes a minimum of 10M instructions. An equal number of instructions are executed in the configuration with our thread fusion algorithm.

From SPECComp, we selected *equake*, *wupwise* and *swim*. For these benchmarks, the parallel sections that we simulate cover 40%, 95% and 33% of the total execution time respectively.

From the RMS suite, we selected *dense_mmm* (kernel code implementing matrix multiplication, coverage 99%), *fimi* (finite itemset mining [8], 40% coverage), *gauss* (Gauss-Seidel iterative solver, coverage 99%), *genenet* (Bayesian network structure-learning problem, to measure regulatory relationships between genes [13], coverage 90%), *kmeans* (partition-based clustering algorithm for mining [21], coverage 99%), and *modulenet* (coverage 54%).

5.3. Thread Fusion Performance

This section presents the performance in terms of execution time and energy reduction when thread fusion is implemented in an in-

order core. For this experiment, we assume that synchronization points are set by either the compiler or the programmer at the beginning of loop iterations.

Figure 3 shows the execution time and energy consumption of a processor with thread fusion enabled normalized to a processor running parallel threads in non-fused mode. On average, the slowdown due to thread fusion is less than 1%, whereas energy is reduced by approximately 10%. *Gauss* is the application that experiences the biggest slowdown (16%) with an energy reduction of 13%. The sources of the slowdown are: (a) cache misses that occur only to one of the two loads belonging to a fused load and (b) missed synchronizations.

On the other hand, *fimi* runs 5% faster, consuming 18% less energy. In fused mode, up to two fused instructions can be issued per cycle as long as there are enough register file ports. This means that one fused branch and one fused integer instruction can be issued in parallel as long as each requires only one read port. Furthermore, one fused floating-point instruction can be issued along with one fused integer instruction. Therefore, thread fusion can also improve performance, such as in *fimi*, since the effective issue bandwidth is increased under some situations.

Figure 4 shows a breakdown of the distribution of the issue cycles among the following cases:

- mmFT: cycles in fused mode that the issue is stalled due to a miss in the data cache by either thread.

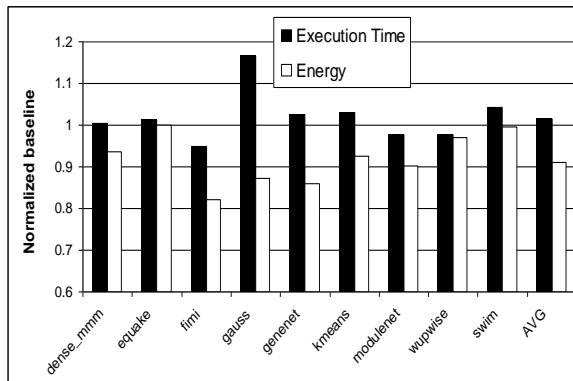


Figure 3. Execution time and energy consumption of fused threads normalized to a system running two parallel threads.

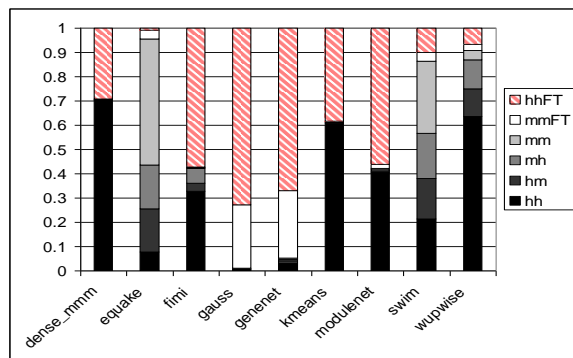


Figure 4. Distribution of issue cycles for Thread Fusion.

- hhFT: cycles in fused mode where the issue could proceed since there is not a pending load miss.
- mm: cycles that the processor is executing in normal mode but no instructions are issued since both threads have a pending miss.
- hh: cycles executing in normal mode where there is no pending miss.
- hm & mh: cycles executing in normal mode where one of the threads has a pending miss.

Figure 5 shows the above distribution of issue cycles when the processor is running in normal mode. Figure 6 plots both the percentage of cycles the processor is executing in fused mode and the percentage of synchronizations that were initiated but eventually failed.

Looking at Figure 4, we observe that *gauss* and *genenet* experience a high cache miss rate when running in fused mode (30%). In addition, Figure 6 shows that the processor executes this application in fused mode almost all the time (close to 100% for *gauss*).

As illustrated in Figure 5, both threads for *gauss* are stalled 5% of the time, whereas 15% of the time one of the threads is stalled due to a cache miss while the other one is executing at full speed. This explains the performance drop using Thread Fusion in this application. When fusing threads, the issue is stalled 30% of the cycles due to a miss in one or both of the instructions from a fused load.

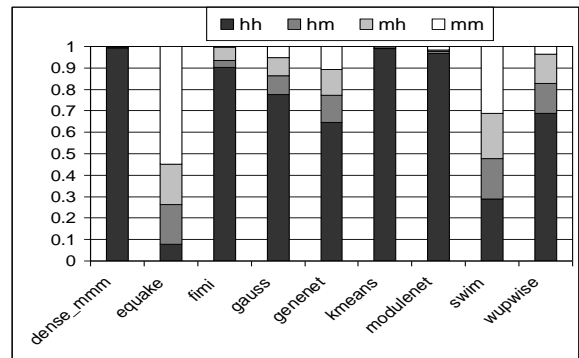


Figure 5. Distribution of issue cycles for the Baseline.

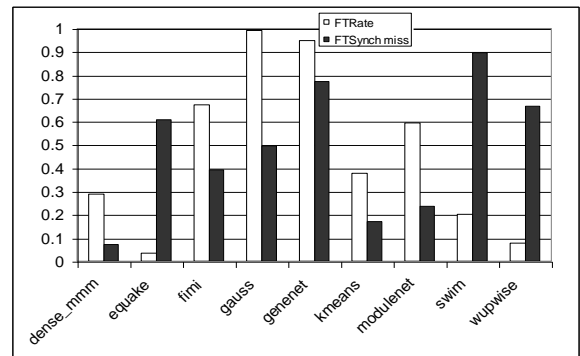


Figure 6. Percentage of cycles the processor is running in fused mode (FT Rate) and percentage of wrong synchronizations (FT Synch miss).

On the other hand, *fimi* and *dense_mmm* have a negligible cache miss rate. Whereas *fimi* has a high rate of fused instructions and achieves great energy reduction and even speedup, *dense_mmm* has a modest fused mode ratio and the energy reduction is smaller. *Modulenet* also experiences a similar behavior. Finally, *equake* shows negligible energy savings due to the small percentage of cycles running in fused mode. The reason for this is the high percentage of synchronizations missed combined with a small number of total synchronizations (not shown here).

6. RELATED WORK

The thrifty barrier [16] is an interesting proposal targeted to heterogeneous applications. Threads that arrive early to the barrier put their cores to low-power states. Dynamic Vectorization techniques [20][25] attempt to increase the ILP exploited by a processor by overlapping the execution of very regular loops with the loop continuation [25]. Replication and Widening [17] is another proposal to increase the available ILP that can be exploited on VLIW machines.

Micro-op Fusion [9] and instruction collapsing [3] are techniques that merge instructions into a single one to reduce energy consumption or increase bandwidth. For instance, two consecutive micro-ops from the same thread that meet some requirement are merged into a single micro-op in order to save energy.

The main differences between our work and previous works are the following. First, this study focuses on reducing power when two threads from a parallel application are running on an SMT core. The target of this work is homogeneous parallel sections. Therefore, our proposal can be combined with any of the previous studies on multicore systems. In particular, it fits perfectly with the thrifty barrier since it targets different types of applications. Second, with respect to the dynamic vectorization and the widening works [20][17][25], our work does not focus on improving performance. Our proposal attempts to reduce energy consumption when an SMT core is running two parallel and homogeneous threads by means of using less resources and reusing activity.

Finally, Thread Fusion can be implemented along with Micro-Op Fusion or Collapsing since an orthogonal and synergistic effect can be obtained. Once the processor enters the fused mode, Micro-Op fusion and/or Collapsing can be applied to further reduce energy or increase bandwidth.

7. CONCLUSIONS

This paper has presented Thread Fusion as an energy efficient way of executing parallel applications in SMT cores. It is based on the observation that threads from homogeneous applications often run the same code simultaneously. This is exploited by forcing them to run in lock-step mode.

Two identical instructions from two parallel threads are *fused* dynamically into a single instruction. This fused instruction behaves as if it were a single instruction in the front-end of the machine. Total activity and thus energy is reduced since fused execution utilizes roughly half of front-end's resources compared to non-fused execution.

Our experiments on average show energy savings of 10% with less than 1% slowdown, and up to 18% energy savings with 5% speedup for some applications.

8. REFERENCES

- [1] V. Aslot, M.J. Domeika, R. Eigenmann, G. Ger, W.B. Jones and B. Parady. "SPEComp: a New Benchmark Suite for Measuring Parallel Computing Performance". In *Proc of the International Workshop on OpenMP Applications and Tools: OpenMP shared Memory Parallel Programming*, 2001.
- [2] S.Y. Borkar. Platform 2015: Intel processor and platform evolution for the next decade. Technical report, Intel White Paper, Mar. 2005.
- [3] A. Bracy and A. Roth. "Serialization-Aware Mini-Graphs: Performance with Fewer Resources". *Proc. International Symposium on Microarchitecture*, 2006.
- [4] D. M. Brooks, V. Tiwari and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimization. In *International Symposium on Computer Architecture*, June 2000.
- [5] J. Donald and M.Martonosi. "Techniques for Multicore Thermal Management: Classification and New Exploration". *Proc. International Symposium on Computer Architecture*, 2006.
- [6] P. Dubey. Recognition, Mining and Synthesis Moves Computers to the Era of Tera. Technology@Intel Magazine. <http://www.intel.com/technology/magazine/computing/recognition-mining-synthesis-0205.htm>, 2005
- [7] M. Ekman, F. Dahlgren and P.Stenstrom. "Evaluation of Snoop-Energy Reduction Techniques for Chip-Multiprocessors". *Workshop on Duplicating, Deconstructing and Debunking*, 2002.
- [8] Frequent Itemset Mining Implementations Repository. <http://fimi.cs.helsinki.fi>
- [9] S. Gochman, R. Ronnen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Speerber, R.C. Valentine. "The Intel Pentium M Processor: Microarchitecture and Performance". *Intel Technology Journal vol 7(2)*, 2003.
- [10] R.A. Hankins, G.N. Chinya, J.D. Collins, P.W. Wang, R. Rackvic, H. Wang, J.P. Shen. "Multiple Instruction Stream Processor". *Proc. of the International Symposium on Computer Architecture*, 2006.
- [11] Intel Corp., Computer-Intensive Highly Parallel Applications and Uses. Intel Technology Journal, 9(2), May 2005.
- [12] C. Isci, A. Buyuktosunoglu, C-Y Cher, P. Bose and M. Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. *Proc. International Symposium on Microarchitecture*, 2006.
- [13] A. Jaleel, M. Mattina and B. Jacob. "Last Level Cache (LLC) performance of data-mining workloads on a CMP—A case study of parallel bioinformatics workloads". *Proc. International Symposium on high Performance Computing*, 2006.
- [14] P. Kongetira. K. Aingaran; K. Olukotun. "Niagara: A 32-Way Multithreaded Sparc Processor". *IEE Micro*, vol 25(2), pp. 21-29, 2005
- [15] D.J. Kuck. Platform 2015 software: Enabling innovation in parallelism for the next decade. Technical report, Intel White Paper, Mar. 2005.
- [16] J. Li, J. Martínez and M. Huang. "The Thrifty Barrier: Energy-Efficient Synchronization in Shared-Memory Multiprocessors". *Proc. International Symposium on High Performance Computer Architectures*, 2004.
- [17] D. Lopez, J. Llosa, M. Valero, and E. Ayguade. Widening Resources: A cost-Effective Technique for Aggressive ILP Architectures. *Proc. International Symposium on Microarchitecture*, 1998.
- [18] I. Martel, D. Ortega, E. Ayguadé and M. Valero. "Increasing Effective IPC by Exploiting Distant Parallelism". *Proc. International Conference on Supercomputing*, pp 348-355. 1999.
- [19] K. Olukotun, B.A. Nayfeh, L. Hammond, K.Wilson and K.-Y. Chang. "The Case for a Single Chip Multiprocessor". *Proc. International Conference on Architectural Support for Operating Systems*, 1996.
- [20] A. Pajuelo, A. Gonzalez and M. Valero. "Speculative Dynamic Vectorization". *Proc. International Symposium on Computer Architecture*, 2002.
- [21] J. Pisharath, Y. Liu, B. Ozisikilmaz, R. Narayanan, W. Liao, A. Choudhary, G. Memik.. NU-MineBench Project. <http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html>
- [22] Sun Microsystems. Throughput Computing. Technical report, Sun White Paper, Nov. 2005.
- [23] D. Tullsen, S. Eggers and H. Levy. "Simultaneous Multithreading: Maximizing On-Chip Parallelism". *Proc. International Symposium on Computer Architecture*, 1995.
- [24] R.Uhlig, R.Fishtein, O. Gershon, I. Hirsh, and H. Wang." SoftSDV: A Pre-silicon Software Development Environment for the IA-64 Architecture". Intel Technology Journal, Vol 3, Issue 4,1999.
- [25] S. Vajapeyam, P.J. Joseph and T. Mitra. "Dynamic Vectorization: A Mechanism for Exploiting Far-Flung ILP in Ordinary Programs". *Proc. International Symposium on Computer Architecture*, 1999.