Published in IET Computers & Digital Techniques
Received on 30th March 2009
Revised on 1st August 2009
doi: 10.1049/iet-cdt.2009.0040

ISSN 1751-8601

Energy efficiency via thread fusion and value reuse

R. Rakvic[1] *J. González*[2] *Q. Cai*[2] *P. Chaparro*[2]
G. Magklis[2] *A. González*[2]
[1]*United States Naval Academy, Annapolis, Maryland, USA*
[2]*UPC-Intel Lab Barcelona, Barcelona, Spain*
E-mail: rakvic@usna.edu

Abstract: Energy consumption has become the dominant metric when it comes to designing high-performance simultaneous multi-threaded (SMT) microprocessors. The authors propose a fusion of threads to help reduce energy consumption for these future SMT microprocessors. Threads are fused by merging two dynamic instances of the same static instruction into a single instruction thereby reducing unnecessary redundant computation in the front-end of the processor. The result is that power consumption is reduced in the pipeline until the execution stage. The authors have performed full system simulation, and our simulation results show average energy reduction of 10% with little impact on performance (less than 1%). They also extend thread fusion by proposing mechanisms to reduce the number of register file accesses and functional unit activity by reusing computations when the input values of the two dynamic instances of a fused instruction are the same. Our experiments show 5% energy savings in the integer register file and 10% in the integer functional units using this technique.

1 Introduction

Chip multiprocessors (CMPs) [1] have become the dominant approach to advance microprocessor performance. Sample recent microprocessors include the Niagara architecture [2], Sun's Throughput Computing [3] and Intel's Tera-scale computing [4]. These CMPs also include a great number of threads within a core. Design teams accomplish this by employing simple, small cores, such as in-order cores, capable of simultaneous multithreading (SMT) execution [5]. From an energy viewpoint, it is also a challenge to build dozens of cores on a single chip [6–8]. How to address the cost of cooling, the limitations on total and maximum power dissipation, and the existence of multiple hot spots [9, 10] in such architectures is still an open research topic.

These processors are built in order to efficiently execute parallel applications [11, 12]. Common types of parallel applications involve a parallel loop that terminates at a barrier that synchronises all parallel threads. In this study, we classify these parallel loops as either balanced or unbalanced [13, 14].

We have observed that when a parallel application is balanced, it often times executes the same code simultaneously. This results in a redundant amount of wasteful activity within the processor. We have devised a scheme, termed thread fusion, that takes advantage of this redundant behaviour in order to reduce total energy consumption in a two-way SMT in-order core [13]. We pull two dynamic instances of the same static instruction from two parallel threads and fuse them dynamically into a single instruction. In the front end of the microprocessor, this fused instruction acts as a single instruction. In the middle and back-end of the core, the fused instruction is cloned into two instructions which are executed in parallel. The result is that power consumption is reduced in the pipeline until the execution stage. We have performed full system simulation, and our simulation results show average energy reduction of 10% with little impact on performance (less than 1%).

This work also significantly extends this idea by observing that, sometimes, the source operands of the two instructions that form a fused instruction have the same value (e.g. load from a global pointer). This information can be used to further reduce activity. For example, we could read only once from the register file and bypass the value to both operations, or if both sources are the same, we could execute only once and replicate the result. For this purpose, we propose Value Reuse as an effective way to reduce activity even further. Value prediction and reuse has recently been introduced for different performance enhancements [15–20], but not for the purpose of energy reduction with thread fusion. Specifically, we propose to include a mechanism in the issue stage to detect identical inputs early. This information is used in later stages to save register file accesses and redundant functional unit computations. Our experiments show 5% energy savings in the integer register file and 10% in the integer functional units using this technique.

In Section 2, our baseline SMT in-order core is introduced. The motivation for our work is presented in Section 3. The background for thread fusion is explained in Section 4, including the pipeline changes required to implement it. Applying value reuse in addition to thread fusion is introduced in Section 5. Our full-system simulation and analysis is in Section 6. Related work is described in Section 7. A conclusion is provided in Section 8.

2 SMT microarchitecture

Future CMP processors with dozens or hundreds of cores must include a simple and efficient core. An in-order core utilising SMT to run multiple threads accomplishes these requirements. The main drawback of in-order cores 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: two threads that share the core resources. In particular, issue slots that cannot be used by one thread (e.g. due to a cache miss) are available to the other thread. This way high execution bandwidth can be maintained. Illustrated in Fig. 1*a* is a simple illustration of our pipeline for a two way threaded processor. 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 as well as a branch target buffer (BTB).

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

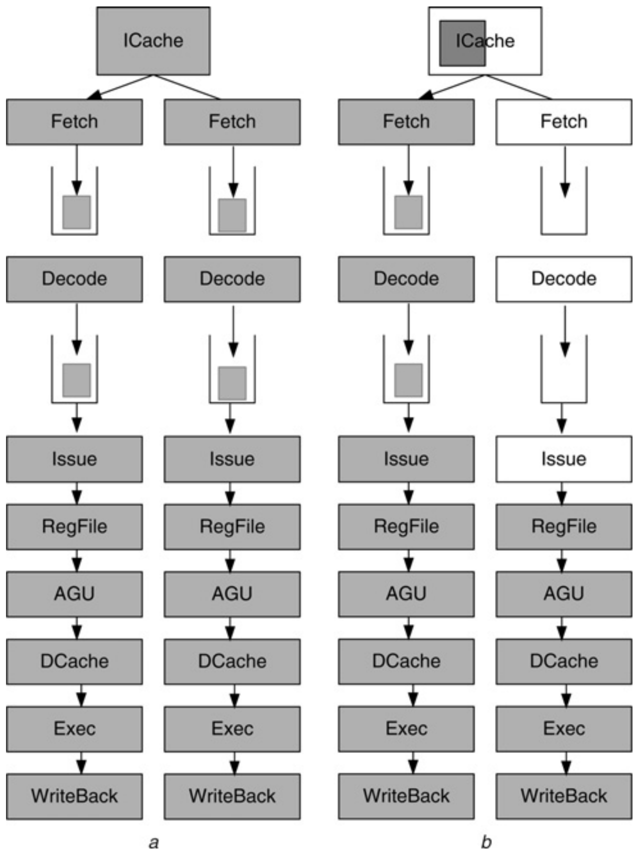


Figure 1 *Pipeline utilisation*

a Two threads in normal mode
b Two threads in fused mode
In gray, the blocks used in the execution of instructions in both cases

- *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 arrive 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 single instruction multiple data (SIMD) arithmetic instructions are executed in the respective functional units.

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

In order to support SMT, the pipeline also includes the following features:

- Two ports in the instruction and data caches, one per thread,

- Two instruction buffers in the fetch stage as well as two instruction queues, one per thread,

- One architectural register file and scoreboard table per thread.

We assume this core to be a 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 modified exclusive shared invalid (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 utilise a simple example to further motivate our work. Fig. 2 shows a parallel loop containing an innermost loop. Fig. 2*b* shows the 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. If the core manages to synchronise these threads and enter Thread Fused mode at the beginning of some iteration (at instruction 3), instructions 4 to 8 from both threads will be executed in lockstep mode. We will refer to the instruction that results from merging two identical instructions from different parallel threads as fused instruction.

For instance, 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, utilising the resources of a single instruction therefore using 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 utilise the same resources as two independent instructions.

Fig. 1 shows the utilisation of the pipeline when two parallel threads run together. Fig. 1*a* shows the occupancy of the different stages when two threads execute independently (normal execution in a SMT core). All the resources are available for both threads (recall that there is only one data cache but with two read ports). Fig. 1*b* shows the pipeline utilisation when the two threads are fused and they run in lockstep mode.

The 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. Theoretically, we should not expect any performance penalty with thread fusion since the processor bandwidth is not modified.

4 Thread fusion

This section describes how thread fusion works and the extensions needed in the in-order core. The proposed core has two execution modes:

Normal mode: Instructions are processed as usual; two threads running together utilise 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 utilise the instruction queue and the scoreboard of thread *A*.

```
#omp_parallel
for (i=0; i < limit; i++)
{
   p=q[i];
   while (p!=NULL)
   {
      b[i] = a[i] + p->elem;
      p = p->next;
    }    . . .
}
```

```
// r2 is i * word size
(1)     r3 = load  r2(r4);
(2)     r6 = load r2(r10)
while:
(3)     beq r3,0, outwhile
(4)     r5 = load 0(r3)
(5)     r11 = r5 + r6
(6)     r3 = load 4(r3)
(7)     jmp while
(8)     store r11 r2(r12)
(9)     jmp while:
```

Figure 2 *Example of a parallel loop*

a C code
b Assembly code

4.1 Triggering thread fusion

The core is initially in normal mode. A synchronisation point (SP) is used to switch to fused mode. A SP is the first program counter (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.

SPs can be inserted by the compiler via marking the instruction with a special hint, by the user with an Open MP (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.) SPs are stored in a small table located in the fetch (eight entries is enough to capture the SPs).

4.2 Executing fused threads

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

Fetch: After synchronising 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 as far as decoding is concerned nothing has 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 will likely 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 are likely different. Later we describe a technique to reduce energy consumption even further when fused instruction sources have the same value.

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 AGU 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 cycle. 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. If this occurs in normal mode, instructions depending on the offending load are flushed and the thread is stalled until the miss is resolved. Meanwhile, the other thread can proceed normally. This is one of the main benefits of SMT: if one thread is stalled the other can continue, utilising all available resources.

However, when the processor is in fused mode, fused instructions represent instructions for both threads, and fused instructions issue from a single queue. Therefore 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 by one cycle, 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 synchronisations (when one of the threads arrives to a SP and waits until the watchdog timer expires) constitute a source of performance loss. In order to reduce the number of failing synchronisations, a mechanism has been implemented to keep track of the combinations of PCs that constantly result to failing synchronisations.

Every time thread *A* fetches a SP, the current PC of thread *B* is stored in an additional column of the SP table, in the same entry as the SP. If the synchronisation 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 synchronisation is not initiated.

We have found experimentally that storing a 'potentially' offending PC along with each SP is enough to capture most failing synchronisations. 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 SP, the other thread is stalled on a cache miss.

In order to realise important energy savings, the processor must run in fused mode as much as possible. When the SPs are decided statically by either the user or the compiler, it may happen that they are not effective enough to obtain a high rate of fused instructions. Here we propose a mechanism to add new SPs 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 SP. 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 SP. When the saturating counter is greater than 1, the candidate PC becomes a SP. 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 eight SPs (either already established or being evaluated). Each entry will consist of the following fields:

- *PC–SP*: PC of the SP. 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 synchronisation 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 synchronisation misses.

5 Dynamic value reuse

In order to reduce energy even further, we propose to take advantage of computations that are the same in both threads by reusing the computation results or source operands. The core keeps track of the registers that have the same value in both architectural register files and avoids repeating activity related to those registers (e.g. do not use two register file ports to read the same value).

5.1 Detecting reusable values

First, the core must detect that a given architectural register is identical in both register files (thread *A* and thread *B*). We propose to add comparators at the output of each pair of identical functional units in order to detect whether a particular fused instruction produces the same value for both output registers. This may increase the power consumption as well as the latency of the functional units.

Regarding power, we believe that we can gain this power back with the savings from the reduction in the number of writes to the register file. If the two instructions represented by a fused instruction generate the same value, then the value is just stored in the register file of thread *A*, avoiding the write into the register file of thread *B*.

As far as complexity is concerned, the comparator can be integrated in the design of the functional units instead of being located just at the output. Therefore the comparison can be done in parallel with the operation itself and the total latency will not be impacted significantly.

5.2 Managing reusable values

For value reuse to work, the microarchitecture must track the values that are identical in both architectural register files. We propose to add an extra bit to each scoreboard entry to indicate whether the corresponding register has the same value for both threads. This bit will be set/reset by the comparators associated to each pair of functional units.

In fused mode, when a fused instruction is issued, it is cloned so that two identical instructions are executed simultaneously. If the comparator associated with the functional units indicates that the outputs match, the scoreboard is updated to indicate that both register files have the same value. In addition, this bit is forwarded along with the value to the different levels of the bypass to indicate to all possible fused instructions that depend on this register that the bypassed value is the same for both threads.

5.3 Benefiting from reusable values

Once the scoreboard has up-to-date information about values that are identical in both register files, the next step is to take advantage of this feature.

When a fused instruction is at the head of the instruction queue of thread *A*, it continuously checks the scoreboard table to see whether its inputs are available. At this time, it can verify whether any of the inputs are the same for both register files. Different situations arise at this point:

- *There is no match*: Both register files are accessed to read the input values corresponding to each thread and two functional units are used. If the result is the same for both instructions, the value is just written in the register file of thread *A*, and the extra bit of the scoreboard is updated properly.

- *One input matches*: The register file of thread *A* is accessed to read the reusable input, whereas both register files are accessed for the other input. The reusable value is written to the corresponding latches at the entry of the two functional units that will execute the two instructions. One register file access is therefore avoided. Since the microprocessor was already able to feed any ALU from any register file, we can reuse the value without changing the design. Fig. 3*a* shows the pipeline utilisation for this case.

- *Both inputs are identical (If a fused instruction with two sources src1 and src2 is cloned into two instructions, i1 and i2, then having both inputs identical means that src1 of i1 is the same as scr1 of i2 and src2 of i1 is the same as src2 of i2)*: In this case, both inputs are read from the register file of thread *A* and the values are written to the latches of the ALU associated to thread *A*. The instruction is executed just once. Since both inputs match, the result of the two executions will be the same; therefore, there is no need to perform the computation twice. The result of the execution is stored in the register file of thread *A*. If the fused instruction is a load or a store, then the effective address will be computed once but we will access the cache twice, even though the address is the same. This is to maintain the correctness of the coherence protocol. Fig. 3*b* shows the pipeline utilisation for this case. Note that the activity is significantly decreased when compared to Fig. 1*a*.

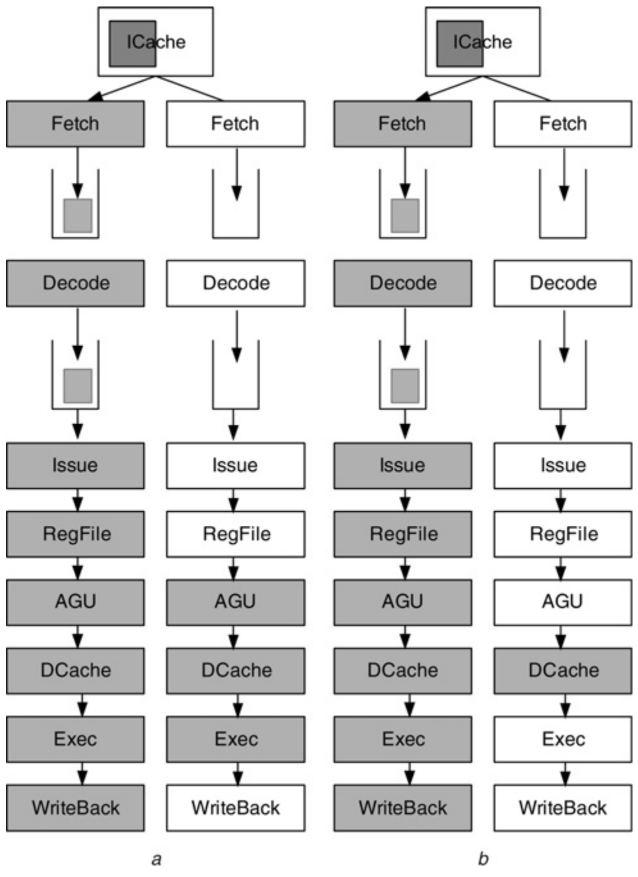


Figure 3 *Microprocessor utilisation*

a Utilisation if one input is the same as well as the output
b Utilisation when both inputs are the same

When the processor exits fused mode, the latest version of some registers may be stored only in the register file of thread *A*, because of value reuse. This means that in order to know the location of the latest version of each architectural register, the processor must read the extra bit of the scoreboard even when working in normal mode. However, the comparisons of the execution results will be disabled and the write back of new values will be performed to each register file independently.

6 Results

In this section, first, we present our experimental methodology and then we discuss the results we obtained for our two proposed techniques: thread fusion and thread fusion with value reuse.

6.1 Simulation methodology

Our experiments were carried out using an execution-driven simulator. The functional model of our simulator is using SoftSDV [21]. The simulator also incorporates a power model based on the activity factors and energy per access, similar to Wattch [22].

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

The model for our baseline is a very simple in-order two-way microarchitecture, similar to a realistic state-of-the-art processor [23], not a futuristic baseline. The focus of this idea is within a single core, but the results may easily be extrapolated across multiple cores. For instance, in a multiple core configuration, multiple benchmarks could be executing on different cores simultaneously, resulting in a sum of energy savings. Alternatively, an individual benchmark could be threaded across multiple cores, and the energy savings could be summed. One could also reasonably expect to see similar net energy savings on a complex out-of-order core. The front end of the out-of-order core is similar to ours, and the net energy savings should be similar. However, the percentage energy savings may differ based on the energy consumption of the back-end of the core.

Our two way SMT core 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. This means that 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.

6.2 Benchmarks

To evaluate our techniques, we use a combination of benchmarks from the SPEComp [13] and from the Recognition, Mining and Synthesis (RMS) Benchmark suite [24, 25]. Each benchmark is parallelised 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. For each program, the loop that represents the largest percentage of program execution is simulated. This loop is repeatedly simulated until one of the threads executes a minimum of 10 M instructions. Since these loops are extremely repetitive, more than 10 M instruction simulation is unnecessary and shown to be the same.

The benchmarks that we have chosen are balanced and loop based. From SPEComp, we selected equake, wupwise and swim. The simulated parallel sections 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 [26], 40% coverage), gauss (Gauss–Seidel iterative solver, coverage 99%), genenet (Bayesian network structure-learning problem, to measure regulatory relationships between genes [27], coverage 90%), kmeans (partition-based clustering algorithm for mining [28], coverage 99%) and modulenet (coverage 54%). As we will show, the benchmarks have a strong integer component. As we showed previously with fimi, there are integer calculations within these repetitive loops.

6.3 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 (value reuse is not exploited). For this experiment, we assume that SPs are set by the user (either the compiler or the programmer) at the beginning of loop iterations.

Fig. 4 shows the execution time and energy consumption of a processor with thread fusion enabled normalised 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 about 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) wrong synchronisations.

On the other hand, fimi runs 5% faster, consuming 18% less energy. The reason for this speedup is because of the potential increase in execution bandwidth. Two fused instructions can proceed through the remainder of the

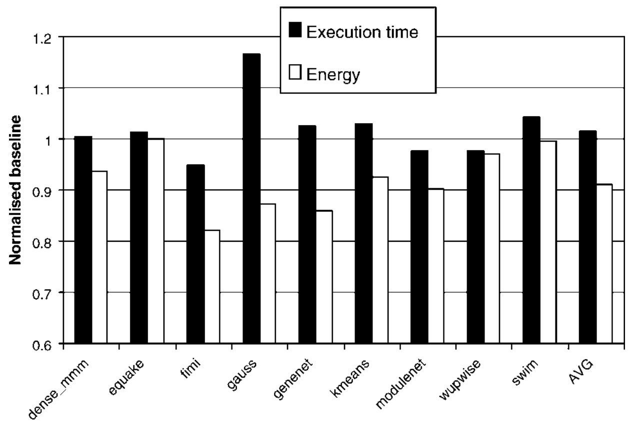


Figure 4 *Execution time and energy consumption of fused threads normalised to a system running two parallel threads*

pipeline if the resources support it. For example, in the RegFile stage, if the register file has enough ports (four read two write) to meet the current demand from two fused instructions, then both fused instructions may proceed to the next pipeline stage.

One fused branch and one fused Integer addition can be issued in parallel as long as each of the involved instructions requires only one read port. Furthermore, one fused FP instruction can be issued along with one fused Integer instruction. Therefore thread fusion may result in speedup, such as in fimi, since the effective issue bandwidth is increased under some situations.

Fig. 5*a* shows a breakdown of the distribution of the issue cycles among the following cases:

- *mmFT*: Cycles in fused mode that the issue was stalled due to a miss in the data cache (by both threads or any of the threads).

- *hhFT*: Cycles in fused mode where the issue could proceed since there was not any pending load miss.

- *mm*: Cycles that the processor is executing in normal mode but no instructions were issued since both threads have a pending miss.

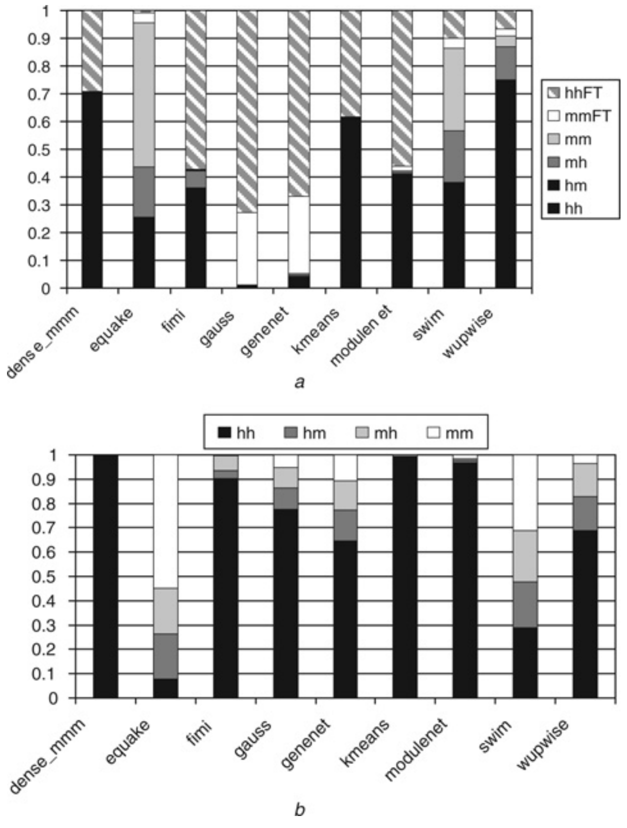


Figure 5 *Issue cycle distribution*

a Distribution of issue cycles for thread fusion
b Distribution of issue cycles for the baseline

- *hh*: Cycles executing in normal mode where there was no pending miss.

- *hm & mh*: Cycles executing in normal mode where one of the threads had a pending miss.

Fig. 5*b* shows the above distribution of issue cycles when the processor is running in normal mode, whereas Fig. 6 plots the percentage of cycles the processor is executing in fused mode, along with the percentage of synchronisations that were initiated but eventually failed.

Looking at Fig. 5*a*, we observe that gauss and genenet experience a high percentage of cache miss rate when running in fused mode (30%). In addition, Fig. 6 shows that the processor executes this application in fused mode almost all the time (close to 100% for gauss).

Looking at Fig. 5*b*, we observe that for gauss, in the baseline, when two threads are executed independently, the percentage of cycles where both threads are stalled is 5%, whereas 15% of the cycles 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 forming a fused load.

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

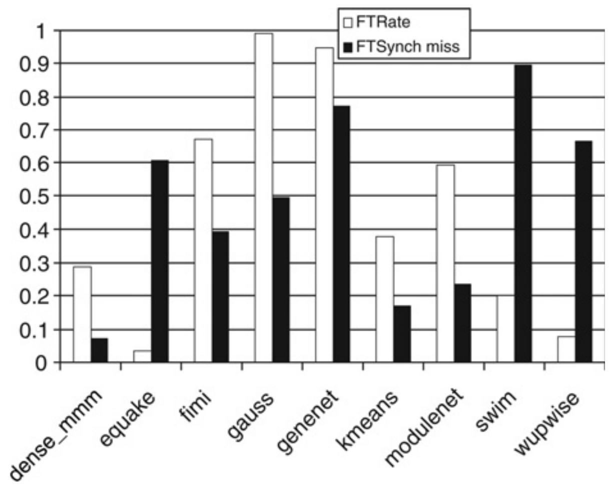


Figure 6 *Percentage of cycles the processor is running in fused mode (FT rate) and percentage of wrong synchronisations (FT synch miss)*

high percentage of synchronisations missed combined with a small number of total synchronisations.

6.4 Thread fusion and value reuse

This section describes the energy reduction achieved due to the re-utilisation of computation results at the presence of identical registers in both architectural register files.

Fig. 7 shows the distribution of instruction repeatability. In this figure, we can see the percentage of fused instructions whose components have the same inputs, the percentage of fused instructions whose corresponding instructions have only one identical input and the percentage of fused instructions with no identical inputs. We observe that applications like fimi, modulenet or wupwise show some potential for value reuse because 30% of fused instructions have the same inputs for both instructions. On average, 15% of fused instructions have all inputs identical and another 15% have just one. These results suggest that value reuse can provide additional benefits in energy savings that can compensate for its cost.

Once some repeatability has been demonstrated, we check potential energy savings. These savings come mainly from the reduction of the register file and functional unit activity.

Fig. 8*a* shows the breakdown of reads to the different register files. Only accesses to the register file are considered; values read from the different bypass levels are ignored. Fig. 8*b* shows the dynamic energy due to register file reads in a processor with thread fusion and value reuse enabled, normalised to a processor with thread fusion only. Recall that the scoreboard table includes a bit per register indicating whether that register has the same value in both register files. If the bit is enabled, only the register file from thread *A* is read.

For fimi and gauss we can observe a significant reduction in the number of Integer register file reads (20 and 10%), respectively. Surprisingly, wupwise does not demonstrate equal energy reduction despite its repeatability. Most of these results are read from the bypass. Finally, average

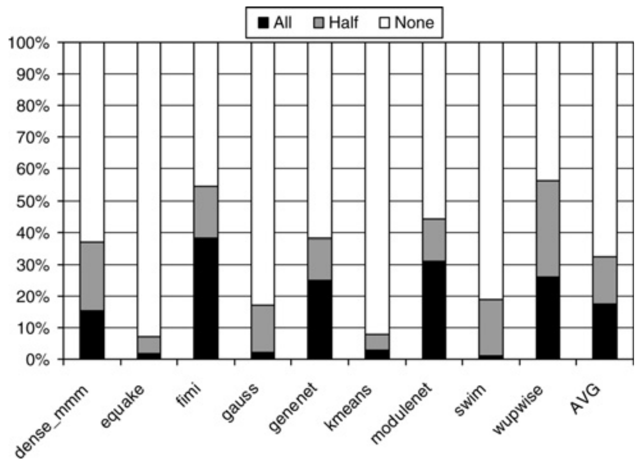
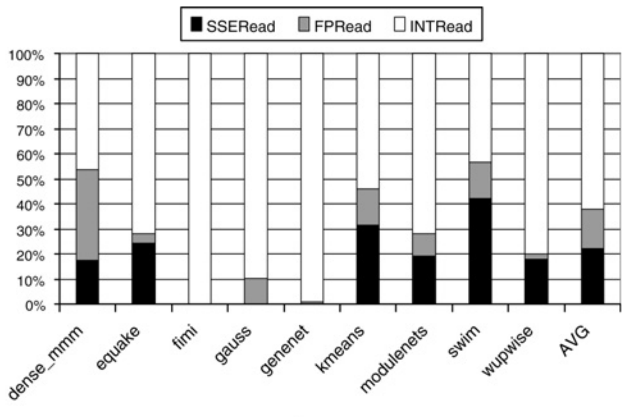
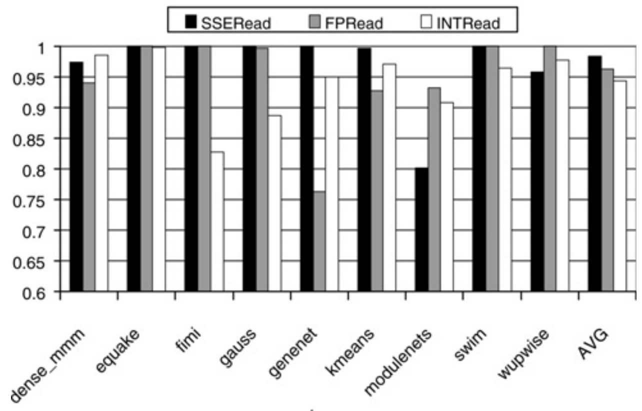


Figure 7 *Breakdown of instruction repeatability*



a



b

Figure 8 *Read access type and energy*

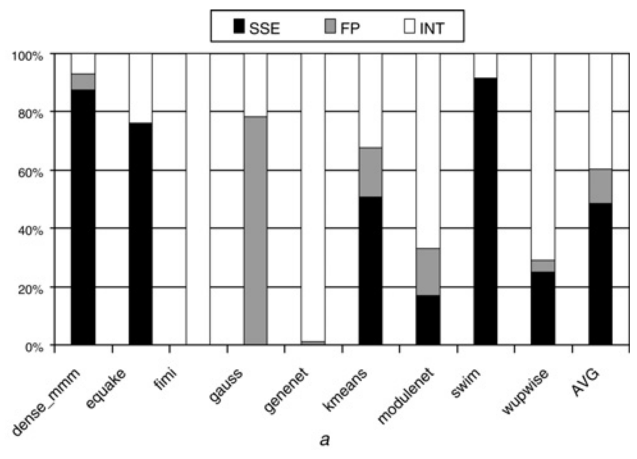
a Breakdown of read accesses to each type of register file
b Dynamic energy due to read accesses to the register file with value reuse, normalised with thread fusion without value reuse

energy savings for the different register files are 3% for streaming SIMD extension (SSE) and 5% for Integer.

Fig. 9*a* shows the breakdown of the computation in the different functional unit types for all the evaluated applications as well as the arithmetic mean. Fig. 9*b* shows the dynamic energy consumption of the functional units (by type) normalised to a processor with thread fusion but without value reuse. Remember that an operation in a functional unit is avoided when all of the corresponding inputs of the two instructions composing a fused instruction are identical.

Results are very promising for the Integer functional unit, since its dynamic energy dissipation is reduced by 12% on average. Genenet, fimi and modulenet are the programs that obtain the highest energy reduction. This is consistent with the numbers shown in Fig. 7. Energy reduction for FP and SSE is modest on average (2 and 5%), with the exception of modulenet which shows an energy reduction of up to 20% from reuse.

Finally, note that this reduction of the functional unit utilisation translates directly into a reduction of the



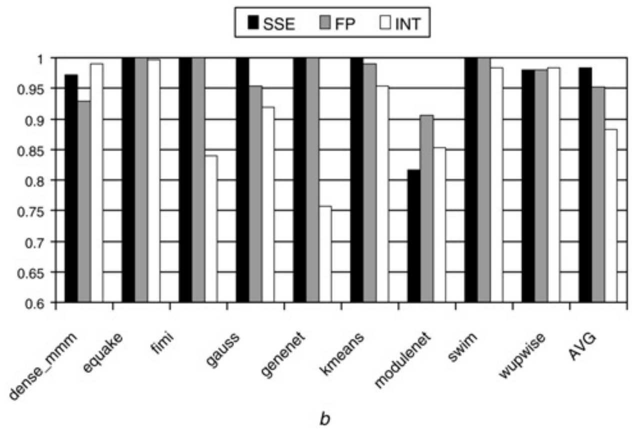


Figure 9 *Functional unit usage and power usage*

a Distribution of the computation for each functional unit type (SSE, floating point and integer)
b Dynamic power for each functional unit type normalised to thread fusion without value reuse

utilisation of the register file write ports: the fused instructions that execute in a single functional unit because of reusability write the result in only one of the architectural register files. As a result, the total number of register file writes is reduced by 20%, which compensates for the power dissipated by the comparators added to the functional units to detect reuse.

7 Related work

There has been previous work on power and thermal aware multicore systems ([6, 9, 29–34] among others). These previous studies focus both on multiprogramming workloads and on parallel applications and some of them attempt to adapt the system behaviour to the application characteristics. Li and Martínez [32] try to find the best dynamic configuration regarding the number of active cores as well as their frequency in order to meet a particular power budget. Other papers have focused on evaluating the power/complexity of the interconnect, since it may become a major obstacle to multi-core performance [35, 36].

The thrifty barrier [37] is an interesting proposal targeted to heterogeneous applications. Threads that arrive early to the barrier put their cores to low-power states. Dynamic vectorisation techniques [38, 39] attempt to increase the ILP exploited by a processor by, for instance, overlapping the execution of very regular loops with the loop continuation [39]. Replication and widening [40] is another proposal to increase the available instruction level parallelism (ILP) that can be exploited on very long instruction word (VLIW) machines.

Micro-op Fusion [41] and instruction collapsing [42] are techniques that merge instructions to either reduce energy consumption or increase bandwidth. In Micro-op Fusion, two consecutive micro-ops from the same thread that meet some requirements are merged into a single micro-op in order to save energy.

The main differences between our work and previous works are twofold. First, we focus on reducing power when two threads from a parallel application are running on a SMT core. The target is homogeneous parallel sections, that is, when the threads reach the synchronisation at about the same time. 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 vectorisation and the widening works [38–40], our proposal does not focus on improving ILP but on reducing power with minimal impact on performance. Those proposals attempt to speedup single thread execution by widening or vectorising regular code using special purpose units. Our proposal attempts to reduce energy consumption when a SMT core is running two parallel and homogeneous threads by means of using less resources and reusing activity.

Thread fusion can be implemented along with micro-op fusion or collapsing, since there orthogonal and synergistic effects can be obtained. Once the processor enters the fused mode and fused instructions are fetched, micro-op fusion and/or collapsing can be applied to further reduce energy or increase bandwidth.

Finally, value prediction and reuse has recently been introduced for different performance enhancements [15–20], but not for the purpose of energy reduction with thread fusion. Specifically, we propose to include a mechanism in the issue stage to detect identical inputs early. This information is used in later stages to save register file accesses and redundant functional unit computations.

8 Conclusions

To take advantage of balanced loop code, threads are fused by merging two dynamic instances of the same static instruction into a single instruction thereby reducing unnecessary redundant computation in the front-end of the processor. Power consumption is hence reduced in the front end of the pipeline until the execution stage. Our simulation results show average energy reduction of 10% with little impact on performance (less than 1%). Additionally, we have proposed value reuse as an effective way to reduce activity even further. Specifically, we propose to include a mechanism in the issue stage to detect identical inputs early. This information is used in later stages to save register file accesses and redundant functional unit computations. Our experiments show 5% energy savings in the integer register file and 10% in the integer functional units using this technique.

9 References

[1] OLUKOTUN K., NAYFEH B.A., HAMMOND L., WILSON K., CHANG K.-Y.: 'The case for a single chip multiprocessor'. Proc. Int. Conf. on Architectural Support for Operating Systems, 1996

[2] KONGETIRA P., AINGARAN K., OLUKOTUN K.: 'Niagara: a 32-way multithreaded Sparc processor', *IEE Micro.*, 2005, **25**, (2), pp. 21–29

[3] Sun Microsystems: 'Throughput computing'. Technical report, Sun White Paper, November 2005

[4] BORKAR S.Y.: 'Platform 2015: Intel processor and platform evolution for the next decade'. Technical report, Intel White Paper Report, March 2005

[5] TULLSEN D., EGGERS S., LEVY H.: 'Simultaneous multithreading: maximizing on-chip parallelism'. Proc. Int. Symp. on Computer Architecture, 1995

[6] ISCI C., BUYUKTOSUNOGLU A., CHER C.-Y., BOSE P., MARTONOSI M.: 'An analysis of efficient multi-core global power management policies: maximizing performance for a given power budget'. Proc. Int. Symp. on Microarchitecture, 2006

[7] POLETTI F, POGGIALI A., BERTOZZI D., *ET AL.*: 'Energy-efficient multiprocessor systems-on-chip for embedded computing: exploring programming models and their architectural support', *IEEE Trans. Comput.*, 2007, **56**, (5), pp. 606–621

[8] MARCHAL P., GOMEZ J.I., ATIENZA D., MAMAGKAKIS S., CATTHOOR F.: 'Power aware data and memory management for dynamic applications', *IEE Proc. Comput. Dig. Tech.*, 2005, **152**, (2), pp. 224–238

[9] DONALD J., MARTONOSI M.: 'Techniques for multicore thermal management: classification and new exploration'. Proc. Int. Symp. on Computer Architecture, 2006

[10] NOURANI M., CHIN J.: 'Test scheduling with power-time tradeoff and hot-spot avoidance using MILP', *IEE Proc. Comput. Digital Tech.*, 2004, **151**, (5), pp. 341–355

[11] Intel Corp: 'Computer-intensive highly parallel applications and uses', *Intel Technol. J.*, 2005, **9**, (2)

[12] KUCK D.J.: 'Platform 2015 software: enabling innovation in parallelism for the next decade'. Technical report, Intel White Paper Report, March 2005

[13] GONZALEZ P., CAI Q., CHAPARRO P., MAGKLIS G., RAKVIC R., GONZÁLEZ A.: 'Thread fusion'. Int. Symp. on Low Power Electronics and Designs (ISLPED), 2008

[14] MARTEL I., ORTEGA D., AYGUADÉ E., VALERO M.: 'Increasing effective IPC by exploiting distant parallelism'. Proc. Int. Conf. on Supercomputing, 1999, pp. 348–355

[15] VINTAN L.N., FLOREA A., GELLERT A.: 'Focalising dynamic value prediction to CPU's context', *IEE Proc. Comput. Digit. Tech.*, 2005, **152**, p. 473

[16] GELLERTA A., FLOREA A., VINTANA L.: 'Exploiting selective instruction reuse and value prediction in a superscalar architecture', *J. Syst. Archit.*, 2009, **55**, (3), pp. 188–195

[17] CHANG S.C., LI W.Y.H., KUO Y.J., CHUNG C.P.: 'Early load: hiding load latency in deep pipeline processor'. Proc. Asia-Pacific Computer Systems Architecture Conf. (ACSAC), Taiwan, August 2008

[18] GOLANDER A., WEISS S.: 'Reexecution and selective reuse in checkpoint processors', *HiPEAC J.*, **2**, (3), pp. 242–268

[19] LIAO C.-H., SHIEH J.-J.: 'Exploiting speculative value reuse using value prediction'. Proc. Seventh Asia-Pacific Conf. on Computer Systems Architecture, Melbourne, Victoria, Australia, 1 January 2002, pp. 101–108

[20] SODANI A., SOHI G.S.: 'Dynamic instruction reuse'. Proc. 24th Ann. Int. Symp. on Computer Architecture, Denver, Colorado, USA, 1–4 June 1997, pp. 194–205

[21] UHLIG R., FISHTEIN R., GERSHON O., HIRSH I., WANG H.: 'SoftSDV: a pre-silicon software development environment for the IA-64 architecture', *Intel Technol. J.*, 1999, **3**, (4)

[22] BROOKS D.M., TIWARI V., MARTONOSI M.: 'Wattch: a framework for architectural-level power analysis and optimization'. Int. Symp. Comput. Architecture, June 2000

[23] Intel Corporation: Available at: http://www.intel.com/products/processor/atom/index.htm, 30 July 2009

[24] DUBEY P.: 'Recognition, mining and synthesis moves computers to the era of tera'. Technology@Intel Magazine, available at: http://www.intel.com/technology/magazine/computing/recognition-mining-synthesis-0205.htm, 2005

[25] HANKINS R.A., CHINYA G.N., COLLINS J.D., *ET AL.*: 'Multiple instruction stream processor'. Proc. Int. Symp. on Computer Architecture, 2006

doi: 10.1049/iet-cdt.2009.0040

[26] Frequent Itemset Mining Implementations Repository: Available at: http://fimi.cs.helsinki.fi

[27] JALEEL A., MATTINA M., JACOB B.: 'Last level cache (LLC) performance of data-mining workloads on a CMP—a case study of parallel bioinformatics workloads'. Proc. Int. Symp. on High Performance Computing, 2006

[28] PISHARATH J., LIU Y., OZISIKYILMAZ B. ET AL.: 'NU-MineBench project'. Available at: http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html

[29] DAVIS J.D., LAUDON J., OLUKOTUN K.: 'Maximizing CMP throughput with mediocre cores'. Proc. Int. Conf. on Parallel Architectures and Compilation Techniques, 2005

[30] HUH J., BURGER D., KECKLER S.: 'Exploring the design space of future CMPSs'. Proc. Int. Symp. on Parallel Architectures and Compilation Techniques, 2001

[31] KUMAR R., JOUPPI N.P., TULLSEN D.M.: 'Cojoined-core chip multiprocessing'. Proc. Int. Symp. on Microarchitecture, 2004

[32] LI J., MARTÍNEZ J.F.: 'Dynamic power performance adaptation of parallel computation on chip multiprocessors'. Int. Symp. on High Performance Computer Architectures, 2006

[33] GONZALEZ R., HOROWITZ M.: 'Energy dissipation in general purpose microprocessors', *IEEE J. Solid State Circuits*, 1996, **31**, (9), pp. 1277–1284

[34] FERRI C., BAHAR R.I., LOGHI M., PONCINO M.: 'Energy-optimal synchronization primitives for single-chip multi-processors'. Proc. 19th ACM Great Lakes Symp. VLSI, Boston, MA, USA, 10–12 May 2009

[35] EKMAN M., DAHLGREN F., STENSTROM P.: 'Evaluation of snoop-energy reduction techniques for chip-multiprocessors'. Workshop on Duplicating, Deconstructing and Debunking, 2002

[36] KUMAR R., ZYUBAN V., TULLSEN D.M.: 'Interconnections in multi-core architectures: understanding mechanism, overheads and scaling'. Proc. Int. Symp. on Computer Architecture, 2005

[37] LI J., MARTÍNEZ J., HUANG M.: 'The thrifty barrier: energy-efficient synchronization in shared-memory multiprocessors'. Proc. Int. Symp. on High Performance Computer Architectures, 2004

[38] PAJUELO A., GONZALEZ A., VALERO M.: 'Speculative dynamic vectorization'. Proc. Int. Symp. on Computer Architecture, 2002

[39] VAJAPEYAM S., JOSEPH P.J., MITRA T.: 'Dynamic vectorization: a mechanism for exploiting far-flung ILP in ordinary programs'. Proc. Int. Symp. on Computer Architecture, 1999

[40] LOPEZ D., LLOSA J., VALERO M., AYGUADE E.: 'Widening resources: a cost-effective technique for aggressive ILP architectures'. Proc. Int. Symp. on Microarchitecture, 1998

[41] GOCHMAN S., RONNEN R., ANATI I., *ET AL*.: 'The Intel Pentium M processor: microarchitecture and performance', *Intel Technol. J.*, 2003, **7**, (2)

[42] BRACY A., ROTH A.: 'Serialization-aware mini-graphs: performance with fewer resources'. Proc. Int. Symp. on Microarchitecture, 2006