Distributed Loop Controller Architecture for Multi-threading in Uni-threaded VLIW Processors

Praveen Raghavan *†, Andy Lambrechts *†, Murali Jayapala *,
Francky Catthoor *† and Diederik Verkest *†‡
* IMEC vzw, Kapeldreef 75, 3001 Leuven, Belgium
† KULeuven, Belgium ‡ VUB, Belgium
{ragha, lambreca, jayapala}@imec.be

Abstract

Reduced energy consumption is one of the most important design goals for embedded application domains like wireless, multimedia and biomedical. Instruction memory hierarchy has been proven to be one of the most power hungry parts of the system. This paper introduces an architectural enhancement for the instruction memory to reduce energy and improve performance. The proposed distributed instruction memory organization requires minimal hardware overhead and allows execution of multiple loops in parallel in a uni-processor system. This architecture enhancement can reduce the energy consumed in the instruction and data memory hierarchy by 70.01% and improve the performance by 32.89% compared to enhanced SMT based architectures.

1. Introduction

Modern embedded applications require sustained operation for long periods of time with no or minimal recharging of the battery. In some cases, like sensor-networks and in-vivo biomedical implants, battery-less operation may be preferred, where power is obtained by scavenging energy sources. In order to achieve such low power constraints it is crucial that the energy consumption is reduced in all parts of the system.

Therefore, the designer has to look at the complete system and tackle the power problem in each part. This paper presents a novel architectural enhancement that reduces the power consumed in the instruction memory hierarchy, which is one of the highest power consuming parts of the system [16]. The instruction memory energy bottleneck becomes more apparent after techniques like loop transformations, software controlled caches, data layout optimizations in [3, 14] have been applied to other components of the system.

State of the art architecture enhancements to reduce the energy consumed in the instruction memory hierarchy for VLIW processors include using loop buffers [13], NOP compression [12], SILO cache [5], code-size reduction [12], etc. In spite of these enhancements, the datapath and instruction memory organizations are centralized and have low energy efficiency [13]. Hence there is a need for a distributed and scalable solution. The well known *L0 buffer* or *loop buffer* is an extra level of memory hierarchy that is used to store instructions corresponding to loops. It is a good candidate for a distributed solution as shown in [13]. But current distributed loop buffers support only one thread of control.

To improve both performance as well as energy efficiency it is crucial to boost the parallelism [8]. Since loops form the most important part of a program, techniques like loop fusion and other loop transformations are applied to exploit the parallelism (boosting ILP) within loops on a single threaded architecture. However not all loops can be efficiently exploited in this manner (explained in Section 2). Therefore there is a need for a multi-threaded platform, that can support execution of multiple loops, with *minimal hardware overhead*.

This paper proposes a multi-threaded distributed instruction memory hierarchy that can support execution of multiple incompatible loops (see Figure 1) in parallel. In addition to regular loops, irregular loops with conditional constructs and nested loops can also be mapped. Sub-routines and function calls within the loops must be selectively inlined or optimized using other loop transformations like code hoisting or loop splitting, to fit in the loop buffers. Alternatively, sub-routines could be executed from level-1 cache if they do not fit in the loop buffers. A generic schematic of the proposed architecture is shown in Figure 2(c). The loop buffers are clustered, each loop buffer has its own local controller, and each local controller is responsible for indexing and regulating accesses to its loop buffer. The novelties of the proposed architecture enhancement are as follows:

3-9810801-0-6/DATE06 © 2006 EDAA

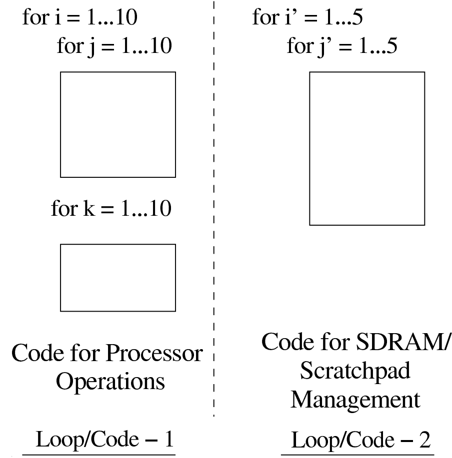


Figure 1. Incompatible Loop Organizations: a simple example

1. An energy-efficient and scalable, distributed controller organization
2. Multi-threaded incompatible loop operation in uni-threaded processors is enabled and
3. Overall energy savings are obtained along with enhancement in performance.

This paper is organized as follows: Section 2 presents a survey of state of the art architectural features that reduce the power consumption in a system and their disadvantages and also motivates the need to solve this problem. Section 3 introduces the proposed architectural enhancements and the corresponding techniques. Section 4 illustrates the software/compiler support required for this architecture. Section 5 presents the experimental setup for the proposed architecture and the results. Finally, Section 6 concludes this paper.

2. Motivating Example and Related Work

The example code shown in Figure 1 shows two loops with different loop organizations. In the context of embedded systems with software controlled memory hierarchy, the above code structure is realistic. Code 1 gives the loop structure for the code that would be executed on the data path of the processor. Code 2 gives the loop structure for the code that is required for data management in the data memory hierarchy. This may represent the code that fetches data from the external SDRAM and places them on the scratchpad memory or other memory transfer related code. Code 1 can be assumed to execute some operations on the data that was obtained by Code 2. The above code example can be mapped on different platforms. The advantages and disadvantages of mapping such a code on state of the art techniques/systems are described below.

2.1. Architecture Issues

The L0 buffer or loop buffer architecture is a commonly used technique to reduce instruction memory hierarchy energy [6, 13]. This technique proposes an extra level of instruction memory hierarchy which can be used to store loops. Additionally, several compiler transformations are proposed to improve loop buffering [19, 20]. Such state of the art L0 organizations like the ones shown in Figure 2(b) allow only single-threaded operation. Although the loop buffers are distributed, they contain a single loop controller and therefore such an organization does not support multi-threaded operation.

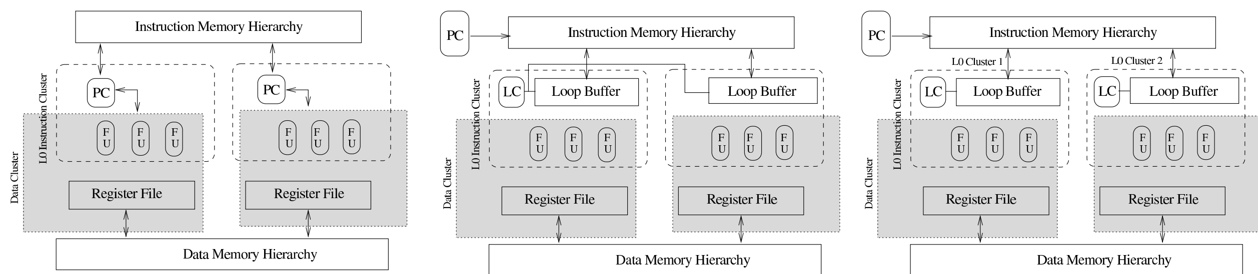
In uni-processor platforms (Figure 2(b)), loop fusion is a commonly used technique to execute multiple threads in parallel. Here, the candidate loops with different threads of control are merged into a single loop, with single thread of control. However these techniques cannot handle incompatible loops like the one shown in Figure 1. Even if advanced loop morphing [11] is applied the extra overhead due to *if-then-else* constructs and other control statements required can be very large, resulting in loss of both energy and performance (see Section 5 for comparison).

Multi-threaded architectures and Simultaneous Multi-Threaded (SMT) processors [10, 15, 21], can also execute multiple loops in parallel. However, these architectures are intended for larger granularity tasks than loops. Hence, the overhead of context management and switching is large. The data sharing in these architectures between two processes/threads is done at the cache level, which requires extra reads and writes from/to the memory and register file. SMT processors (shown in Figure 2(a)) need multiple fetch/decode units and complete program counter logic for each of the threads, which requires extra hardware overhead (explained in Section 5).

In the proposed architecture enhancement (refer Figure 2(c), detailed in Section 3), multiple loops can be executed in parallel, without the overhead/limitations mentioned above. Multiple synchronizable Loop Controllers (LCs) enable the execution of multiple loops in parallel as each loop has its own loop controller. However, the LC logic is simplified and the hardware overhead is minimal as it has to execute only loop code. Data sharing and synchronization is done at the register file level and therefore context switching and management costs are eliminated. (detailed in Section 4)

2.2. Technological Issues

In addition to the motivation mentioned above, there exists a need for non-shared distributed resources. It is often the case in embedded systems, that the same processor



(a) SMT Based VLIW Processor (b) VLIW processor with Single Loop Controller (c) VLIW processor with Distributed Loop Controller

Figure 2. Different Processor Architectures supporting Multi-threading

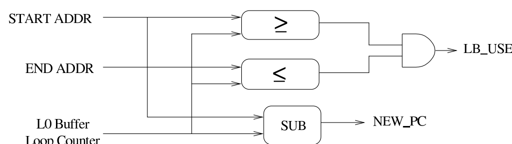


Figure 3. L0 Controller for every Cluster

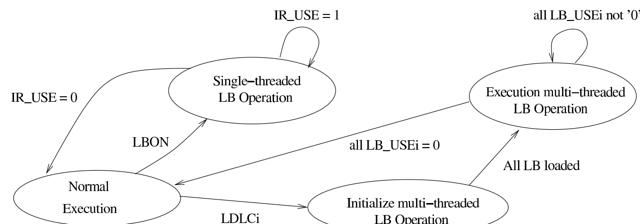


Figure 4. A state diagram illustrating the switching between single and multi-threaded mode of operation

needs to run different processes with different characteristics. Recently there has been a strong academic as well as industrial trend towards application-specific units to reduce the energy consumed for performing a specific task. Each distributed instruction cluster of the VLIW can be considered as an application specific cluster. A distributed instruction cluster processor with its own loop buffer and minimized resource sharing [9], reduces the extra energy cost due to the routing and interconnect requirement considerably as it can be placed physically closer to its cluster.

It has been shown in [7] that local interconnect is one of the growing problems for energy-aware design. It is therefore crucial that the instruction memories for different clusters of the VLIW are closer to their execution units. A distributed L0 loop buffer configuration for each VLIW cluster with separate loop controllers as shown in Figure 2(c), can significantly reduce the energy consumed in the local wiring.

3. Proposed Architecture Enhancement

This section presents the details of the proposed architectural enhancement that saves energy consumption and improves performance by enabling a synchronized multi-threaded operation in a uni-processor platform.

3.1. Extending a Uni-processor to Support Execution of Multiple Threads

We propose to extend a uni-processor model to support two modes of loop buffer operation: Single-threaded and Multi-threaded. The extension to multi-threaded mode is done with special concern to support L0 buffer operation. A VLIW instruction is divided into bundles, where each bundle corresponds to an L0 cluster. An L0 controller (illustrated in Figure 3) along with a counter (e.g. 5 bits) is responsible for indexing and regulating accesses to the L0 buffer. Unlike conventional Program Counters (PCs), the controller logic is much smaller and consumes lower energy, with the loss in flexibility that only loops can be executed from the loop buffers. The *LB_USE* signal indicates execution of an instruction inside the L0 buffer. The *NEW_PC* signal is used to index into the L0 buffer.

The L0 controllers can be operated in single/multi-threaded mode. The state diagram of the L0 Buffer operation is shown in Figure 4. The single threaded loop buffer operation is initiated on encountering the *LBON* *<addr>* *<offset>* instruction. Here *<addr>* denotes the start address of the loop's first instruction and *<offset>* denotes the number of instructions to be fetched to the loop buffer starting from address *<addr>*. In the single threaded mode, the loop counter of each cluster is incremented in lock-step every cycle. This mode of operation is similar to the L0 buffer operation presented in [13], but in the proposed approach an entire cluster can be made inactive for a given loop nest to save energy.

In the multi-threaded mode, the loop counters are still in lock-step, but not necessarily at every cycle. Instead they

synchronize at loop boundaries or explicit synchronization points identified by the compiler (explained in Section 4). To spawn execution of multiple loops that have to be executed in parallel, each L0 cluster is provided with a separate instruction (*LDLCi <addr> <offset>*) to explicitly load different loops into the corresponding L0 clusters. Here *i* denotes the cluster number. For instance, in the following example two instructions LDLC1 <addr1> <offset1> and LDLC2 <addr2> <offset2> are inserted in the code to indicate that the loop at *addr1* is to be executed in cluster 1 and the loop at the *addr2* is to be executed in cluster 2.

```
                —
         LDLC1 <addr1> <offset1>
         LDLC2 <addr2> <offset2>
addr1:           for (...){
                 Loop Body }
addr2:           for (...){
                 Loop Body}
                —
```

Once the instruction *LDLCi* is encountered, the processor operates in the multi-threading mode. During the initialization phase all the active loop buffers are loaded with the code that they will be running. For example, the ith loop buffer will be loaded with *offseti* number of instructions starting from address *addri* specified in instruction *LDLCi*. Meanwhile, each cluster's loop controller copies the needed instructions from the instruction memory into the corresponding loop buffer. If not all the clusters are used for executing multiple loops, then explicit instructions are inserted by the compiler to disable them.

When a cluster has completed fetching a set of instructions from its corresponding address, the loop buffer enters the execution stage of the *Multi-threaded* execution operation. During the execution stage, each loop execution is independent of the others. Although the loop counters are not in lock-step, the different loop buffers are synchronized at specific synchronization points (where dependencies were not met) that are identified by the compiler. Additionally, the compiler or the programmer must ensure the data consistency or the necessary data transfers across the data clusters.

4. Software/Compiler Support

The code generation for the proposed architecture is similar to the code generated for a conventional VLIW processor, except for the parts of the code that need to be executed in multi-threaded mode. As mentioned in the previous section, additional instructions are inserted to initiate the multi-threaded mode of operation.

Figure 5 shows the assembly code for the two incompatible loops presented in Figure 1. Code 1 is loaded to L0 Cluster 1 and Code 2 is loaded to L0 Cluster 2. If, for

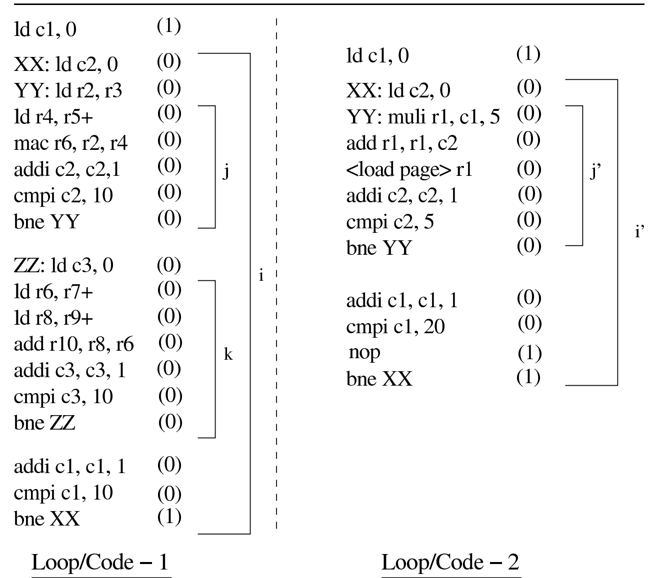


Figure 5. Assembly code for the code shown in Figure 1 (extra synchronization bits are shown in brackets)

two iterations of loop *i*, only one iteration of loop *i'* has to be executed, then there is a need to identify this dependency and need to insert necessary synchronization points to respect this dependency. The compiler needs to extract and analyze data dependencies between these two loops. For this purpose, the two loops shown in Figure 1 are first represented in a polyhedral model [18]. Once the different codes are represented in a common iteration domain [18], a data dependency analysis can be done [11]. On analyzing the data dependencies between different codes, the synchronization points can be derived. The synchronization points are then annotated back on the original code shown in Figure 5 within brackets.

Synchronization between the two clusters is achieved by adding an extra bit to every instruction. These extra bits are shown in Figure 5. A '0' means that the instruction can be executed independently of the other cluster and a '1' means that the instruction can only be executed if the other cluster issues a '1' as well. The handshaking/instruction level synchronization can be implemented in multiple ways. For example, instruction *ld c1, 0* of both the clusters would be issued simultaneously. The number of bits required for synchronization is one less than the number of clusters. If necessary extra *nop* instructions may be inserted to obtain correct synchronization. This instruction level synchronization reduces the number of accesses to the instruction memory and hence is energy-efficient. Due to limited space, instruction level synchronization is not discussed here. Further details of the analysis performed by the compiler are beyond the scope of this paper.

It can be seen from the assembly code in Figure 5, that using the synchronization bits the data sharing can be done at the register level instead of the cache level like in the case of SMT processors. This reduces the number of reads and writes to the memory and register file and further saving energy.

5. Results

Section 5.1 presents the experimental setup that was used to demonstrate this work. Section 5.2 presents the energy savings that were obtained on the proposed architecture and the reasons for the gains.

5.1. Experimental Platform Setup

The experiments were performed on the CRISP [17] simulator which is built on the Trimaran [2] VLIW frame-work. The simulator was annotated with power models for different parts of the system. The power models for the different parts of the processor where obtained using Synopsys *Physical Compiler* and *DesignWare components*, *UMC130nm* technology, *1.2*V Vdd. The power was computed after complete layout was performed and were back-annotated with activity reported by simulation using *ModelSim*. The complete system was clocked at *500*MHz (roughly the clock frequency of most embedded systems). The extra energy consumed due to the synchronization hardware was also estimated using *Physical Compiler* after layout, capacitance extraction and back-annotation. The memory models were obtained from [4].

Special instructions were inserted to enable multi-threaded operation on the VLIW. The experiments were performed on a VLIW with four slots. All slots were considered to be homogeneous and form one data cluster i.e. all four slots share the same global register file. Two slots are grouped into one L0 instruction cluster. Hence the VLIW processor has one common data cluster and two L0 instruction clusters.

5.2. Energy and Performance Analysis

For the benchmarks [1] used, which is a representative set for embedded systems domain. The output of the first benchmark was assumed to be the input to the second benchmark. This was done to create an artificial dependency between the two threads. Figure 6 and 7 respectively, show the energy savings and performance gains that can be obtained when multiple kernels are run on different L0 instruction clusters of the VLIW processor with the multi-threading extension presented in Section 3.1. The energy savings are considered for the memories (instruction and data) of the processor as they are the dominant part of any SoC [16].

In the *Sequential case* (Baseline case), two different codes were executed on the VLIW one after the other. The VLIW has a centralized loop buffer organization. In the

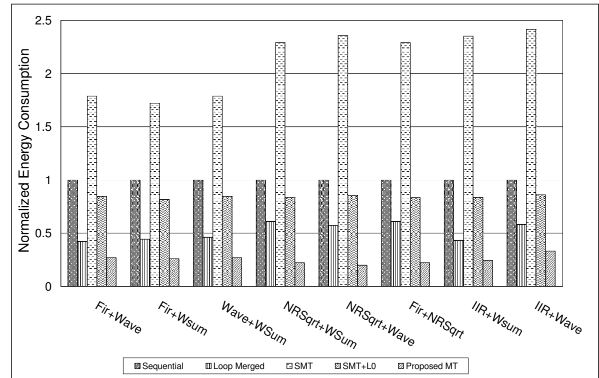


Figure 6. Energy Savings normalized to Sequential Execution

loop merged case, a variant loop fusion technique proposed in [11] was applied and executed on the VLIW with a centralized loop buffer organization and with a central loop controller. For the *SMT* case, a complete program counter and instruction memory of 32KB was used. The SMT was also enhanced with an energy efficient loop buffer as well. Although, SMT and Loop buffer technique are orthogonal, for our comparison to be fair we apply the loop buffering technique to the SMT architecture (*SMT+L0*). The *proposed multi-threading (MT)* architecture has a 5-bit loop counter for each cluster and some basic logic (as shown in Figure 3), whereas in case of SMT, the PC logic involved is more complex and would therefore consume more energy. All the results were normalized with respect to the sequential execution. Also compiler optimizations like software pipelining, loop unrolling etc. were applied in all the different cases.

The *Loop-Merged*(Morphed) technique saves over the *Sequential* technique since extra memory accesses are not required and data sharing is performed at register file level. This also explains the gains in performance compared to the sequential case. For instance, in case of the *FIR+WSum* benchmark, the number of accesses to the data memory (energy savings of *55.55 %*) and the number of cycles spent (performance gain of *25.00 %*) in accessing the level 1 data memories were avoided.

The *SMT+L0* technique reduces the energy further since both the tasks are performed simultaneously but the data sharing is still at the cache-level. In case of the *FIR+WSum* benchmark run on the SMT processor, the number of accesses to the loop buffer reduces (energy gain of *18.85 %*), but the accesses to the data memory DL1 still remain. This may be avoided by using a cache-coherency protocol. However such techniques require hardware overhead, which also consumes extra energy. Since the tasks can be done in parallel, there are gains in peroformance as well. The energy savings is lower in case of the last 5 benchmarks (contain-

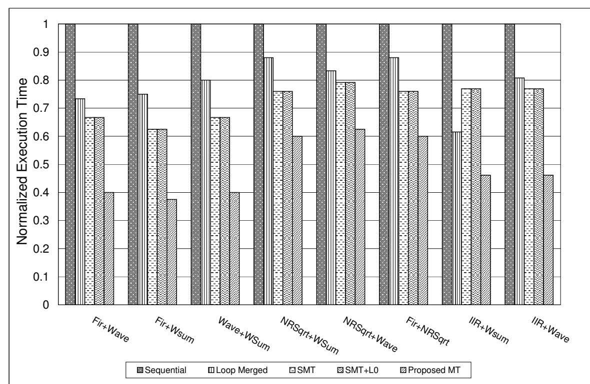


Figure 7. Performance Comparison normalized to Sequential Execution

ing NRSqrt and IIR) because, as these kernels are large, the number of instructions for the loop is large as well, therefore leading to a larger energy consumption.

In case of the *proposed MT*, the tasks are performed simultaneously like in the case of SMT, but the data sharing is at the register-level. For example, in the *FIR+WSum* benchmark, the accesses to the LB have been reduced and the extra memory accesses have also been eliminated (energy gain of *74.08 %*). This explains the energy as well as performance gains over the SMT+L0 technique. Further gains are obtained due to the reduced logic requirement for the loop controllers and the distributed loop buffers. Therefore, the proposed technique has the advantages of both loop-merging as well as SMT and avoids the pit-falls of both these techniques. The results show that the proposed multi-threading (MT) has an energy savings of *74.76 %* over sequential, *49.82 %* over advanced loop merged and *70.01 %* over the enchanced SMT (SMT+L0) technique.

6. Conclusion and Future Work

This paper presented an architectural enhancement to reduce the energy consumed in the instruction memory hierarchy. The proposed architecture enables multi-threaded operation of loops in a uni-threaded processor platform. The hardware overhead required was shown to be minimal. An average energy saving of 70.01% was demonstrated in the instruction memory hierarchy over state of the art SMT techniques along with a performance gain of 32.89%.

The loading of the loop buffers from the IL1 cache is currently done sequentially which is not energy efficient. We are currently developing a scheme to load multiple loop buffers in a parallel and energy-efficient way. We are also currently studying the influence of the number of L0 clusters on the results.

References

[1] *TI DSP Benchmark Suite*. http://focus.ti.com/docs/toolsw/folders/print/sprc092.html.

[2] *Trimaran: An Infrastructure for Research in Instruction-Level Parallelism*. http://www.trimaran.org, 1999.

[3] R. Banakar et al. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proc of CODES*, May 2002.

[4] L. Benini et al. A power modeling and estimation framework for vliw-based embedded system. *ST Journal of System Research*, 3(1):110–118, April 2002. (Also presented in PATMOS 2001).

[5] T. M. Conte et al. Instruction fetch mechanisms for VLIW architectures with compressed encodings. In *Proc of MICRO*, December 1996.

[6] S. Cotterell et al. Synthesis of customized loop caches for core-based embedded systems. In *Proc of ICCAD*, November 2002.

[7] W. Dally. Low power architectures. In *ISSCC, Panel Talk on "When Processors Hit the Power Wall"*, February 2005.

[8] H. De Man. Ambient intelligence: Giga-scale dreams and nano-scale realities. In *Proc of ISSCC, Keynote Speech*, February 2005.

[9] A. El-Moursy et al. Partitioning multi-threaded processors with a large number of threads. In *Proc of ISPASS* , March 2005.

[10] E.Ozer et al. Weld: A multithreading technique towards latency-tolerant vliw processors. In *Proc of HiPC*, 2001.

[11] J. I. Gómez et al. Optimizing the memory bandwidth with loop morphing. In *Proc of ASAP*, pages 213–223, 2004.

[12] A. Halambi et al. An efficient compiler technique for code size reduction using reduced bit-width ISAs. In *Proc of DAC*, March 2002.

[13] M. Jayapala et al. Clustered loop buffer organization for low energy VLIW embedded processors. *IEEE Trans on Computers*, 54(6):672–683, June 2005.

[14] M. Kandemir et al. Compiler-directed scratch pad memory optimization for embedded multiprocessors. In *IEEE Trans on VLSI*, pages 281–287, March 2004.

[15] S. Kaxiras et al. Comparing power consumption of an smt and a cmp dsp for mobile phone workloads. In *CASES*, pages 211–220, November 2001.

[16] A. Lambrechts et al. Power breakdown analysis for a heterogeneous NoC platform running a video application. In *Proc of ASAP*, pages 179–184, July 2005.

[17] P. Op De Beeck et al. CRISP: A template for reconfigurable instruction set processors. In *Proc of FPL*, August 2001.

[18] F. Quillere et al. Generation of efficient nested loops from polyhedra. In *Intl. Journal on Parallel Programming*, 2000.

[19] J. W. Sias et al. Enhancing loop buffering of media and telecommunications applications using low-overhead predication. In *Proc of MICRO*, December 2001.

[20] S. Steinke et al. Assigning program and data objects to scratchpad for energy reduction. In *Proc of DATE*, March 2002.

[21] D. M. Tullsen et al. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc of ISCA*, pages 392–403, June 1995.