Reducing the Overheads of Hardware Acceleration Through Datapath Integration

Pekka Jääskeläinen*, Heikki Kultala, Teemu Pitkänen, and Jarmo Takala

Tampere University of Technology, P.O. Box 553, FIN-33101 Tampere, Finland

*

ABSTRACT

Hardware accelerators are used to speed up execution of specific tasks such as video coding. Often the purpose of hardware acceleration is to be able to use a cheaper or, for example, more energy economical processor for executing the majority of the application in software. However, when using hardware acceleration, new overheads are produced mainly due to the need to transfer data to and from the accelerator and signaling the readiness of the accelerator computation to the processor. We find the traditional mechanisms suboptimal for fine-grain hardware acceleration, especially when energy efficiency is important.

This paper explores a technique unique to Transport Triggered Architectures to interface with hardware accelerators. The proposed technique places hardware accelerators to the processor data path, making them visible as regular function units to the programmer. This way communication costs are reduced as data can be transferred directly to the accelerator from other processor data path components and synchronization can be done by polling a simple ready flag in the accelerator function unit. Additionally, this setup enables the instruction scheduler of the compiler to schedule the hardware accelerator like any other operation, thus partially hide its latency with other program operations.

The paper presents a case study with an audio decoder application in which fine-grain and coarse-grain hardware accelerators are integrated to the processor data path as function units. The case is used to study several different synchronization, communication, and latency-hiding techniques enabled by this kind of setup.

Keywords: hardware acceleration, processor architectures, Transport-Triggered Architectures, synchronization, communication

1. INTRODUCTION

Accelerating software by means of adding extra hardware blocks to run the core functions of the algorithm faster is a common technique to reach the realtime requirements of the application without the need of a more powerful processor to run the application software. Hardware acceleration is widely used in embedded mobile applications in which it is often desirable to use a cheap general purpose processor that consumes little power and is adequate for the most frequent tasks required from the device. In such devices, hardware accelerators are used to reach the high performance requirements of, for example, video decoding.

However, hardware acceleration comes with added costs due to the additional communication and synchronization of the execution with the main processor and the accelerators. This paper evaluates a case in which fine-grain and coarse-grain hardware accelerators are integrated to the processor data path in order to reduce the hardware accelerator interfacing costs. Such integration is natural for a class of processors called Transport Triggered Architectures (TTA) in which operand writes and result reads of data path operations are programmer-visible, thus their timing is fully decided by the programmer/compiler. The contribution of the paper is a discussion and evaluation of several interfacing techniques this kind of integration enables.

* pekka.jaaskelainen@tut.fi

The rest of this paper is organized as follows. Following subsections introduce the background of the overheads and common interfacing methods of hardware acceleration, give a brief introduction to the Transport Triggered Architectures, and look into some previous work related to this paper. Section 2 discusses the technique of integrating hardware accelerators to the TTA data path. Section 3 presents the setup we used to benchmark the different interfacing techniques, and the evaluated interfacing methods itself. Section 4 provides the results from the benchmarks, and Section 5 concludes the paper with a look into some ideas we have for further work in this area.

1.1 About the Overheads of Hardware Acceleration

Several overheads incurred by hardware acceleration are related to the fact that the accelerators are components external to the processor core and the use of accelerators itself is often not accounted for in the processor design. Therefore, several additional overheads due to the need to synchronize the accelerated function calls with the software running in the processor are produced when compared to executing basic instruction set instructions.

One of the overheads is due to communication. Transferring the input and output data between the accelerators and the CPU can become expensive. In data-intensive applications it is usual that the data transports become the bottleneck to the system throughput. This overhead is sometimes avoided with a shared memory design as one can use pointers to communicate the location of input/output thus often avoiding extra copying of data. However, a shared memory design requires either a multiported memory or memory arbitration, which complicates the hardware design further. Further complexity of cache coherency is added in case a data cache is used.[1]

In case of coarse-grain acceleration, the execution time of the accelerated function call can be relatively long, and the waiting time would be better used for something beneficial by the CPU. A popular method to hide long latencies is multithreading. The idea in multithreading is to allow other threads to use the idle processor resources while long latency operations are being executed in other threads. Support for multithreading has been implemented in processor hardware level, for example, Simultaneous Multithreading (SMT) is a popular technique for implementing multithreading on superscalar processors to hide memory latencies and to increase throughput.[2] However, to get benefit from multithreading the executed application must be multithreaded, i.e., there has to be other threads to switch to. This is sometimes not the case in lower-end embedded systems that execute only a single application in the CPU and do not include multitasking operating systems. In such cases the application must be (partially) rewritten to take advantage of multithreading.

When fine-grain acceleration is used, it is sometimes possible to hide most of the accelerator latency with succeeding instructions from the program stream using an instruction scheduler in the processor hardware or in the compiler. Hiding the latency using an instruction scheduler requires the scheduler to know the latencies of the accelerated functions. This is often not the case when the processor and the accelerators are designed separately and are independent IP blocks in the system.

Finally, regardless whether multithreading is used or not, the processor needs to be signalled of the completion of the accelerated task so it can proceed executing code that uses the results of the accelerated function. Interrupt is one of the most popular synchronization mechanism. The accelerator, once completed its function, asserts an interrupt signal which makes the CPU save its current execution context and switch execution to an interrupt handler which acknowledges the interrupt flag, reads the produced results, or something similar.

The overhead of interrupts is due to the context switch which requires many of the processor's registers to be saved to memory before and restored after the execution of the interrupt handler. In addition, when the memory hierarchy includes a cache, the interrupt handler execution is probable to induce more cache misses, thus introducing additional less apparent overheads to the program execution.[3] The cache miss overhead increases as the speed gap between the processor and memory grows. In modern processors, the cost of missing the lowest level of the cache hierarchy can be in hundreds or even thousands of processor cycles.[4] This overhead can be reduced to some extent by using techniques such as dynamic scheduling, prefetching or speculative execution.[5,6] However, the added runtime complexity of these methods often results in higher energy consumption, thus makes them less attractive for mobile embedded systems.

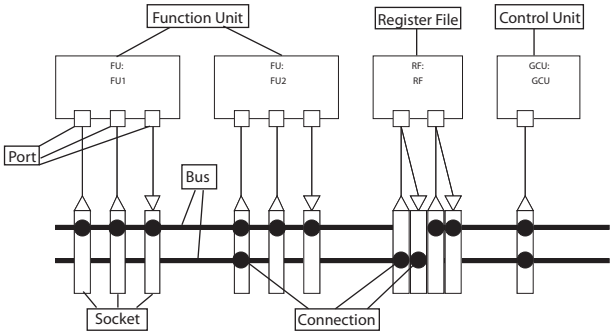

Figure 1. Simple Transport Triggered Architecture processor.

In many cases interrupting the processor after completion of the accelerator function is unnecessary as the application might not require immediate knowledge of the completion of the task. In such a case polling might be a more suitable synchronization mechanism: the processor can go on executing something unrelated to the accelerated function, for example, an another thread, and at a suitable spot query the accelerator whether the task has completed or not. In order to save energy, the processor can switch itself to a low power mode while waiting for the task to complete, waking up once in a while to poll for readiness. This leads to another common way of synchronizing the CPU and the accelerator: simply lock the whole processor when starting the accelerated function call and let the accelerator unlock it after finishing. Clearly, this kind of synchronization might be inefficient with regards to execution speed as the CPU is stalled while it could do something useful.

1.2 Transport Triggered Architectures

Transport Triggered Architectures (TTA) is a class of processors derived from the VLIW-paradigm.[7] TTA improves on traditional VLIW architectures mainly through the reduced requirements for register file ports, higher level of modularity, and additional compiler instruction scheduler freedom.[8]

Figure 1 presents a simple TTA with 2 transport buses, one register file (RF), and two function units (FU). The modularity of TTA is visible in this picture: FUs and RFs are connected through a programmable interconnection network through sockets and buses, which form an indirection separating the FU and RF interfaces from each other. That is, when one adds new FUs to the design, no new RFs or RF ports need to be added, like is often the case with VLIW architectures. The most important difference of TTAs in comparison to VLIWs in the point of view of this paper is that TTAs are programmed by defining data transports between the data path components of the processor, including FUs and RFs, while in VLIW processors the programmer defines which operations are executed in each function unit without having explicit control to the required data transports. That is, registers are read and written implicitly when operations are executed.

1.3 Related Work

Integrating hardware accelerators to a VLIW processor data path is discussed by Busá, Werf and Bekoojie.[9] Their idea is very similar to the principle of this paper: the accelerator is visible as a custom operation to the processor and the instruction scheduler is able to schedule it along with other data path instructions. Their paper concentrates on scheduling the input and output data to the coarse grained operation by means of a scheduling method that takes into account the *timeshape* of the coarse-grained operation to enable transporting data to and from the operation during its execution. One of the main differences in our paper is that we use a TTA processor, thus can exploit software bypassing and the additional scheduling freedom. In addition, their paper seems to assume the latency of a "coarse-grained" hardware accelerator operation is in tens of cycles, thus suitable for a scheduling method they present (their case study uses a 26 cycle operation). In our work, we aim to support hardware accelerators with latencies in tens of thousands of cycles. In such a case, the instruction scheduler is usually not sufficient as such to hide the latency, but requires additional latency-hiding methods such as light-weight threading, which we also evaluate in this paper.

Several concepts related to the ones explored in this paper were published by Lau, Pritchard and Molson in their 2006 paper.[10] Their paper presents techniques involved in extracting hardware accelerators from C procedures semiautomatically. The presented tool is capable of generating the interfacing code ("a driver function") for controlling the accelerator from the main processor. The two most common interfacing options, polling and interrupts, are supported. Their compiler is capable of hiding memory latencies due to communicating data to/from the accelerator and schedule the accelerator control code along with the surrounding application code. However, their work does not seem to include concepts to hide the latency of the accelerator itself. In case the polling option is chosen, the main processor executes a do-nothing loop while waiting for the accelerator. On the other hand, they state that the interrupt synchronization version allows parallel execution of the processor and multiple accelerators by means of a multiprocess realtime operating system (OS). We consider the use of a multiprocess OS sometimes to be overkill especially for embedded mobile applications because the additional software layers of the OS increase energy consumption.[11] One of the major conceptual differences in their approach compared to ours is that they separate custom operations and hardware accelerators. We do not make such a distinction: hardware accelerators are abstracted like custom operations which just might have very long execution latencies. Additionally, we do not force the use of shared data address space between the accelerators and the main processor. In our case, the designer of the system can choose to use a local memory in the accelerator, which involves communication through the FU ports or shared registers, or to share the data address space, which enables communication using pointers to the shared memory.

2. DATAPATH INTEGRATED HARDWARE ACCELERATION WITH TTA

The availability of programmer-controlled data transfers in TTA opens new possibilities. One of the key points that lead us to experiment with data path integration of hardware acceleration with TTAs is that in case of traditional, so called *operation-triggered architectures*, the time instant the results of an operation are transported to the register file is fixed by the operation latency, while in case of TTA the result transport time is freely chosen by the programmer. In addition, as the bypass network of TTAs is fully programmable, programmed bypassing of the register file is possible (an optimization called "software bypassing"[12]). These capabilities are exploited in this paper for low overhead interfacing between hardware accelerators and the master processor to avoid the need for expensive communication and synchronization mechanisms such as shared memory and interrupts.

In our system we treat hardware accelerators with very long latencies just like other function units in the data path: the coprocessor functions are executed like any simple instruction set operation such as addition. The main additional challenge for the compiler from these "very long latency operations" used to access the coarse-grain hardware accelerators is latency hiding. Instruction scheduler of the compiler does a satisfactory job when hiding relatively short latencies, from a couple of cycles up to tens of cycles, but the enormous latencies of the coarse-grain accelerator operations are in practice impossible to hide only with instructions from the scope of the instruction scheduler. Therefore, one of our experiments included the use of very light-weight co-operative threads to hide the accelerator latency.

As a conclusion, in our proposed system, the efficiency of hardware acceleration is improved mainly through the following means:

1. Transport data from the data path operations producing it directly to the hardware accelerator through function unit ports.
    - Avoids the need for a data memory connection in the hardware accelerator.
    - Potential for exploiting software bypassing by not storing the intermediate values to general purpose registers, but writing them directly to the hardware accelerator.
2. Treat the hardware accelerator as a processor operation.
    - Allows hiding the accelerator latency (at least partially) by the compiler instruction scheduler.
    - Allows scheduling the data transports required by the accelerator call using a single load-store unit in the processor.

3. Exploit the additional scheduling freedom of TTA.

   - The program can read results from the accelerator FU when it sees fit. That is, the program is free to execute any code, including another thread, to hide the accelerator latency before reading the results without the fear of the FU overwriting general purpose registers implicitly after it finishes the computation.

3. EVALUATION SETUP

We evaluated several different techniques to communicate and synchronize between hardware accelerators and the TTA processor. The main focus on the study was the effect of the different hardware accelerator interfacing methods to the total cycle count and the energy consumption. Cycle counts were produced using a cycle-accurate TTA processor simulator of our TTA toolset called TTA-Based Codesign Environment (TCE).[13] The simulator provides cycle counts for the processor core and assumes a static memory latency.

Energy consumption was estimated to be directly proportional to the count of active instruction cycles and the count of data memory accesses in the main processor. This coarse approximation gives rough estimates for comparing the energy efficiency of the different interfacing methods. Naturally, the accurate energy consumption numbers depend on many factors, such as the implementation of the processor components and the accelerators, the used process technology and the memory hierarchy.

3.1 The Benchmark Application and the Hardware Configuration

The application chosen for the case study was an open source fixed point implementation of Ogg Vorbis audio decoder called Tremor.[14] The Tremor source code was modified to use hardware accelerated execution of the *modified discrete cosine transform* (MDCT).[15]

Two different acceleration cases were evaluated. The first case used a short latency accelerator for evaluating the use of the compiler instruction scheduler to hide shorter accelerator latencies. In this version, we added a custom operation to compute two outputs y_0 and y_1 with a simple butterfly operation defined as follows

$$y_0 = x_0 T_0 + x_1 T_1 \ ; \ y_1 = x_0 T_1 - x_1 T_0 \tag{1}$$

where the variables x_0 and x_1 are input operands and T_0 and T_1 are constants obtained from a lookup table. The custom operation is extensively used in the MDCT code.

The BUTTERFLY custom operation has four inputs and two outputs. This type of operand counts are rarely seen in traditional architectures, which are often limited to single output and two input instructions. Another "TTA-speciality" in this operation is that its execution timing is fully visible to the compiler, which allows for more detailed scheduling of data transports. In this case the two outputs are ready at different time instants of the custom operation execution. The computation of y_0 takes four cycles while y_1 is ready after eight cycles. Passing this information to the compiler enables scheduling the first result data transport while the second result is being computed.

In the second acceleration case, we experimented with long latency hardware acceleration by executing the whole MDCT algorithm in an hardware accelerator. Some refactoring was done to the code to allow parallel decoding of both the channels in a stereo stream. The parallel operation was used to experiment with hiding the accelerator latency with two lightweight co-operative threads sharing the same accelerator.

TCE allows selecting all resources for the designed TTAs freely. We used this capability to design a simple but realistic TTA processor for running the software parts of the audio decoder in the two acceleration cases. Resources of the used TTA processor are listed in Table 1.

The MDCT accelerator FU provides opcodes for starting all the different MDCT transform sizes used by Tremor. In addition, opcodes are provided for transferring the input and output data to and from the accelerator. This way shared memory is not required and the accelerator communication can be scheduled by the compiler.

The latencies for the different transformation sizes in our implementation are given in Table 2. They are estimations based on results a straightforward implementation of the MDCT accelerator. The accelerator was implemented quickly as another TTA processor without spending much time on optimization.

Table 1. Relevant resources in the simulated TTA-processor.

resource	quantity
Integer ALU	2
Integer multiplier	1
MDCT accelerator FU	1
BUTTERFLY accelerator FU	1
32-bit general purpose registers	128
Register file read ports	4
Register file write ports	1
Transport buses	4

3.2 Experimented Interfacing Methods

The test application was used to evaluate several different mechanisms for interfacing with hardware accelerators. The different cases are summarized in Table 3 and are explained in more detail below.

Methods for interfacing with fine-grain accelerators:

Zero latency. A computational case for evaluating the effectiveness of the instruction scheduling to hide the latency of the fine-grain accelerators. This version assumes that the use of fine-grain accelerators produces no additional overhead to the system except for the data transports. That is, the operation latency is zero.

Differing result latencies. The two results of the BUTTERFLY operation are ready at different times (the first result after four cycles and the second result after eight cycles). This information is provided to the compiler so it can schedule the data transports more efficiently.

Equal result latencies. Both results of the BUTTERFLY are assumed to be ready at the same time, after eight cycles. That is, the additional benefit of TTA scheduling freedom is not fully utilized in the result data transports.

Methods for interfacing with coarse-grain accelerators:

Polling. Poll the MDCT FU output and busy loop. This alternative keeps the processor active, executing the useless task of a do-nothing loop while waiting for the accelerator task to complete.

Interrupts. Interrupt the CPU after completion of the MDCT. A busy loop is still executed, but the processor is interrupted at the completion of the accelerator task. In addition to the overheads of the **polling** method, this adds the overheads of the interruption.

This is a computational case as our TTA template does not currently support hardware interrupts. We approximate the cost of an interrupt to consist only of saving and loading all general purpose registers to memory. The latency of a memory access is assumed to be two cycles. Thus, the total overhead due to saving and loading all the registers per interrupt is: $2 * 2 * 128 = 512$ cycles. This is a vast underestimation because

Table 2. Latencies of different MDCT transformation sizes.

MDCT size	latency (cycles)
64	1200
128	2700
256	6000
512	13000
1024	28000
2048	60000
4096	131000
8192	282000

Table 3. Evaluated methods for hardware accelerator interfacing.

fine-grain acceleration

case	synchronization	latency hiding
no hardware acceleration (baseline)	none needed	none needed
zero latency (unrealistic comparison case)	compiler	none needed
differing result latencies (4/8 cycles)	compiler	instructions from the basic block
equal result latencies (8 cycles)	compiler	instructions from the basic block

coarse-grain acceleration

case	synchronization	latency hiding
polling	poll FU output and busy loop	none
interrupts	interrupt CPU after completion	none
CPU locking	lock CPU while waiting	surrounding code
polling and threading	poll FU output and switch thread	code from other threads

in architectures such as TTAs, the context of the processor is not limited to the general purpose registers, but includes also the pipeline registers of function units, etc.

CPU locking. Lock the CPU while waiting. This is a more sensible alternative to the previous ones energy-wise. The processor is locked by the accelerator until the computation has finished. Some energy should be saved as no instructions need to be fetched from the instruction memory and executed for nothing. In addition, runtime might be improved as the accelerator operation looks like a zero latency operation to the instruction scheduler, thus it is better able to parallelize it with surrounding code.

Polling and threading. Poll FU output as in **polling**, but switch threads always if the result is not ready, thus hide latency with instructions from other threads. In the Tremor case we have two threads, both decoding one channel in the stereo audio stream. This allows executing the software parts of the second channel while the first channel is using the MDCT FU.

Datapath integration. Data is written to the accelerator directly from the producer of the data and read to the consumer of the data, instead of being copied into memory. Several of the previously mentioned methods are combined with this one.

4. RESULTS

The fine-grain acceleration was benchmarked by executing the 2048-point MDCT with the code accelerated with the BUTTERFLY operation. Table 4 lists the cycle counts of the different cases. The result table includes a percentage number indicating the slowdown (increase in cycle count) when compared to the optimal case with the zero latency operation.

This benchmark shows that the detailed knowledge of the operation execution timing in the compiler is very beneficial. In the *differing result latencies* case, the compiler instruction scheduler is able to hide most of the custom operation latency (4/8 cycles) with other code from the same basic block. The ability to schedule the first result data transport 4 cycles earlier in this case seems to help quite a bit, as comparison with the case *equal*

Table 4. Results for the fine-grain acceleration cases.

case	cycle count	slowdown
zero latency	92 296	
differing result latencies	93 570	1.4%
equal result latencies	103 938	12.6%
no acceleration	121 634	31.8%

Table 5. Results for the coarse-grain accelerator cases. Cycle count of decoding a 2.7 sec vorbis file

case	cycle count	TTA active cycles	data memory accesses
software only	96 727 535 (1.00)	same	19 910 620
interrupts (estimated)	87 848 318 (0.91)	same	11 744 095
polling:			
data from memory	87 669 118 (0.91)	same	11 654 495
data path integration	84 378 329 (0.87)	same	10 714 097
CPU locking:			
data from memory	87 354 490 (0.90)	74 022 490	11 655 195
data path integration	84 139 964 (0.87)	70 807 964	10 714 097
co-operative threading:			
data from memory	79 134 509 (0.82)	same	11 770 794
data path integration	78 746 124 (0.81)	same	10 934 360

result latencies shows. In this version, the cycle count is increased by 12.6% from the optimal which makes the 1.4% increase of the *differing result latencies* case seem negligible.

Results for the coarse-grain acceleration benchmarks are given in Table 5. The *cycle count* column gives the total count of cycles required to decode the test sample. This number includes also the cycles when the TTA was locked. *TTA active cycles* is the count of cycles the TTA was executing instructions, thus the locked cycles are not included in this number. This number together with the total count of *data memory accesses* should give indication of the energy efficiency of each alternative. All alternatives are compared to the *software only* case with the ratio in parenthesis.

Data path integration removes the need to transfer some of the intermediate results to the memory, resulting in a both faster and more energy-effective system in all three main alternatives; polling, locking and threading. None of the cases show very large improvement in cycle count through data path integration. The best results from data path integration are in case of polling with four percent unit better speedup and 8% memory access reduction compared to acceleration without data path integration.

Co-operative threads hide the accelerator latency about 6 percent unit better than the polling and locking versions which simply wait until the result is ready. However, with more aggressive organization of the decoding code to threads, more benefit from threading could be gained. Locking the processor while waiting for the results produces about 15% reduction in active processor cycles. The combination of co-operative threads and data path integration gives only a very slight performance increase over the threaded code without data path integration, but on the assumed energy savings from the reduced count of memory accesses the difference is somewhat bigger: 7.1% of memory accesses could be eliminated.

The latency hidden by the threading can be calculated from the performance difference of the polling version and the threaded version The reduction in cycle count in the threaded version is directly the number of accelerator execution that could be hidden by executing other threads. When data path integration was not used, both the pre- and post-filters that were executed before and after the transform could be executed in parallel with the transfer, and 64% of the accelerator latency could be hidden. When data path integration was used, the first post-filter following the transform could not be exected in parallel with the transform as it was reading the data from the accelerator unit, so there was less code to be executed in parallel with the transform. In this case 42% of the accelerator latency could be hidden with threading.

5. CONCLUSIONS AND FUTURE WORK

This paper explored several interfacing methods for fine-grain and coarse-grain hardware acceleration enabled by data path integration using Transport Triggered Architectures (TTA).

The results show that the detailed programming of data transports in TTAs allows hiding fine-grain hardware accelerator latencies efficiently. In our benchmark only 1.4% cycle count increase was measured in comparison to

the zero latency baseline when a fine-grain hardware accelerator was used. The latency of the custom operation was hidden efficiently by means of our compiler instruction scheduler that is able to exploit operation execution timing information.

In the case of the coarse-grained acceleration the data transfers to and from the accelerator can consume noticeable time and energy and by transporting the data directly between the producer operations, the accelerator function unit, and the consumer operations, the overhead can be largely eliminated. The concept of co-operative threads as a way of hiding long accelerator latencies was proven to be beneficial performance-wise. However, the data path integration did not produce large additional execution time benefits on top of co-operative threading. We identified the main benefit of data path integration with coarse-grain accelerators to be practical: the accelerator data transports are done through the processor data path thus avoiding the need for a data memory connection from the accelerator.

In the future, we will look into compiler assisted techniques for low overhead threading to hide coarse-grain accelerator latencies automatically. One idea for improving the hiding of accelerator latencies is to implement a global instruction scheduler that is able to move the accelerator calls above loops and function call sites. This type of optimizations are enabled by the unique programmability of TTAs which allows free placing of the operation result reads.

ACKNOWLEDGMENTS

This work has been supported by the National Technology Agency of Finland under research funding decision 40153/05 and Academy of Finland project number 205743.

REFERENCES

1. R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, "Implementing a cache consistency protocol," in *ISCA '85: Proc. 12th annual int. symp. on computer architecture*, pp. 276–283, IEEE Computer Society Press, (Los Alamitos, CA, USA), 1985.
2. D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," in *ISCA '95: Proc. 22nd annual int. symp. on Computer architecture*, pp. 392–403, ACM, (New York, NY, USA), 1995.
3. D. Tsafrir, "The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)," in *ExpCS '07: Proc. 2007 workshop on Experimental computer science*, ACM, (New York, NY, USA), 2007.
4. J. C. Mogul and A. Borg, "The effect of context switches on cache performance," in *ASPLOS-IV: Proc. 4th int. conf. on Architectural support for programming languages and operating systems*, **26**, pp. 75–84, ACM Press, (New York, NY, USA), April 1991.
5. K. Gharachorloo, A. Gupta, and J. Hennessy, "Hiding memory latency using dynamic scheduling in shared-memory multiprocessors," in *ISCA '92: Proc. 19th int. symp. on Computer architecture*, pp. 22–33, ACM, (New York, NY, USA), 1992.
6. A. Pajuelo, A. González, and M. Valero, "Speculative execution for hiding memory latency," in *MEDEA '04: Proc. 2004 workshop on MEmory performance*, pp. 49–56, ACM, (New York, NY, USA), 2004.
7. J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *ISCA '83: Proc. 10th int. symp. on Computer architecture*, pp. 140–150, IEEE Computer Society Press, (Los Alamitos, CA, USA), 1983.
8. H. Corporaal, *Microprocessor Architectures: from VLIW to TTA*, John Wiley & Sons, Chichester, UK, 1997.
9. N. G. Busá, A. van der Werf, and M. Bekooij, "Scheduling coarse-grain operations for VLIW processors," in *ISSS '00: Proc. 13th int. symp. on System synthesis*, pp. 47–53, IEEE Computer Society, (Washington, DC, USA), 2000.
10. D. Lau, O. Pritchard, and P. Molson, "Automated generation of hardware accelerators with Direct Memory Access from ANSI/ISO standard C functions," *FCCM '06: 14th IEEE symp. on Field-Programmable Custom Computing Machines* , pp. 45–56, 2006.
11. T. Rintaluoma, O. Silven, and J. Raekallio, "Interface overheads in embedded multimedia software," *Embedded Computer Systems: Architectures, Modeling, and Simulation* , pp. 5–14, 2006.
12. H. Corporaal and J. Hoogerbrugge, "Code generation for Transport Triggered Architectures," in *Code Generation for Embedded Processors*, pp. 240–259, Springer-Verlag, Heidelberg, Germany, 1995.
13. P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala, "Codesign toolset for application-specific instruction-set processors," in *Proc. Multimedia on Mobile Devices 2007*, pp. 65070X–1 – 65070X–11, 2007. `http://tce.cs.tut.fi/`.
14. The Xiph Open Source Community, "Tremor - the reference ogg vorbis decoder." WWW. *See* `http://xiph.org/vorbis/`.
15. T. K. Truong, P. D. Chen, and T. C. Cheng, "Fast algorithm for computing the forward and inverse MDCT in MPEG audio coding," *Signal Process.* **86**, pp. 1055–1060, May 2006.