IMPLEMENTING HARDWARE MULTITHREADING IN A VLIW ARCHITECTURE

Stephan Suijkerbuijk and Ben H.H. Juurlink
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
The Netherlands
email: {steef,benj}@ce.et.tudelft.nl

ABSTRACT
Hardware multithreading is a well-known technique to increase the utilization of processor resources. However, most studies have focused on superscalar processor organizations. This paper analyzes which type of hardware multithreading is most suitable for a VLIW architecture and proposes two buffers to increase the efficiency of hardware multithreading. An important goal of our work is that no software changes should be necessary so that legacy code can be executed without change. In order to achieve this we show that there must be a maximum amount of time a hardware thread may be active. The experimental results show that the implemented technique attains a maximum speedup of 27% compared to the single-threaded VLIW processor.

KEY WORDS
VLIW processor, multithreading, block interleaving, TriMedia.

1 Introduction

1.1 Motivation and Contribution

Various micro-architectural techniques to bridge the memory gap have been proposed and implemented. One of these techniques is hardware multithreading. The idea is to start executing a different thread when the current thread is stalled. All hardware multithreading schemes assume that the workload consists of several independent tasks. This can be different programs or parallel threads of a single program. Implementing hardware multithreading requires more micro-architectural changes than some other solutions to bridge the memory gap [2]. However, since the memory gap continues to increase, multithreading has become increasingly worthwhile to implement. In [2] it was concluded that multithreading can increase performance significantly, even with respect to other memory latency hiding techniques. Recently, *simultaneous multithreading* [10] or *hyper-threading* has been implemented in the Intel Xeon processor [5].

Simultaneous multithreading (SMT) is targeted at superscalar processors. The goal of this work is to investigate which type of multithreading is most suitable for VLIW processors and to assess the potential performance improvements. Our qualitative analysis indicates that it is difficult to implement SMT in a VLIW processor without architectural changes, which is an important requirement for our work since legacy code must run without change. We, therefore, propose to implement *block interleaving*. In addition, we propose two buffers to increase the efficiency of hardware multithreading, called the *pending buffer* and the *memory subsystem buffer* (MSS buffer). The pending buffer prevents multiple threads from requesting the same cache line. The MSS buffer is used to match the memory request rate of the multithreaded VLIW processor with main memory bandwidth. We evaluate the performance of the multithreaded VLIW processor using a cycle-accurate simulator of the TriMedia, a VLIW media processor developed by Philips semiconductors [7]. Our experimental results show that a maximum speedup of 27% can be achieved.

1.2 Hardware Multithreading Techniques

Basically, three different hardware multithreading techniques can be distinguished: cycle-by-cycle interleaving, block interleaving, and simultaneous multithreading.

In cycle-by-cycle interleaving, such as implemented in the HEP [8], the processor switches to a different thread each cycle. In principle, the next instruction of a thread is fed into the pipeline after the retirement of the previous instruction. This eliminates the need for forwarding datapaths, but implies that there must be as many threads as pipeline stages. This can be a problem for contemporary superpipelined processors. Furthermore, in order to fully hide the memory latency, the number of threads must be larger than the memory latency in cycles.

In block interleaving [4, 12, 11], also referred to as coarse-grain multithreading, the processor starts executing another thread if the current thread experiences an event that is predicted to have a significantly long latency. If it can be predicted that the latency is larger than the cost of a thread switch, then the processor can at least hide part of the latency by executing another thread. Figure 1 illustrates block interleaving.

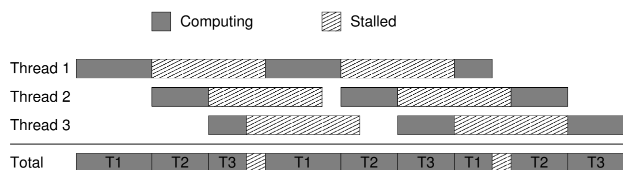

Figure 1. Block interleaved multithreading. The processor switches to another thread if the current thread is stalled.

Both cycle-by-cycle interleaving as well as block interleaving attempt to eliminate *vertical waste*. Vertical waste means that no instructions are issued during a cycle because the current thread is stalled. Simultaneous multithreading (SMT) [10] also tries to eliminate *horizontal waste* (unused instruction slots in a cycle) because it fetches and issues instructions from different threads simultaneously. SMT is designed for superscalar processors, however. To implement this technique in a VLIW processor, several VLIW instructions have to be combined at runtime, as was done in [6]. This can be very difficult and may increase the cycle time because resource conflicts may occur since not every operation can be placed in each instruction slot. Furthermore, as noted in [6], a bit needs to be added to every VLIW instruction to indicate that it can be issued across multiple cycles without violating dependencies. This implies that legacy codes cannot be executed in multithreading mode and cannot benefit from multithreading. Moreover, typically more than 80% of the instruction slots are filled, indicating that there is not much horizontal waste in optimized VLIW applications.

In view of these considerations, we decided to implement block interleaving, switching threads when there is a first-level cache miss.

1.3 Organization

This paper is organized as follows. Section 2 discusses related work. Section 3 describes the organization of the multithreaded VLIW processor. The experimental results are presented in Section 4. Finally, conclusions are drawn in Section 5.

2 Related Work

For an excellent survey of hardware multithreading proposals the reader is referred to [11].

Collins et al. [1] studied a research SMT processor implementing the Itanium instruction set architecture. Itanium is a VLIW-like architecture that fetches and executes instructions in units of bundles. The modeled processor has a fetch bandwidth of two bundles per cycle. If two or more threads are ready to fetch, two of them are allowed to fetch one bundle each. Consequently, this processor cannot eliminate horizontal waste within bundles. Moreover, Collins et al. did not explicitly study this architecture, but used idle hardware thread contexts to prefetch data in speculative threads in order to improve single thread performance.

The Weld architecture [6] attempts to eliminate horizontal as well as vertical waste. Horizontal waste is reduced by combining (welding) VLIWs from different threads dynamically in order to fill empty slots. This is only possible if there are no structural hazards among the instruction slots. The results show that operation welding improves performance by 8.5% on average. The Weld architecture also performs thread speculation. With thread speculation the average speedup is approximately 25% compared to a single-threaded processor. One drawback of the Weld architecture is that it requires a new instruction and some other extensions to the instruction set architecture. This implies that legacy codes cannot be executed in multithreading mode and cannot benefit from multithreading.

Other contributions of this work are:

- We propose the pending buffer to prevent several hardware threads from requesting the same cache line. Although a similar structure needs to be present in superscalar processors with nonblocking caches, such a buffer has not been proposed before for multithreaded VLIW processors.
- We show that if a write miss is followed by a read request from another hardware thread, the pending buffer avoids incoherency problems. Others have not discussed this issue.
- Furthermore, we show that in a multithreaded CPU there is the danger that spin locks create deadlocks. Other studies have not addressed this problem. In fact, most studies have assumed a multitasking workload where threads do not communicate.

3 The Multithreaded VLIW Processor

3.1 The TriMedia Processor

The TriMedia is a VLIW multimedia processor developed by Philips Semiconductors [7]. A VLIW processor can execute multiple instructions simultaneously if they are independent and free of structural hazards. A VLIW compiler guarantees a functional correct distribution of different executable instructions. The TriMedia is a VLIW processor with 5 instruction slots with the capability of being a stand alone processor as well as a co-processor for multimedia applications. Multimedia instructions have been added to the instruction set to optimize the TriMedia for multimedia applications such as MPEG2 encoders and decoders.

This research is based on the TM1000, which is a TriMedia instance, but the results also apply to other VLIW processors. The TM1000 has a 16KB data cache and a 32KB instruction cache. Both caches are 8-way set associative with a block size of 64 bytes and LRU replacement. A high bandwidth of 400 MB/s supplies the data streams to the processor to perform its calculations.

3.2 Top Level Architecture

To achieve fast context switching, every hardware thread has its own program counter and register file. The TriMedia has 128 registers and the compiler assumes this number. To reduce the size of all register files, we would have to reduce the number of registers available to each hardware thread. This, however, would require changing the compiler which is beyond the scope of this research.

The data and instruction caches are not duplicated because it would increase the chip size too much to justify multithreading. Consequently, every hardware thread needs to share the caches with all other hardware threads. This can have positive as well as negative effects. Data that is fetched by one hardware thread, can be accessed quickly by another thread when caches are shared. However, cache pollution may occur if one thread replaces a cache line that is needed by another thread. This cannot be prevented, but we can decrease the probability of cache pollution by using a reasonable cache size and by using a global clock to implement LRU. If one hardware thread accesses a cache line, the other hardware threads will know that this cache line has been used recently.

The functional units do not have to be duplicated. At any time only one hardware thread is active and requires functional units. When a multicycle operation is executed by a pipelined functional unit and a switch occurs, then the next hardware may issue operations to this unit but the first result produced by this unit is written back to the register file of the previous hardware thread.

The internal MMIO (Memory Mapped Input Output) registers are fully duplicated. These registers contain some values which may be shared between hardware threads, but the size of these registers is small. Therefore, duplication is preferred. In the TriMedia, the interrupt vectors reside in the MMIO space. In other VLIW architectures, these might need to be duplicated specifically.

Thread switches are performed when there is a first level (L1) instruction or data cache miss. Whether a memory access will produce a cache miss is unknown when the instruction is fetched. Therefore, when the cache miss is detected, other instructions that are later in program order are already in the pipeline. This problem can be solved in two ways. First, we can let the pipeline execute until it is empty. The disadvantage of this method is that the cost of context switching increases, since the pipeline is completely empty and unused when the switch is performed. Alternatively, as described in [12], a single-cycle context switch overhead can be achieved by keeping multiple copies of the internal pipeline registers and being able to swap in the whole state in a single cycle. This is the method we have implemented.

3.3 Pending and Memory Subsystem Buffers

In this section we introduce two hardware buffers: the *pending buffer* and the *memory subsystem* (*MSS*) *buffer*.

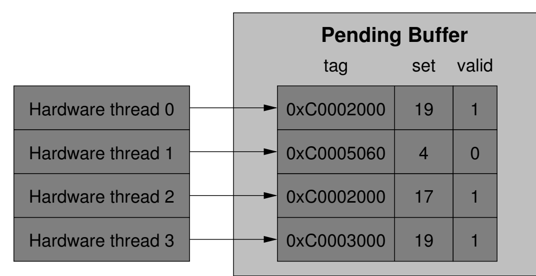

Figure 2. The pending buffer. Every hardware thread has a dedicated entry to put its line that is being fetched. The entire pending buffer is searched in case of a cache miss.

These buffers are used to decrease memory bandwidth requirements, to guarantee cache coherence, and to match the request rate of the multithreaded CPU with main memory bandwidth.

The pending buffer prevents several hardware threads from requesting the same cache line, thereby decreasing unnecessary traffic on the bus interface. It is a fully-associative cache-like structure that is shared between all hardware threads. Every thread has a dedicated entry in the pending buffer and each entry consists of the address (tag and index (set) bits) of a requested cache line and a valid bit, as illustrated in Figure 2. If a hardware thread experiences a cache miss, it writes the tag and index of the requested cache line to its corresponding entry and sets the valid bit. If another hardware thread incurs a cache miss, it searches all entries in parallel to check if another thread requested the same cache line. If this is the case, it does not make the request to fetch the data but informs the hardware thread that requested the cache line that it waits for it too. When the data arrives in the cache, all threads waiting for the requested cache line are re-activated.

We remark that a similar structure must be present in CPUs with nonblocking caches that allow several simultaneously outstanding misses. The TriMedia VLIW processor, however, uses an in-order pipeline which stalls on a cache miss, and therefore has no structure that records information about outstanding misses.

In addition to reducing the memory bandwidth requirements, the pending buffer also avoids possible incoherency problems. When there is a write miss in the cache, then there is a thread switch and there is the possibility that the next hardware thread reads the same address. Therefore, we must force the hardware thread that made the write request to finish first, before the read of the other hardware thread is executed. The pending buffer guarantees this.

The bus interface connects the TriMedia to main memory. The processor sends its requests for data to main memory through the bus interface. The bus interface can have at most one outstanding request. This is a problem when multithreading is implemented, because then there can be as many outstanding requests as hardware threads. This problem can be solved with multiple bus interfaces, a

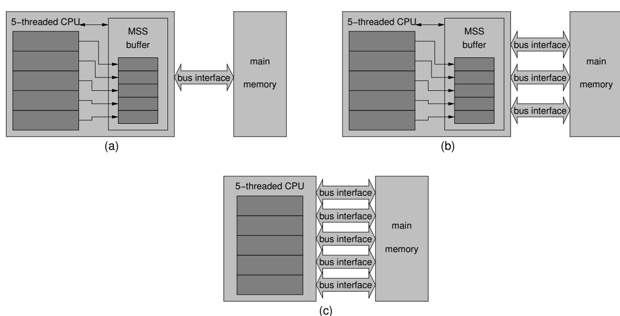

Figure 3. Three possibilities of interfacing a 5-threaded CPU to the main memory: (a) single bus interface, (b) several bus interfaces, (c) fully connected interface (no MSS buffer needed).

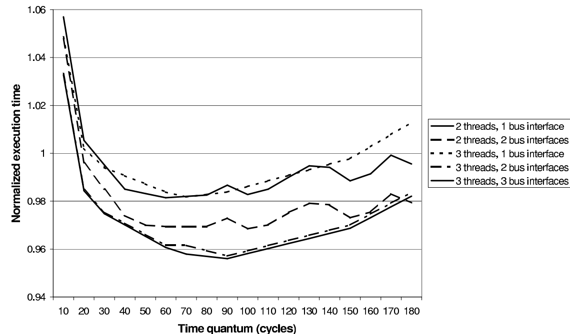

Figure 4. Normalized execution time of the benchmark *opt-mpeg2dec* as a function of the time quantum, for various number of hardware threads and bus interfaces.

memory subsystem buffer (*MSS buffer*), or a combination of both. This is illustrated in Figure 3.

If there is only one bus interface, the MSS buffer must serialize the memory requests made by the multithreaded processor. Clearly, the bus interface can become a bottleneck if the processor issues requests faster than memory can service them. If main memory bandwidth can be increased (for example, by organizing it in several memory banks) but not as much as the number of hardware threads, the MSS buffer is used to match the request rate of the processor with the number of bus interfaces and to distribute requests over different memory banks. This is illustrated in Figure 3(b) where at most five requests need to be distributed over three bus interfaces. If there are as many bus interfaces as hardware threads, then the MSS buffer is not needed. In that case the bus interfaces match the request rate with main memory bandwidth. However, it is unlikely that main memory can service requests at the same rate as they are issued.

3.4 Time Quantum

To implement atomic operations, the TriMedia uses load linked (LL) and store conditional (SC) instructions [3]. These instructions are used to implement *spin locks*, locks that a processor continuously tries to acquire by spinning in a loop until it succeeds. Spin locks are useful when the lock is expected to be held for a short amount of time. However, in a multithreaded CPU there is the danger that spin locks create deadlocks. If one hardware thread is spinning on a lock that must be released by another thread, it will wait forever because the other thread will never become active since there is no cache miss.

An obvious solution to this problem is to forbid spin locks in user applications but to enforce a thread switch when trying to acquire a lock. However, as remarked before, an important objective of this research is not to necessitate software changes so that legacy code can be run unmodified. We, therefore, decided to implement a time quantum during which a hardware thread may be active before a thread switch is performed. Thus, a thread switch is performed either when there is an instruction or data cache miss or when the time quantum has expired.

Ideally, the time quantum is as large as possible. Every time a switch is forced because the time quantum has expired, cycles are unnecessarily wasted. Either the switch is too early and the hardware thread is just interrupted while performing its task, or the switch is too late and the spin lock loop has been executed a few times. It depends on the number of spin lock loops. If they are many, then the time quantum must be small. If there are few, then the time quantum must be large.

4 Experimental Evaluation

To evaluate the performance of the multithreaded TriMedia, we used `cakesim`, a cycle-accurate, execution-driven simulator of the CAKE multiprocessor [9]. `cakesim` was modified to simulate a multithreaded TriMedia. Two different MPEG-2 decoders, *opt-mpeg2dec* and *normal-mpeg2dec*, were used as benchmarks. *Opt-mpeg2dec* is an MPEG-2 decoder optimized for the TriMedia, while *normal-mpeg2dec* is an off the shelve MPEG-2 decoder. Both benchmarks have been multithreaded using *pthreads* and mainly exploit the available data-level parallelism. In our simulations, we varied the cache sizes, the time quantum, the number of hardware threads, and the number of bus interfaces. The memory system is modeled in detail, with CAS latency, row to column delay, row access time, etc., and we used the default memory system parameters. Because the simulations are very time-consuming, at most three hardware threads have been simulated.

Figure 4 depicts the execution time of the optimized MPEG-2 decoder *opt-mpeg2dec* as a function of the time quantum. The execution time (number of cycles) has been normalized w.r.t. the execution time of the single-threaded TriMedia processor. The actual TriMedia cache sizes have been used (16KB data cache and 32KB instruction cache). Both caches are 8-way set-associative.

Several observations can be drawn from this figure. First, the optimal time quantum is approximately 90 cycles. This is quite small and indicates that there are many spin lock loops. This justifies software changes to detect spin lock loops at compile time. Second, the maximum performance improvement is approximately 4.5%, and when the time quantum is too small or too high, the multithreaded TriMedia performs even worse than the single-threaded processor. One has to keep in mind, however, that *opt-mpeg2dec* is heavily optimized for the TriMedia and achieves a very good CPI of 1.087 already on the single-threaded processor. Since the minimum CPI is 1 (cycles per VLIW instruction, corresponding to 5 operations per cycle), this implies that the performance cannot be improved by more than $1 - 1/1.087 = 8.0\%$. Another observation is that the multithreaded TriMedia with three hardware threads and three bus interfaces achieves the highest performance, as can be expected. However, three threads with two bus interfaces perform almost as well.

We have also performed experiments with different cache sizes. Reducing the cache size can have positive as well as negative effects on the performance of the multithreaded processor relative to the performance of the single-threaded processor with the same cache size. On the one hand reducing the cache size generally increases the number of cache misses and hence the number of memory stall cycles that may be hidden by multithreading. On the other hand, if the cache size is decreased then there is more interference between the threads because the data fetched by one thread may replace data needed by another thread.

Figure 5 depicts the execution time of the benchmark *opt-mpeg2dec* on the single-threaded processor and on the two-threaded processor with two bus interfaces for three different cache sizes. For the multithreaded processor, the optimal time quantum is used. For this benchmark, if the size of the instruction and data cache is decreased, hardware multithreading provides more performance benefit. With an 8KB data cache and a 16KB instruction cache (half the real sizes), the maximum performance gain is around 15%. In this case the increased number of memory stall cycles which can be hidden by multithreading plays a more important role than the interference caused by the fact that there is less cache memory available to each thread. With a 32KB data cache and a 64KB instruction cache (twice the actual sizes) hardware multithreading provides only a marginal benefit of approximately 0.3%. For this cache configuration there are very few memory stall cycles that can be hidden by executing another thread.

The benchmark *normal-mpeg2dec* is an off-the-shelve MPEG-2 decoder. It achieves a CPI of 1.403 on the single-threaded processor, indicating that it has more room for improvement than *opt-mpeg2dec*. Figure 6 depicts the normalized execution time for this benchmark as a function of the time quantum. Again, the actual TriMedia cache sizes have been used.

For this benchmark, the multithreaded TriMedia is substantially faster than the single-threaded processor.

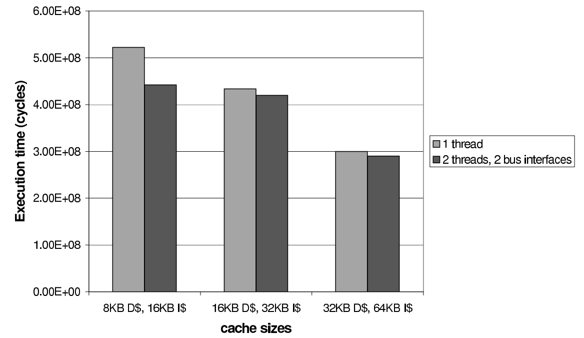

Figure 5. Execution time of *opt-mpeg2dec* on the single-threaded processor and on the two-threaded processor with two bus interfaces for various cache configurations. For the multithreaded processor, the optimal time quantum is used.

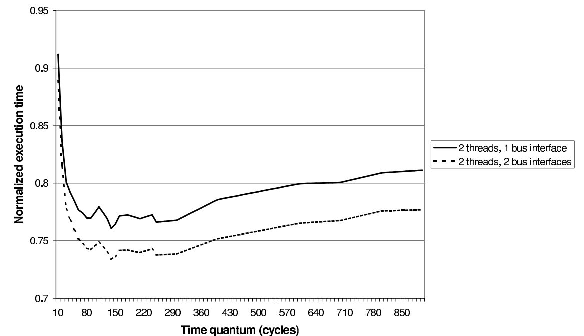

Figure 6. Normalized execution time of *normal-mpeg2dec* as a function of the time quantum, for various number of hardware threads and bus interfaces.

When the time quantum is between 170 and 330 cycles, the performance improvement is approximately 27%. Since the maximum performance improvement is $1 - 1/1.403 = 28.7\%$, this is reasonably close to the maximum. Furthermore, the performance is less dependent on the value of the time quantum. This indicates that there are fewer spin lock loops in this program than in the *opt-mpeg2dec* benchmark.

Figure 7 compares the execution time of the benchmark *normal-mpeg2dec* on the single-threaded processor to the execution time on the two-threaded processor with two bus interfaces for different cache sizes. In this case the performance boost due to multithreading is relatively independent of the cache sizes. For an 8KB data cache and a 16KB instruction cache it is 25.0%, for a 16KB data and a 32KB instruction cache it is 26.6%, and for a 32KB data and a 64KB instruction cache it is 22.6%. For this benchmark, although reducing the cache size increases the number of memory stall cycles that may be hidden by multithreading, the multithreaded architecture is unable to hide all or most memory stall cycles due to increased cache interference. Vice versa, increasing the cache size reduces the number of memory stall cycles on the single-threaded processor but it also reduces the negative effects of cache sharing on the two-threaded processor.

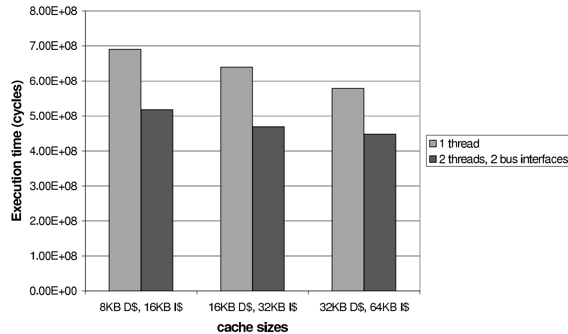

Figure 7. Execution time of *normal-mpeg2dec* on the single-threaded processor and on the two-threaded processor with two bus interfaces for various cache configurations. The optimal time quantum is used.

5 Conclusions

In this paper we have evaluated the efficiency of implementing block interleaving in the TriMedia, a VLIW processor developed by Philips semiconductors. Several micro-architectural changes are required to enhance the TriMedia with the capability of block interleaving. Caches and functional units can be shared and the general purpose and MMIO registers need to be duplicated. Furthermore, to achieve very fast context switching, the internal pipeline registers have to be stored on a thread switch.

We have proposed the pending buffer to prevent several hardware threads from requesting the same cache line, thereby reducing memory bandwidth requirements. We have also proposed the memory subsystem buffer to match the request rate of the multithreaded processor with main memory bandwidth.

Thread switches are performed at every first level cache miss and each time the time quantum has expired. The time quantum is used to prevent deadlocks, because otherwise a hardware thread may wait in a loop indefinitely for a spin lock to become free. Deadlocks can also be avoided by implementing locks differently, but one of the objectives of this research was to require no software changes, so that legacy code can be run without change.

We have provided experimental results for two benchmarks, an MPEG-2 decoder optimized for the TriMedia and an off-the-shelve MPEG-2 decoder. For the optimized decoder we have achieved a maximum performance improvement of approximately 4.5%. Given that we have ignored cycle time effects, this seems to indicate that implementing block interleaving in a VLIW processor is not worthwhile. However, the optimized MPEG-2 decoder achieves a very good CPI already on a single-threaded processor and, therefore, its performance cannot be improved by more than 8.0%. Furthermore, to achieve this CPI significant efforts from the application developers were required. The off-the-shelve decoder exhibits more vertical waste since only compiler optimizations have been applied to this benchmark. For this benchmark we have achieved a performance improvement of approximately 27%, whereas the maximum performance improvement is 28.7%.

Concluding, we find that if memory latency can be hidden by scheduling loads well before the results of the loads are used, then hardware multithreading is not very effective for VLIW processors. However, to do so sometimes requires significant efforts from the application developers. If this is not feasible and/or if an application is memory-bound, then hardware multithreading can be employed to hide the memory latency in VLIW processors. Furthermore, if hardware multithreading is to be implemented, spin locks have to be addressed.

References

[1] J.D. Collins, H. Wang, D.M. Tullsen, C. Hughes, Y-F. Lee, D. Lavery, and J.P. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *Proc. Int. Symp. on Computer Architecture*, 2001.

[2] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proc. 18th Int. Symp. on Computer Architecture*, 1991.

[3] J.L. Hennessy and D.A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[4] J. Kreuzinger and T. Ungerer. Context Switching Techniques for Decoupled Multithreaded Processors. In *Proc. Euromicro'99*, 1999.

[5] D.T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J. Alan Miller, and M. Upton. Hyper-threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1), 1002.

[6] E. Özer, T.M. Conte, and S. Sharma. Weld: A Multithreading Technique Towards Latency-Tolerant VLIW Processors. In *Proc. Int. Conf. on High Performance Computing*, 2001.

[7] S. Rathnam and G. Slavenburg. An Architectural Overview of the Programmable Multimedia Processor, TM-1. In *Proc. Compcon'96*, 1996.

[8] B. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *Proc. 4th Symp. Real Time Signal Processing IV*, 1981. SPIE.

[9] P. Stravers and J. Hoogerbrugge. Homogeneous Multiprocessing and the Future of Silicon Design Paradigms. In *Proc. Int. Symp. on VLSI Technology, Systems and Applications*, 2001.

[10] D.M. Tullsen, S. Eggers, and H.M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proc. 22th Int. Symp. on Computer Architecture*, 1995.

[11] T. Ungerer, B. Robič, and J. Šilc. A Survey of Processors With Explicit Multithreading. *ACM Computing Surveys*, 35(1), 2003.

[12] W. Weber and A. Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proc. Int. Symp. on Computer Architecture*, 1989.