

# Performance Evaluation of Interleaved Multithreading in a VLIW Architecture

S. Suijkerbuijk<sup>1,2</sup>, P. Stravers<sup>2</sup>, S. Vassiliadis<sup>1</sup>, B.H.H. Juurlink<sup>1</sup>

<sup>1</sup>Computer Engineering Laboratory, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology  
Mekelweg 4, 2628 CD Delft, The Netherlands

<sup>2</sup>Philips Research Laboratory  
Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands

stephansuijkerbuijk@hotmail.com

**Abstract**—*Interleaved multithreading* is a technique in which the processor starts executing a different task when the current thread is stalled. However, whereas different forms of hardware multithreading have been extensively evaluated in superscalar processors, an evaluation of multithreading techniques in a *VLIW architecture* is frequently missing. The objective of this paper is to determine an efficient method of implementing interleaved hardware multithreading in the *TriMedia* and evaluate the performance. The *TriMedia* is a multimedia VLIW processor designed by Philips semiconductors. Currently, multithreading is not implemented in the *TriMedia*. First, the details of the used interleaved multithreading method are given. After that, the architectural changes that are made in to cycle-accurate simulator of the *TriMedia* are described. Then, the various test result are presented. Finally we discuss the conclusions that can be drawn from the simulation results.

**Index Terms**—Interleaved multithreading, VLIW processor, *TriMedia*

## I. INTRODUCTION

Interleaved multithreading [1] is a technique in which the processor starts executing a different task when the current thread is stalled. The *TriMedia* [7] is a multimedia VLIW processor designed by Philips semiconductors. The objective of this research is to assess the performance potential of a *TriMedia* enhanced with interleaved multithreading.

A cycle-accurate simulator of the *TriMedia* was modified to support interleaved multithreading. Every hardware-thread capable of running a task has its own general-purpose registers. To limit the increase in chip size, the hardware-threads share the data and instruction caches. Another micro-architectural modification prevents fetching a cache-line that has already been

requested by another hardware-thread, but has not arrived yet.

Thread switches occur when there is a primary cache miss and threads are scheduled in a round-robin manner. Without software changes, there must be a maximum amount of time a hardware thread may be active, since otherwise a thread may hold the CPU indefinitely. This amount of time is called the time quantum. We assume that the pipeline register are replicated, which allows to perform thread switches in one cycle.

The performance of the enhanced has been measured using two different mpeg2 decoders. The results show that the time quantum should be carefully chosen. If it is too small, there is too much thread switching overhead. If it is too large, the original *TriMedia* performs even better than the multithreaded one. In addition, the results show that the multithreaded processor decreases the number of cycles by at most 25

Concluding, we have identified what the improvement would be if interleaved multithreading would be implemented in the *TriMedia*. The performance benefit is not sufficient to warrant implementing this technique. The advantage of changing the software should be investigated in order to fully assess the performance potential of interleaved multithreading.

This paper is organized as follows. In Section II we briefly describe different hardware multithreading techniques and the *TriMedia* processor. Section III discusses how multithreading can be integrated in the *TriMedia* VLIW processor, describes which micro-architectural structures need to be duplicated and which can be shared, and introduces three buffers that improve performance. Simulation results are presented in Section IV and conclusions are drawn in Section V

## II. BACKGROUND

### A. Multithreading in Hardware

Throughout the years a trend can be observed that both the processor and the memory chips become faster. However, the growth of the processing power exceeds the growth of the speed of memory. This causes what is called the *memory gap*. The processor requires data faster than it can access it. With multiprocessors this data can even reside in remote memory locations with an even bigger latency. The memory gap has become a performance limiting factor.

Various micro-architectural techniques to bridge the memory gap have been proposed and implemented. Hardware multithreading is one of these techniques. All hardware multithreading schemes assume that the workload of a CPU consists of several independent tasks. This can be different programs or parallel threads of a single program. Implementing a hardware multithreading techniques requires significantly more micro-architectural changes than some other solutions to bridge the memory gap as stated in [2]. However, since the memory gap continues to increase, multithreading has become increasingly worthwhile to implement. Besides this trend, some changes in the processor micro-architecture for other memory latency tolerance techniques were expandable for multithreading and the research on multithreading made it more efficient. In [2] it was concluded that multithreading or multiple contexts can increase performance significantly, even with respect to other memory latency tolerance techniques. Recently *simultaneous multithreading* or *hyper-threading* has been implemented in the commercial Intel Xeon processor [6].

### III. HARDWARE MULTITHREADING TECHNIQUES

Opposed to simultaneous multithreading, interleaved multithreading [1] has to switch to another task as depicted in Figure 1. Depending on the switching technique

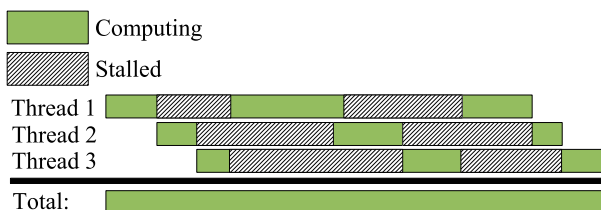


Fig. 1. The workload of a multithreaded processor

and implementation, a range of possibilities exist. If we take cycle-by-cycle interleaving, such as implemented in the HEP [3], the switching penalty must be minimal (essentially zero), since there is a thread switch every cycle. To use this method efficiently there must be as

many executable tasks as there are pipeline stages. Every cycle a new instruction will be fed into the pipeline. With a long pipeline, the number of executable tasks can be smaller than the number of stages. This will result in empty unused pipeline stages. Another method is called interleaved multithreading [10], also referred to as block interleaving. Opposed to cycle-by-cycle interleaving, interleaved multithreading does not execute tasks for a fixed amount of cycles. Instead the processor executes another task if the data needed by a task is predicted to have a significantly long latency. To determine what a significantly long latency is, the latency needs to be compared to the cost of a thread switch. If we can predict that the latency is larger than the cost of switching to another task, then the processor will use at least some of the unused cycles to execute another task when we switch at that moment.

Whether a load or store will produce a cache miss is unknown when the instruction is fed into the pipeline. At the time the decision is made to switch to another task, the pipeline will contain instructions that are after the instruction that causes the switch. We can let the pipeline execute until it is empty. This flushing method has the disadvantage that the switch will cost extra cycles, since the pipeline is completely empty and unused when the switch is made. Alternatively, if we can quickly, in one cycle, store all pipeline registers the chip size will increase, but the cost of a thread switch decreases. The pipeline can be used again when the task is re-activated.

### A. TriMedia Processor

The Trimedia is a VLIW multimedia processor developed by Philips Semiconductor [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 an TriMedia instance. The TM1000 has a data cache of 16KB and an instruction cache of 32 KB. Both caches are 8-way set associative with a block size of 64 bytes and employ the Least Recently Used (LRU) replacement algorithm. A high bandwidth of 400 MB/s supplies the data-streams to the processor to do its calculations.

This research uses the TM1000 as a starting point, but

the conclusions can be easily extended to other VLIW processors.

#### IV. INTEGRATING MULTITHREADING IN THE TRIMEDIA

##### A. The Implemented Multithreading Technique

Before we can integrate multithreading in the TriMedia, we need to select the type of multithreading we are going to implement. Simultaneous multithreading [4], [5] is generally faster than interleaved multithreading, because it eliminates horizontal waste (unused instruction slots in a cycle) as well as vertical waste (no instructions are issued during a cycle, because execution is stalled), while interleaved multithreading eliminates only vertical waste. This comparison is made assuming that both implementations are capable of running the same number of tasks. Simultaneous multithreading, however, is designed for superscalar processors. In a VLIW processor conflict of resources might occur, because of unknown and unpredictable latencies of cache misses at compile time. This makes interleaved multithreading the only possible choice.

In interleaved multithreading only one of the available tasks is executing at any time. When a switch occurs, the processor begins or continues executing another task. Different types of interleaved multithreading exist. There are two possibilities. The switch is performed at fixed intervals or at variable intervals. The already mentioned HEP uses a fixed interval of one cycle. In this specific example we need as many task as pipeline stages. This is not usually the case, which makes cycle-by-cycle interleaving not a suitable choice for interleaved multithreading in a VLIW architecture. even if we increase the length of the interval we will get the problem that the latency of the cache miss appears as empty pipeline stages. The object of this research is to hide these latencies. Therefore the choice is to employ variable intervals.

The ideal moment to switch to another task is just after data is requested that does not reside in a nearby cache. To approach this a task switch is performed when there is a first level cache miss. However, when the data resides in the second level cache, the switch might cost more cycles than the amount of cycles needed to bring the data to the processor. In [11] has been given an implementation of interleaved multithreading with a switch cost of only one cycle. We take this number in our simulations.

When we switch to another task we take the next task in line using the round robin method, returning to the first task when we switch from the last task. The possibility

of priorities has been set aside. The overhead increase with task priorities and the danger arises that a task is never executed. Furthermore, adding priorities to task requires compiler changes, which is out of the scope of this research. The simplicity of the round robin method prevails upon the advantages of priority scheduling.

##### B. Top Level Architecture

The TriMedia can execute only one task at the same time. To create the capability of interleaved multithreading, the TriMedia requires additional hardware. The TriMedia needs to store the state of every task when this task is not active. In order to achieve this several parts of the micro-architecture need to be duplicated and some not.

The data and instruction caches of the TriMedia will not be duplicated for every task that can be handled in parallel, hereafter called *hardware thread*. Consequently, every hardware thread needs to share the caches with all the other hardware threads. The disadvantage is that duplicating the caches is usually faster. The advantage that motivates the decision not to duplicate is the chip size. Duplication of the caches causes an increase in chip size that is too large to justify multithreading on a processor instead of using multiple processors.

Every general purpose registers will be duplicated for every hardware thread. The TriMedia has 128 registers and the compiler assumes this number. Changing the number of registers of hardware thread will necessitate in changes to the compiler, which is out of the scope of this research.

The functional units do not have to be duplicated. At any time only one hardware thread is active and requires functional units. When a complex functional unit that requires more than one cycle is active and a switch occurs, the next hardware thread may issue operations to this functional unit. But the first result predicted by this functional unit is part 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 the easy solution of duplication is preferred. In case of the TriMedia, the interrupt vectors reside in the MMIO space. In other VLIW micro-architectures these might need to be duplicated specifically.

Furthermore the entire state of the TriMedia must be stored in order to be able to continue the task at the moment it stopped due to a switch. This depends on the method of implementation of interleaved multithreading. A store of the complete pipeline can offer a quick switch,

whereas a slow switch may result in a smaller chip when flushing the pipeline. In this research we assume a one cycle switch, which means total duplication of the pipeline registers.

### C. Cache Structures

The hardware threads share the data and instruction caches, which has serious effect when considering cache coherency and speed. Coherency is achieved if the following three conditions are met [9]:

- 1) If there is first a write to a location and then a read by the same processor/hardware thread to the same location, the read should return the value written by that processor/hardware thread, provided that there are no writes to that location by other processors.
- 2) When there is a write to a location, the read from that same location by another processor/hardware thread must return the written value if there is sufficient time between the instructions.
- 3) Writes to the same location are seen by all other processors/hardware threads in the same order.

These rules apply to a multiprocessor environment and to a multithreading environment if the software views the hardware threads as separate virtual processors. The first condition is handled by the cache controller, which handles these request. A write with low priority can be handled at a later time than the read with higher priority, but issued after the write. However, the cache controller must make sure that the read does read the value that was changed by the write instruction.

The second requirement is a bit trickier with multithreading. When there is a write hit to a location in the cache, there is no switch and thus no other hardware thread reads the same address before the write is finished. However, when there is a write miss in the cache, there can be a thread switch and the possibility arises that the next hardware thread reads the same address. Thus we must force the hardware thread that made the write request to finish first, before the read of the other hardware thread is executed. This is done using the *pending buffer* which will be explained in the next section.

The data that is already fetched and/or used by one hardware thread, can be accessed quickly by another hardware thread when using shared caches. If one thread wants to access a cache line, another hardware thread might have invalidated this line to replace it with another cache line. This is called cache pollution. We cannot prevent that from happening completely, because of the limits to the cache sizes, but we can decrease the

probability of cache pollution by using a reasonable cache size and by using a global clock for implementing LRU. If one hardware thread uses a cache line, the other hardware threads will know that this cache line has been used recently.

### D. Buffering

In this section we will introduce the *pending buffer*. The principle is simple, the pending buffer prevents other hardware threads to request the same cache line. The bus interface that connects the multithreaded CPU does not have to request the same cache line twice. The pending buffer decreases unnecessary traffic on the bus interface and guarantees cache coherence.

The pending buffer consists of sets, tags and valid bits as shown in Figure 2. The pending buffer is shared between all the hardware threads on the same multithreaded CPU. Every hardware thread has its own group of set, tag and valid bit. If a hardware thread incurs a cache miss, be it in instruction cache or data cache, it will put the corresponding set and tag in the pending buffer and changes the corresponding valid bit to TRUE. If another hardware thread experiences a cache miss, it checks the entire pending buffer, which has as many entries as hardware threads. If it finds the corresponding set and tag in the buffer, it does not make the request to fetch the data. Instead it informs the hardware thread that requested the cache line that it waits for it too. When the data arrives in the cache, the original hardware thread that stalled first, is woken up. It continues computing as normal, but it also checks if any hardware thread was waiting for the same cache line. If it finds that to be true, it wakes-up the other hardware threads.

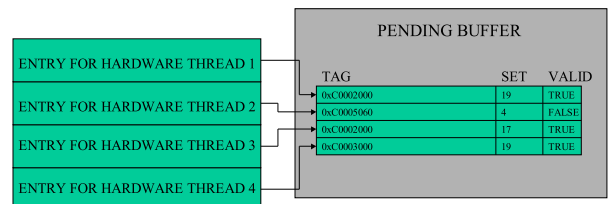


Fig. 2. The pending buffer. Every hardware thread has a dedicated entry to put its line that is being fetched. We need to search the entire buffer if we want to see if a line is pending.

This way of communication is of equal importance when there is a copy back situation. In this situation a hardware thread invalidates a cache line containing address *A* to make room for another cache line that it needs. After a switch another hardware thread may ask for that line. But the line containing address *A* is not in the cache since it was invalidated. Therefore the

line must be fetched, but the copied back value must be found. In order to guarantee that the read finds the copied back value, a copy back buffer is introduced, which is similar as the pending buffer. In this buffer not the address of the line that is fetched is stored, but the address of the copied back line. A read checks if the address of the line is in the copy back buffer and does not continue if it finds that address. A switch follows. The read can only finish if the copy back is successfully completed and the copy back buffer is cleared with the address. Then the line that was copied back must be fetched again. Another possibility is to store the line which is copied back and supply it to the second hardware thread if it requests it before the completion of the copy back. This causes much more overhead, since we do not know if the data is still valid, or requested by other resources.

The bus interface connects the TriMedia to main memory. The processor sends its requests through the bus interface and gets its data. The bus interface is limited to one outstanding request, which will become a problem when multithreading is implemented, because a multithreaded CPU can have multiple outstanding requests. The maximum is equal to the number of hardware threads. This problem can be solved with multiple bus interfaces, a memory subsystem buffer (*mss buffer*) or a combination of both. This is illustrated in Figure 3. The

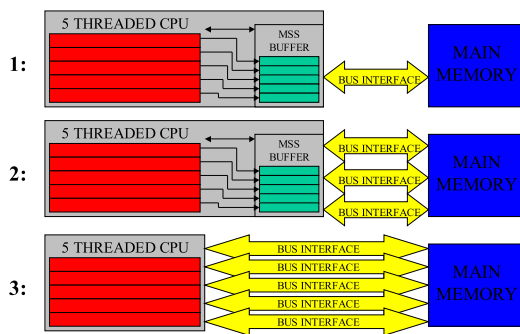


Fig. 3. Three possibilities of bus interfaces to the main memory in 5 threaded CPU. 1: Single Bus Interface, 2: Variable Bus Interface, 3: Full Connected Interface (no mss buffer needed).

first possibility uses only the mss buffer, which sequentializes every request from the multithreaded processor. The disadvantage of this approach is the possibility that all hardware threads are making request using the bus interface. Every request will be handled sequentially, and thus it will take a long time for every request to be handled. The advantage is the single bus interface. Multiplying the bus will result in a greater need for bandwidth of for instance the main memory.

The second option is useful if the bandwidth of the main memory cannot be increased as much as the

number of hardware threads, but it can be increased. The mss buffer will sequentialize every request but the requests will be divided among the bus interfaces.

The last solution is a full connected bus interface, without mss buffer. This is the fastest solution. However, the probability that all connected devices, such as the main memory, are able to multiply their bandwidth dramatically is small.

### E. Quantum Time Expiration

In this implementation of interleaved multithreading switches between hardware threads occur at first level cache misses. The advantage is that the knowledge that a switch needs to be performed is quickly accessible. This decision will result in switches when there is a second level cache miss, with relative few stall cycles. To compensate for this the implementation of the switch is made to only cost one cycle, which is achieved in [11].

The first level cache miss is not the only reason to switch. It can occur that a hardware thread enters a continuous loop because of synchronization mechanisms *busy waiting*, which will be explained in detail in the next section. One hardware thread may be running a task that waits for a reply from another task, running on the same CPU, but on a different hardware thread. In this case the task just waits for the reply in a continuous loop, but the other task will never become active, since there is no first level cache miss.

To prevent this we must either implement a *quantum time expiration*, a fixed maximum number of cycles a hardware thread may be active before a switch must follow or we must identify all situations when a switch needs to be forced. The latter seems nearly impossible. Detecting a continuous loop can be done if the loop body is not too large by storing and checking the current instruction address. However, the loop body may become too large, which is why a quantum expiration is implemented. The switch algorithm is illustrated in Figure 4.

Ideally the value of the quantum expiration is a huge number of cycles. Every time a switch is forced because of quantum time expiration (QTE), cycles are unnecessary wasted. Either the switch is too early and the hardware thread is just interrupted while performing its tasks or the switch is late and the continuous loop has been executed a few times. It depends on the number of such busy waiting loops. If they are many, then the QTE must be small. If a busy waiting loop is an exception, then the QTE must be heightened. The optimal value must be determined by performing experiments.

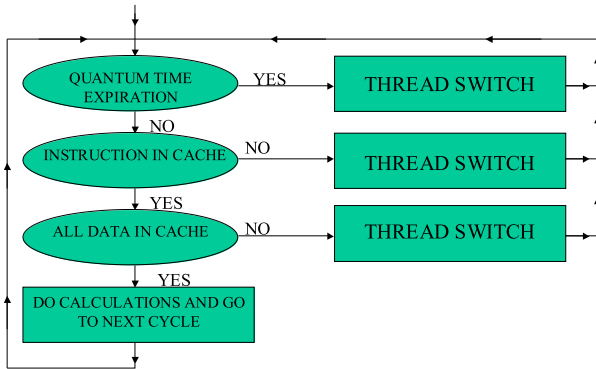


Fig. 4. Events that cause thread switches in the multithreaded TriMedia.

#### F. Switching During Synchronization Between Hardware Threads

To guarantee vertical exclusion, a synchronization method must be implemented. In the TriMedia this is achieved using two instructions: load link (LL) and store conditional (SC).

These instructions can simulate an atomic operation or to acquire a semaphore. In the code below an example of an atomic add simulation is given for the TriMedia. The SC operation fails if the memory location which is loaded by the LL instruction is changed by another processor. Therefore a switch when the task is in a LL and SC loop is dangerous for the functionality for the atomic add simulation. Every switch between LL and SC instruction will automatically guarantee a fail of the SC instruction in the future, resulting in a second pass of the LL and SC loop.

```

LOOP:  IF R1 IIMM(LOOP) → R68
        IF R1 LL32 R64 → R65
        IF R1 IADDI(1) R65 → R66
        IF R1 SC32 R64 R66 → R67
        IF R67 IJMPI(NEXT)
        IF R1 IJMPF R67 R68
  
```

#### G. Software Enhancements

Though this research focuses itself on hardware changes, this section deals with software enhancements that could improve the performance of interleaved multithreading.

At least the bootcode of the TriMedia has to be altered. All the hardware threads try to get their individual number, but this does not happen at the same time. To make sure that every hardware thread has accessed its number to identify itself an intentional cache miss or switch has to be performed after a hardware thread has acquired its CPU number. Changes can be made in

the compiler to speedup performance. Every time the compiler recognizes that a switch needs to be forced it can add an intentional miss, which forces a switch. As said before, not every forced switch can be recognized, but adding these changes will make the optimal QTE value higher, which will improve performance. We cannot remove the QTE completely, unfortunately. This intentional miss can be implemented using two existing instructions, who invalidates a line and then fetches the same line. A second option is to add a complete instruction to the existing instruction set. This option has the disadvantage that additions in hardware must be made and compatibility is lost. These disadvantages do not way against the gain of one cycle per forced switch using existing instructions. A final change could be an adjustable QTE for optimal performance per program.

#### V. DESIGN SPACE EXPLORATION

CAKE is the abbreviation of Computer Architecture for a Killer Experience. CAKE is a multiprocessor architecture [8]. It addresses the need scalable architectures. This means that an interconnection network connects multiple processors and peripherals and other tiles with the same configuration. We used a CAKE cycle accurate simulator to simulate the system on chip (SoC). It had a adjustable number of MIPS, coprocessors and TriMedia. Only the last option was altered to have the capability of interleaved multithreading. Two different MPEG 2 decoders (*opt-mpeg2dec* and *normal-mpeg2dec*) were used as benchmarks. *Opt-mpeg2de* is an MPEG 2 decoder optimized for the TriMedia, whereas *normal-mpeg2dec* is an off the shelve MPEG 2 decoder. Simulations were run using different cache sizes, QTE and the number of hardware threads with bus interfaces. The number of hardware threads is focused on two and three hardware threads, because of the limitations of the simulator.

Figure 5 shows the results for a multithreaded TriMedia running the program *opt-mpeg2dec*. This TriMedia has the original cache size of 16 KB for the data cache and 32 KB for the instruction cache. We see an optimal value of around 90 cycles for the QTE. This value is very small, which shows that there are many occurrences of busy waiting loops. With an ideal QTE of infinite, this value is disappointing and justifies additional software enhancements to detect loops at compile time. If the QTE is set at a value too small or too high, the performance of the multithreaded TriMedia is even worse than the original one without the multithreading overhead.

The normalized cycle count is disappointing as well, with a maximum gain less than 5 percent. These result must be seen with the note that the program *opt-mpeg2dec* had a good CPI to begin with, which leaves

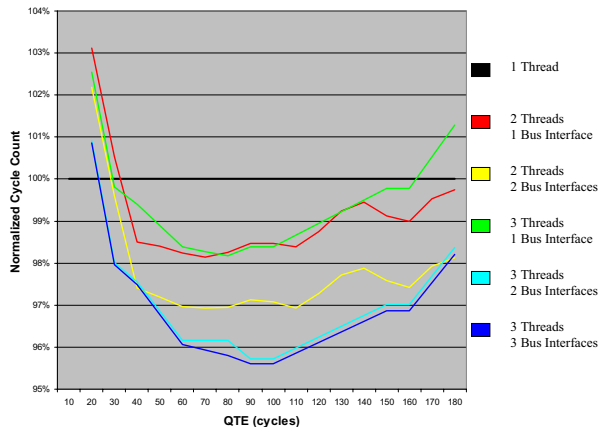


Fig. 5. The relative change in cycle count, when using different implementations of multithreading, for the program `opt-mpeg2dec` as a function of the QTE.

little room for improvement. Another conclusion we can draw from figure 5 is that a third bus interface with three hardware threads is not a significant improvement.

Further experiments with smaller cache sizes gave more improvement, than seen with the original cache size, as one may expect. A maximum gain of around 15% was found with a data cache of 8 KB and an instruction cache of 16 KB. However the optimal value of QTE reduces, which indicates an even worse effectiveness of multithreading. If we increase the cache sizes the results show an even worse scenario, with the original TriMedia performing almost always better than the multithreaded version.

The program `normal-mpeg2dec` is an off the shelf MPEG 2 decoder, which has more room for improvement than the program `opt-mpeg2dec`. As with `opt-mpeg2dec`, we first run a simulation with the original parameter. The results of this test are shown in figure 6. As supposed to the optimized MPEG 2 decoder, this version has a substantial improvement with 2 hardware threads and 2 bus interfaces: 3% against 27%. This increase in effectiveness is also seen in the heightened optimal value of QTE. From 90 cycles to a value of around 140. Keep in mind that this is still low, compared to the cycle count of around 639 million of the program. We also see that the performance is less dependent on the QTE than when program `opt-mpeg2dec` was executed. Here the line inclines less steep after reaching the optimal QTE.

Decreasing the cache size does not aid the multithreading in this program. Apparently multithreading is not capable here to use the additional cache misses to increase its effectiveness. Doubling the cache sizes has only a small beneficial effect.

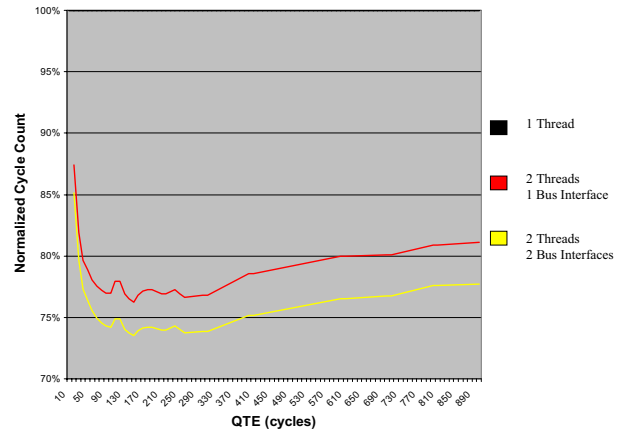


Fig. 6. The relative change in cycle count, when using different implementations of multithreading, for the program `norm-mpeg2dec` as a function of the QTE.

## VI. CONCLUSIONS

In this paper we have evaluated the efficiency of implementing interleaved multithreading in the TriMedia. Several micro-architectural changes are required to enhance the TriMedia with the capability of interleaved multithreading. Caches and functional units need to be shared and the general purpose registers and MMIO need to be duplicated. Furthermore, the state of the TriMedia has to be stored after a switch to another hardware thread to be able to continue from the point the switch was made. This means either copying the pipeline or flushing it, which is slower.

We have proposed to use a pending buffer. This buffer guarantees that two or more hardware threads do not request the same cache line. Furthermore there is a copy back buffer that ensures that the copy back of a line in the cache is finished before that line is requested again by a hardware thread. The third buffer is optional and dependent on the choice of implementation. Every request for data is sent to main memory or other locations through a bus interface. With multiple hardware threads we can have multiple outstanding requests simultaneously. If we do not want to add more bus interfaces, we use a so-called memory subsystem buffer (mss buffer) to sequentialize the requests on the bus interface.

Switches between hardware threads occur in a round robin method at every first level cache miss and at every quantum time expiration (QTE), whichever comes first. The QTE is a maximum amount of cycles a hardware thread may be active to exclude the possibility of a hardware thread being active all the time.

The simulation results have shown that using interleaved multithreading only provides a significant performance improvement when a program is executed

that is not optimized for the TriMedia processor. This improvement is not very high as the QTE has a relatively low value, while a high value is desired. Interleaved multithreading can improve VLIW processors with generic programs, but the hardware changes that need to be made and the increase in chip size, make implementation efforts not worthwhile. We, therefore, recommend further research in changing the software/compiler to enhance the performance of interleaved multithreading in VLIW processors.

#### REFERENCES

- [1] Weber, W. Gupta, A., "Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results," *Proc. 16th International Symposium on Computer Architecture (ISCA'89)*, (1989): 273–280.
- [2] Gupta, A. Hennessy, John. Ghacrachorloo, Kourosh. Mowry, Todd and Weber, Wolf-Dietrich., "Comparative Evaluation of Latency Reducing and tolerating Techniques," *Proc. 18th Annual Symposium on Computer Architecture*, (May 1991): 254–263.
- [3] Smith, Burton J., "Architectures and Applications of the HEP Multiprocessor Computer System," *SPIE*, (1981): 298:241–248.
- [4] Tullsen, D.M. Eggers, S.J. and Levy, H.M. "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 20th Annual Symposium on Computer Architecture*, (June 1995): 278–288.
- [5] Culler, David E. and Pal Singh, Jaswinder., *Parallel Computer Architectures, a Hardware/Software Approach*, (Morgan Kaufmann 1999).
- [6] Marr D. Binns, F. Hill, D. Hinton, G. Koufaty, K. Miller and J. Upton M., "Hyper-threading technology architecture and microarchitecture: a hypertext history.", *Intel Technical Journal*, (February 2002).
- [7] Rathnam, Selliah and Slavenburg, Gert., "An Architectural Overview of the Programmable Multimedia Processor, TM-1," *Proc. Compcon '96*, (February 1996).
- [8] Stravers, Paul and Hoogerbrugge, Jan., "Homogeneous Multiprocessing and the Future of Silicon Design Paradigms," *Proc. International Symposium on VLSI Technology, Systems and Applications (VLIS-TSA) 2001*, (April 2001): 184–187.
- [9] Hennessy, John and Patterson, David., *Computer Architecture, a Quantative Approach* (Morgan Kaufmann 1996).
- [10] Kreuzinger, Jochen and Unger, Theo., "Context Switching Techniques for Decoupled Multithreaded Processors," *Proc. Euromicro'99*, (September 1999): 248–251.
- [11] Nuth, Peter R. Dally and William J., "A Mechanism for Efficient Context Switching," *International Conference on Computer Design*, (October 1991): 301–304.