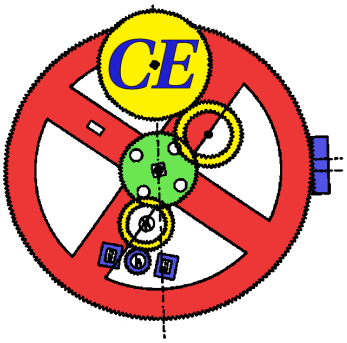

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2004

MSc THESIS

Performance Evaluation of Interleaved Multithreading in VLIW Architectures

Stephan Suijkerbuijk

CE-MS-2004-01

Abstract

Multithreading is a well known method to increase performance. In this research we set out to determine the performance improvement of interleaved multithreading in the multimedia processor called TriMedia. We limit ourselves to the hardware changes that are needed. We do not use simultaneous multithreading, because we set out to use the cycles that the processor stalls and this multithreading technique is not achievable with only hardware changes in a VLIW processor. In interleaved multithreading, the processor switches to another task when a long memory latency operation occurs. These operations consist of all operations that do not have all necessary data in the primary cache. First we present the architectural changes that are needed for an optimal implementation of interleaved multithreading. We determine that there should be a maximum amount of cycles a thread is allowed to be active. After this amount a thread switch will be forced. Our architectural changes are applicable to every VLIW processor, though the simulation results will vary from processor to processor. The benchmarks show us that multithreading leads to a significant speedup of around 25%. We show that the cost in chip size is small enough (21%) to justify the implementation of a multithreaded architecture and it justifies further research in this field. Further research should focus on software enhancements, such as a special compiler.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Performance Evaluation of Interleaved Multithreading in VLIW Architectures

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Stephan Suijkerbuijk
born in Bergen op Zoom, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Performance Evaluation of Interleaved Multithreading in VLIW Architectures

by Stephan Suijkerbuijk

Abstract

Multithreading is a well known method to increase performance. In this research we set out to determine the performance improvement of interleaved multithreading in the multimedia processor called TriMedia. We limit ourselves to the hardware changes that are needed. We do not use simultaneous multithreading, because we set out to use the cycles that the processor stalls and this multithreading technique is not achievable with only hardware changes in a VLIW processor. In interleaved multithreading, the processor switches to another task when a long memory latency operation occurs. These operations consist of all operations that do not have all necessary data in the primary cache. First we present the architectural changes that are needed for an optimal implementation of interleaved multithreading. We determine that there should be a maximum amount of cycles a thread is allowed to be active. After this amount a thread switch will be forced. Our architectural changes are applicable to every VLIW processor, though the simulation results will vary from processor to processor. The benchmarks show us that multithreading leads to a significant speedup of around 25%. We show that the cost in chip size is small enough (21%) to justify the implementation of a multithreaded architecture and it justifies further research in this field. Further research should focus on software enhancements, such as a special compiler.

Laboratory : Computer Engineering
Codenumber : CE-MS-2004-01

Committee Members :

Advisor: Paul Stravers, Philips Research, Eindhoven

Chairperson: Stamatis Vassiliadis, CE, TU Delft

I dedicate this thesis to all the people who gave me help, support, friendship and love

Contents

List of Figures

List of Tables

Acknowledgements

This research was performed at Philips Research in Eindhoven. In this section I would like to thank all the people who made it possible for me to do my research.

I would like to thank my supervisor at Philips Research Dr. ir. Paul Stravers and my supervisor at the TUDelft Prof. dr. ir. Stamatis Vassiliadis, for giving me the opportunity to work at Philips Research and to do my graduation there. My colleagues, who stood by me with counsel and advise. Without them I could not have gotten past all the difficulties I found along my path.

I would also like to acknowledge my colleagues, who I would like to call friends: all the current Coffeeclub members, all the former Coffeeclub members, and many more. Their company gave me relief at a hard days work.

Special thanks to Jayram Nageswaran and Andrei Terechko, who provided me with the most support and guidance. They were always willing to spend time to help me solve difficulties in the research. I am also in debt to Jayram for giving me the opportunity to live in Eindhoven.

Thanks to Jeffrey Kang for revising my thesis as well as Jayram Nageswaran and Andrei Terechko.

To my family and friends, who stood by me with support, drive and computers to aid me to finish my study and complete my thesis.

To all.....thank you

Stephan Suijkerbuijk
Delft, The Netherlands
April 23, 2004

1 Introduction

This first chapter will give some background information to the subjects discussed in this thesis. The first part consist of a brief introduction to the TriMedia processor. Then we will explain the properties of multithreading, both interleaved and simultaneous. The final part of this chapter will give the problem definition for this thesis and how the structure is built up.

1.1 Multithreading in Hardware

Multithreading is already quite well known in the world of computer engineering. The idea is simple. Every processor needs data to do its calculations. When a processor core does not have the data in the cache, the data needs to be fetched. This data can reside in far away locations, such as remote main memory or in off chip caches. During the time that the data is being fetched, the processor stalls. The time that the processor is stalled is dependent on the location of the data which is needed. If it is in the second level cache, the stall time is much smaller than if the data is located much further from the CPU. Let us take the example of a so called multi-tile environment. In this environment there are several chips (tiles), with one or more CPUs, which are connected. They have the ability to exchange data with each other. It takes a lot of time before to exchange data between tiles since the tiles are relatively far apart. In the mean time, from the moment the request for data is made, until the data is received, the requesting processor only waits. This is called stalling. Multithreading is a way to use that stall time to do some other useful job. In stead of having only one core and state of that core (current operation, registers), a multithreaded processor has more cores, which we call hardware threads. In addition to the hardware threads, there are so called software threads or tasks. Each software thread is program code which can be executed independently from other software threads. This can be multiple programs or pieces of a program that can be executed in parallel. For instance one hardware thread can be active and executing the task that is mapped to that hardware thread. When this hardware thread needs data, it stalls while the data is being fetched. In the mean time there is a so called context or thread switch, which means that another hardware thread containing another task gets active and executes its task. At the moment when the data arrives for the first task, it can continue without really being idle in the mean time. Do not confuse a thread switch in hardware with the thread switch in software. The switch in software means that the software thread is finished and another thread needs to be mapped on the hardware thread. The hardware thread switch changes to another hardware thread already containing a software thread. The hardware switch is hereafter called thread switch. There is also the possibility to have multiple hardware threads active in parallel. This is called simultaneous multithreading.

In the past multithreading was considered to be a too complex method for tolerating the stall time of the CPU. This was because the changes that had to be made to the architecture of the existing types of processors were quite extensive. The gain that multithreading gave to the latency tolerance was simply not high enough to justify the time and cost, which needed to be invested in the development of such a processor. During the years several events caused that to change.

First there is the so called memory gap. In computer technology everything gets faster. The CPUs can do its computations faster which means that it needs the data faster as well. We see a trend in the technology that the memory bandwidth, the ability to provide data to the CPU, grows not as fast as the processing speed. The consequence of this is that the time it takes for data to be given to the processor gets relatively bigger. In that time during which the CPU is idle, more calculations can be done. We call this effect the memory gap. The memory gap continued to grow, which made the need for memory latency tolerance techniques bigger and bigger. Memory latency tolerance techniques are techniques that will make sure that the time the processor stalls is used more effectively or that that time is decreased. Especially multiprocessors have a big memory latency problem when the data is in the cache of another processor.

Besides this trend, some changes in the processor architecture for other memory latency tolerance techniques were expandable for multithreading and the research in 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.

1.2 Cycle Penalty of Thread Switching

There are some different switching techniques with different cycle penalty. If we take the cycle-by-cycle interleaving, such as the HEP[9], the cycle penalty must be next to nothing, since there is a thread switch every cycle. There are some conditions to efficiently use this method. For instance if the pipeline has seven stages, then there must be 7 executable threads. This had the consequence that when that instruction finishes, the next instruction can be fed into the pipeline immediately. But in reality this is not efficient, since there must be a minimum of executable threads. With a big pipeline that is not always the case. Another method is called block interleaving [4]. This is the method which is used in this research (interleaved multithreading) as will be explained below. This technique uses signals to go to a switch. This signal can come when a large latency operation is detected. The problem here arises how soon it is known that such a signal should be given. We do not know if there is a cache miss in the beginning of the pipeline. There are a few possibilities to handle the pipeline stages that are already filled. We can throw them away, which means a waste of valuable cycles. We can let the pipeline execute till it is empty. This flushing method has the disadvantage that more and more stages of the pipeline are emptied until it is cleared completely. Then there is the way to store the pipeline stage in its full form. If we can quickly store the stages, the efficiency goes up but the chip size will increase, because we need to store the entire pipeline of that hardware thread. We will to use the pipeline state again when the hardware thread becomes active. In this interleaved multithreading the cycle penalties

vary heavily in every implementation. Simultaneous multithreading is a technique used for superscalar processors. It is difficult to implement this on a VLIW CPU. In SMT every thread run at the same time, sharing functional units. So there is no penalty for thread switching.

1.3 TriMedia Processor

The TriMedia is a multimedia processor developed by Philips Semiconductors [7]. Examples of multimedia applications are MPEG2 encoders and decoders. The TriMedia is a VLIW processor with 5 instruction slots. It can be used as a co-processor for multimedia calculations, because it has a special designed instruction set for multimedia applications, but it can work just fine as a stand-alone processor. A VLIW processor can do multiple operations at the same time, because of multiple instruction slots. These slots are then connected to different functional units. A compiler for a specific VLIW processor makes sure that the instructions are distributed among the slots. The compiler takes into account that some operations need to be sequential and that not every instruction slot can do every operation. The TriMedia core has 128 general purpose registers [8], which makes it a big part of the chip size of the TriMedia

In general the VLIW instructions work as followed. Before the core receives instructions it can handle, it must decode the data that it receives. This data contains every instruction for every slot, but in compressed format. After the decoding, we have a format of instruction it can handle. Of course the maximum instructions in one cycle is five. The minimum is zero. This is also called NOP (No OPeration). The compiler issues a NOP if it needs data that is not available yet. The compiler knows how many cycles every instruction costs. With that knowledge it can issue NOPs in such a way that it can use the correct, just calculated, value of a register. The compiler must also take into account that not every instruction slot can do every operation. For instance, there can be only 2 load/store operations in one cycle because there are only two load/store units, and not five. For more information about all the instructions, see [8].

A TriMedia processor is usually part of a system on chip (SoC) with more processors and coprocessors. An example of such a SoC is the Philips PNX8525 1.1. Here we see the TriMedia core along with several input and output interfaces, coprocessors and other processors. We see that the system has an external connection to the SDRAM through a bus and a main memory interface. All the components are also connected through buses.

Other important parts of the TriMedia are the caches and memory hierarchy. There are two types of caches: the data cache and the instruction cache. In TM1000 (a version of the TriMedia) there is a data cache of size 16K bytes. It has an 8-way set-associativity and a block size of 64 bytes. When the cache is full, the incoming line replaces another through LRU replacement policy. LRU stands for Least Recently Used. The cache line that is used/read least recently is replaced. When the required data is not available the CPU either fetches it from the on-chip peripherals or it goes to the Main Memory Interface which is connected to the SDRAM Main Memory. The TriMedia is designed for multimedia purposes, it has a main-memory bandwidth of 400 MB/s. A high bandwidth is needed to supply the CPU with the multimedia data-streams for calculations. Since

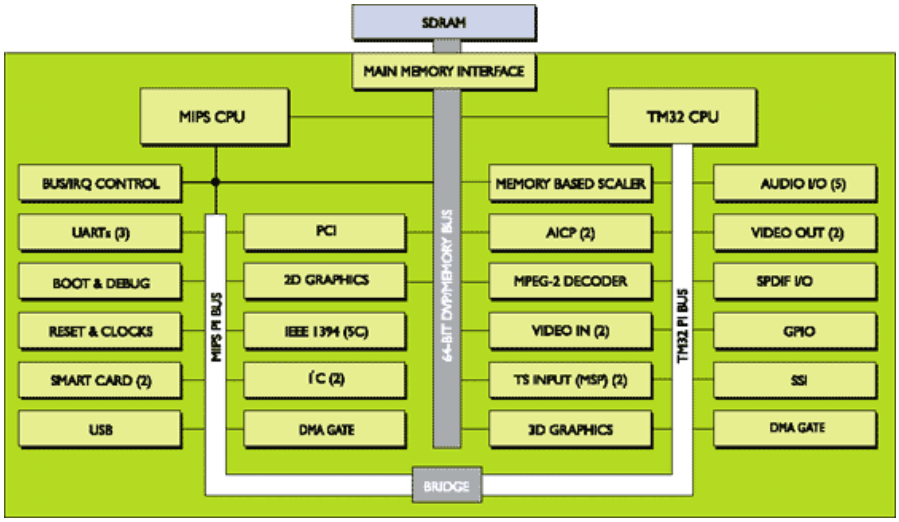

Figure 1.1: The Nexperia system on chip with a TriMedia (TM32) along witch coprocessors and more components.

the CPU has a high requirement for data for processing, the SDRAM, with its 400 MB/s bandwidth, cannot give data that fast. But this is a lesser problem when the multimedia algorithms use locality of reference. This means that the data in the memory that is near the address of the data that is needed is fetched to the CPU data cache as well. The instruction cache has a size of 32K bytes. It has an 8-way set-associativity and a block size of 64, just as the data cache. In this cache the instruction are stored in compressed format. This means that more instructions can be stored, with the cost of decoding them before delivering them to the CPU. An overview of the global TriMedia processor is given in figure 1.2.

A VLIW processor is suitable for multimedia applications for two main reasons. There is a lot of instruction level parallelism (ILP) in multimedia applications, which means that many instruction can be executed simultaneously. Multimedia applications typically have a lot of the same instructions on a large piece of data. A lot of these instructions can be done in parallel. In a VLIW processor there are more instruction slots, which can be executed in parallel. Furthermore there is a low power usage, because there is no hardware scheduling in the processor. A hardware scheduler checks dependencies between instructions and it schedules them at run time. A VLIW processor, such as the TriMedia, does the scheduling of the instructions at compile time with software in stead of hardware.

1.4 Thesis Description

The TriMedia is a multimedia processor. In this processor there is no hardware multi-threading of any kind. This thesis describes the research into interleaved multithreading in the TriMedia processor. It will be determined if interleaved multithreading is a cost effective and efficient method to be implemented in future TriMedia processors.

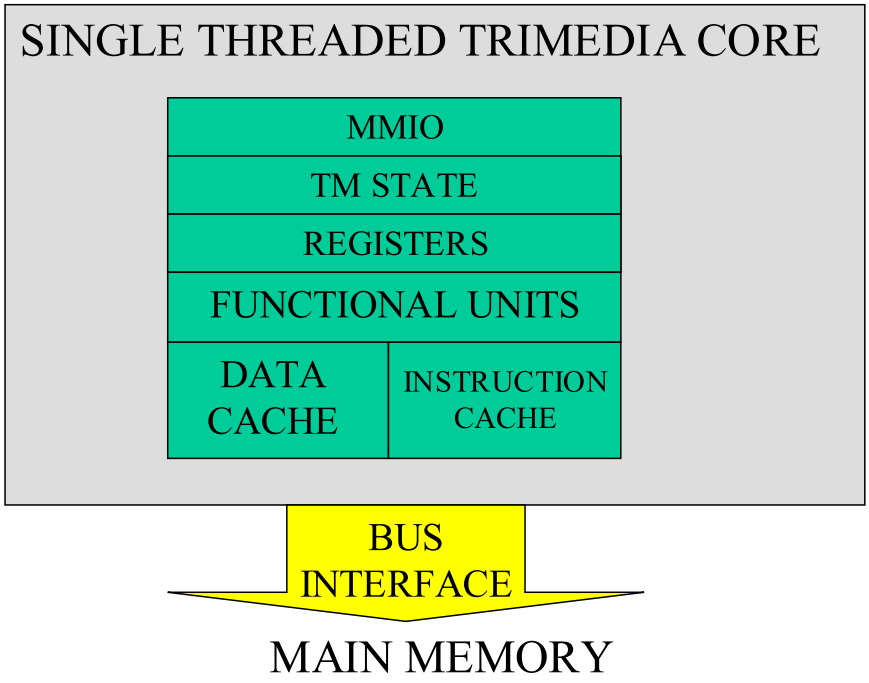

Figure 1.2: Global architecture of the TriMedia. An high speed bus interface connects the core to the main memory

In chapter 2 we will give an architecture proposal for the enhancements for the new TriMedia. We will point out the architectural changes which are needed for interleaved multithreading to be implemented in an optimal way. In chapter 3 the results are shown for the different tests which we have done to determine the performance of the new architecture. In the final chapter we will give the conclusions and recommendations for future work.

Integrating Multithreading in a VLIW Architecture

2

In this chapter the architectural changes are discussed to create a multithreaded TriMedia using the normal TriMedia as a starting point. First there will be a motivation for this architecture and the choice of interleaved form of multithreading. Then the changes are discussed to implement interleaved multithreading in an optimal way. The the structure of the caches and the buffers that are needed is handled, as well as difficulties of the context switches. Then we take a look at the synchronization issues with multiple hardware threads. We conclude this chapter with a look at the possible boost in performance software enhancements could give this multithreaded architecture.

2.1 Interleaved and Simultaneous Multithreading

The type of multithreading which changes hardware threads when it encounters a long memory latency operation because of lack of data, is called interleaved multithreading. For instance a data fetch to a location relatively far away, such as the slow main memory. The advantage of this approach is that we need only a few threads to hide the long memory latency. This is shown in figure 2.1. Here we see that when the processor is

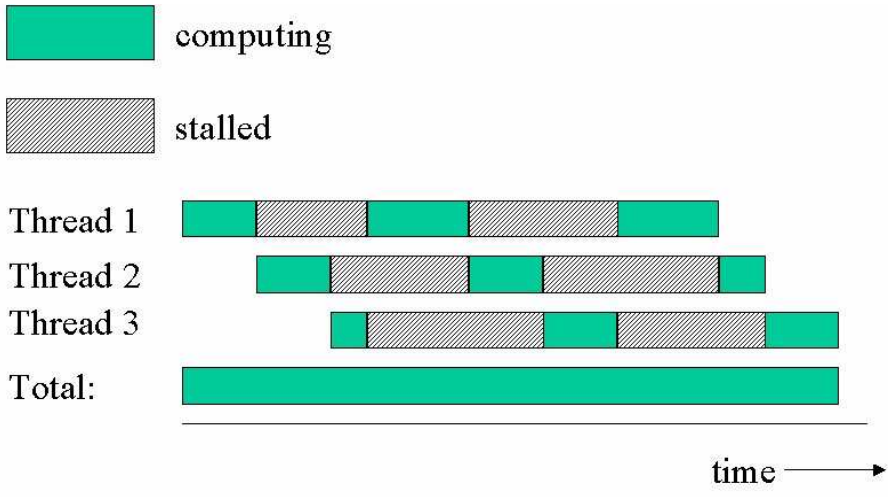

Figure 2.1: The workload of a multithreaded processor

stalled, the context is changed and the CPU goes to work on the next software thread. In this example the context is switched when there is a long memory latency operation. This is a form of interleaved multithreading. In interleaved multithreading, there is only one task active at the time. This in contrast to simultaneous multithreading, where more tasks are active.

Another form of interleaved multithreading is to change threads every cycle. This is implemented in the Denelcor HEP [9]. In this case every thread has the same priority in execution and every thread is being handled. The disadvantage is however, that there needs to be a lot of threads to fill in the gap caused by an operation that needs data which is not in the cache. The switching mechanism needs to be fast as well. In figure 2.1 we assume that the cost in cycles to switch to another thread is non-existent. Usually there is a cost in cycles which is dependent on the implementation of the switching technique. This has an impact on the efficiency of multithreading. The cost must be significantly lower than the time it takes to fetch data from the memory.

We just said that interleaved multithreading switches to another software thread when it encounters a long memory latency operation. But there is also the issue to which hardware thread we need to switch to. In the example of figure 2.1 the switching method is round-robin. Every time the CPU needs to switch to another hardware thread, it takes the next hardware thread, running another task. Eventually after several hardware thread switches, the first hardware thread becomes active again. In this method every task has the same priority. We can assign priority to tasks, with respect to their importance and hard deadlines (real time applications). When there is a hardware thread switch, the CPU will take the hardware thread which handles the task with the highest priority. Without prioritization the overhead of the switching method and thus in the switching costs are diminished. On top of that we have the guarantee that every task is handled. However, prioritization has some advantages as well. There might be a task, which does not do anything else but wait for an event. In this case the task can be executed without cache-misses, but it does not speed up the overall calculations of the workload. When we prioritize, this task is assigned a low level. In effect, the tasks which are not in the critical path are handled last. The danger arises that a task has a low level and is never handled. This can happen when a task is given a too low level for its job.

Simultaneous multithreading is multithreading without context switching. Every hardware thread has a task that runs on the processor in parallel. This means that a hardware thread does not have to wait to start its job. Every hardware thread shares the resources on the processor with the other hardware threads. This type of multithreading is faster, since it utilizes more of the resources available [11]. In a very large instruction word (VLIW) processor it has the best performance when we look at figure 2.2 which comes from [1]. But simultaneous multithreading does not belong on a VLIW processor, but on a superscalar processor. In a VLIW we would get conflicts of resources with simultaneous multithreading, because we do not know the cache misses and latencies of instructions a compiler time. Therefor we do not know if a functional unit is available or not. An implementation of simultaneous multithreading on a superscalar processor is Hyper-Threading, which is used in the Intel Xeon processor family [5]. In this implementation the execution units, branch predictors, control logic and buses are shared. Everything else that is needed is duplicated. According to [5] the added relative chip size is 5%. This is a disadvantage when we consider to implement multithreading. There is also time and money to be invested to implement the multithreaded architecture. This is what we call the cost to achieve a speedup of in this case 25%. In Hyper-Threading the software sees one processor as two logical cores. In simultaneous multithreading the

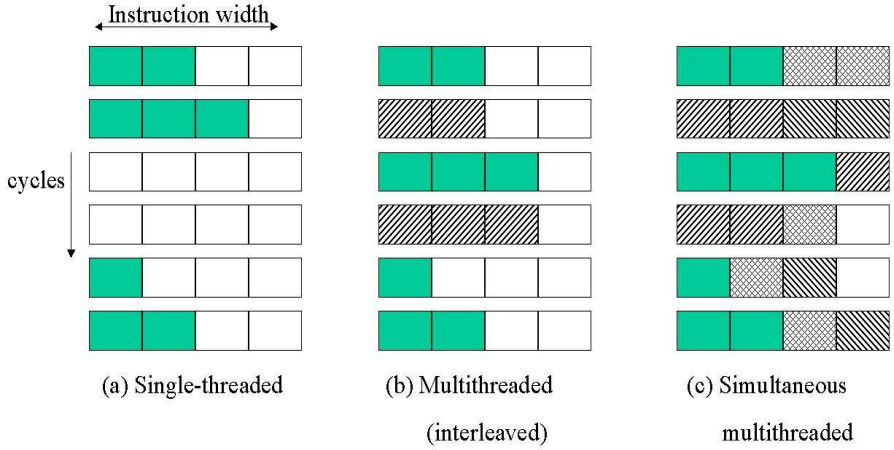

Figure 2.2: Simultaneous multithreading. The potential improvements are illustrated for both simple, interleaved multithreading and simultaneous multithreading for a four-issue processor. Shaded and patterned boxes distinguish operations from different threads, while blank boxes indicate empty slots in instructions.

difficulties lie among others in limiting the overhead, sharing conflicts of the resources, cache coherence and increase in chip size. If the increase in size is too much, the effective performance boost might not be enough to justify the extra chip area and the implementation of a new technique, which cost time and money.

2.2 Preliminary Performance Improvement Investigation

When we look at any single threaded processor, it stalls when it needs data that is not there. The data needs to be fetched from other locations, such as the relative slow main memory. The TriMedia is no different in this aspect. The main idea of this thesis is to use this idle time more effectively. Multithreading is one way to do that. We chose multithreading as the idea to increase performance in the processor because it was not yet implemented in the TriMedia processor. It is interesting to find out what the effects and the speedup would be.

We need to decide which type of multithreading we should use. The implementation of one cycle thread switch as the HEP [9], is deemed not effective enough because every pipeline stage of the processor needs to be filled by another hardware thread. In this case we would have to have a minimum number of software threads (the same amount as the number of pipeline stages) to be effective. Furthermore the cache miss stall time will not be completely filled by the thread switches. Simultaneous multithreading will be more complex if not impossible to integrate in the current VLIW architecture, because of the conflicts of the shared resources, such as the functional units. It also requires a special compiler. The compiler has to assign operations of different software threads to the VLIW instruction slots. But the compiler has to keep in mind that not every

instruction slot can handle all or the same operations. The problem then arises that the compiler does not know when an instruction slot of the VLIW processor is available, because it cannot predict the latency of that operation. In this research we emphasize the hardware changes, which are needed. This leaves us interleaved multithreading to be the best solution.

Let us take the a situation when the cache has a miss rate of 5% and that every miss causes the processor to stall for 30 cycles. In a perfect multithreaded environment these cycles would be filled up by other threads completely. We use 2.1 to calculate the stall cycles per instruction for the single threaded processor.

$$SCPI = MR \times ST \tag{2.1}$$

In formula 2.1 *SCPI* stands for the stall cycles per instruction, *MR* is the miss rate and *ST* is the stall time. In the example the stall cycles per instruction equals $0.05 \times 30 = 1.5$. This we can compare with no stall cycles per instruction for a perfect interleaved multithreaded environment. This is a big enough difference to make it worthwhile to research the possibilities for a multithreaded TriMedia. But this memory latency tolerance technique comes at a cost.

To implement architectural changes on top of the old, the chip size will increase. To let the TriMedia have more hardware threads to store the states, there are some objects that have to be duplicated, such as the registers. Furthermore the thread switch has to be controlled. This and other control we have to implement because of multithreading (overhead) will cost logic. This means taht the chip will increase in size. If the size increase is too big, it might be better to use two already built TriMedias in a multiprocessor platform, instead of investing time and money in a multithreaded version. The performance of such a multithreaded CPU can be dependent on a lot of factors such as cache size and applications. When the cache is smaller, the size of the chip decreases but there will be more misses to be filled with thread switches. The application must have multiple threads to be performed. This research will investigate these performance dependencies. The software will see the multithreaded environment as a multiprocessor environment. This means that if there are 4 hardware threads, the software sees 4 separate CPUs. Furthermore we have the constraint that we emphasize on hardware and we try not to leave software changes such as a new compiler out of the picture.

2.3 Top Level Architecture

There are a lot of changes that need to be made to create an interleaved multithreaded TriMedia out of a single threaded one. We will not immediately go into the details, but first we explain the top level architectural changes that need to be made, with respect to the original CPU. We start with one Trimedia state, and we must expand so that we have the same amount of states as the hardware threads we want. These TriMedia (TM) states are then grouped in a multi state (multithreaded TriMedia). There is also the issue of duplication. What needs to be duplicated and what not.

The TriMedia has an instruction cache and a data cache. To save chip size, this implementation of interleaved multithreading has only one data and instruction cache.

This means that every hardware thread shares its cache with all the other hardware threads. With a simultaneous multithreaded CPU there would be an additional problem of ports. A cache does not have enough access ports to provide data or instruction to all hardware threads if they ask for it at the same time. You can either multiply the access ports or sequentialize the requests by using a buffer. With interleaved multithreading you do not have this problem, since only one hardware thread will access the cache at a time. The advantage in reducing complexity is that you will reduce in chip size and development time and thus money. Simultaneous multithreading needs overhead for cache coherence as well. In 2.4 cache coherency is explained further.

Since we do not increase the caches we have cache pollution. Every hardware thread has to share its cache with all the other hardware threads. Therefor the hardware thread has less space to put its data than the single hardware thread in a non-multithreaded processor. If all hardware threads use completely different data, then the data caches have to be duplicated for every hardware thread, to make sure that the performance for every hardware thread is equal when it would run without any other hardware threads with a dedicated cache. Since there is always some sharing among the hardware threads, this is not the case.

Every hardware thread within the multi state has its own registers. This is because every hardware thread within the encapsulating multi state must seem like a virtual stand alone CPU to the outside world. The compiler must know if the registers are shared or not. If the registers are shared, every hardware thread must be assigned an equal amount of registers. This approach decreases the amount of chip size, but it does not agree with the principle that the outside world must see virtual CPUs with the same characteristics of the TriMedia multimedia processor. Taking that into account, we chose to duplicate all 128 registers. This will also increase performance, since there are more registers. This does not mean that sharing the registers should be banned all together. The TriMedia instruction set is based on 128 registers. But if we can decrease that, it reduces the chip size. This approach might create a smaller chip with a better speedup, when using multithreading. In effect we get a better performance/chip size ratio. And then it might become efficient to implement it. This does requires a change in the compiler though.

Though the registers are duplicated, the functional units are not. Only one hardware thread is active and thus only one hardware thread executes its software task and needs functional units. If we have complex functional units who take up more than one cycle and there is a thread switch after the start of the calculation and before the end, we can use that unit anyways. Only the first result that comes out of the pipeline of the functional unit belongs to the other hardware thread.

The internal MMIO (Memory Mapped Input Output) registers are fully duplicated. There are some fields, which might be shared. But these fields are not that big and numerous and figuring out exactly what can be shared and not, will increase research time and complexity in the architecture. Both valid reasons to just duplicate the entire internal MMIO space.

There are also interrupt vectors. In the TriMedia they reside in the MMIO space, but this may not be the case for other VLIW architectures. The interrupt vectors need to be duplicated if they can be changed and used by the software. Then every software

thread can change the vector for their purpose and thus they need to be duplicated in every hardware thread.

The TriMedia state needs to be duplicated as well. This means all that is necessary to continue operation from the exact point from where it left off. The thread switch can occur when not all instruction of the VLIW operation are completed. These half done operation must be stored somehow, if the thread switch interferes with the execution of them.

2.4 Cache Structures

The hardware threads in an interleaved multithreaded TriMedia do not blindly do their work without looking at what the other does or has done. First and foremost they have a shared data and instruction cache. This has serious effect when considering cache coherency and speed. Coherency is accomplished if three conditions are met [3]. These conditions are given below for a multiprocessor system. Where it says processor in the list, we should read hardware thread if we want the rules to apply to a multithreaded environment.

1. If there is first a write to a location and then a read by the same processor to the same location gives the value written by that processor, if there are no writes to that location by other processors.

2. When there is a write to a location, the read to that same location by another processor must find the written value if there is enough time between the instructions.

3. Writes to the same location are seen by all other processors in the same order.

Where it says processor in the list above, we should read hardware thread. The first item is already handled by the cache controller. The writes do not have to be finished right away, but reads do have a high priority. The read after a write on the same address will always have the written value, because either it finds the write already finished or the write is pending to be written in the cache. In the last case it will read the value that is pending to be written in the cache. If the write location is in the cache the CPU does not stall, and no other hardware thread becomes active. But when there is a cache miss it does do a context switch and another hardware thread becomes active. Since this requirement does not take into account other writes by other hardware threads, the original hardware thread becomes active only after the data is in the cache. Thus the first requirement is met.

The second requirement is a bit trickier with multithreading. When there is a write to a location in the cache, there is no thread switch and thus no other hardware thread reads the same address before the write is finished. However, when there is a cache miss during the write, there can be a thread switch and it is possible that the next hardware thread reads the same address. However, if the address is not in the cache, the other hardware thread stalls too when it tries to read the same cache line. The only aspect we must consider is that the hardware thread that made the write request finishes first before the read of the other hardware thread starts.

abbreviation	*name*	*meaning*
M	Modified	The line has been changed, the memory has the old value
	Modified-Pending	The line will be changed and the change is passed to the other copies.
E	Exclusive-Clean	The line is only in this cache and the main memory, but it is not changed
S	Shared	the line is unchanged, but more copies are on other processors (only multiprocessor)
	Shared-Pending	After a cache-line miss, but before the transfer from main memory is completed
	Shared-Pending-Snooped	After a copy to the snoopbuffer and before the transfer starts
I	Invalid	The data in that line is invalid

Table 2.1: The mesi protocol states in the data cache of the TriMedia in a multithreaded or multiprocessor environment.

The fact that the write completes before the read of another processor starts proves the coherency for one multithreaded processor. In the next section it is explained how that is accomplished via a pending buffer. This pending buffer also makes sure that the third requirement is met.

Having a shared data and instruction cache between the hardware threads has also got the possibility of increasing the speed. The instruction that is already fetched and used by one hardware thread, can be used quickly by another hardware thread, because it finds the cache line in the instruction cache. The instruction cache must be large enough to make sure that there is not too much cache pollution. If one thread want to use a cache line, it is best that other threads won't have overwritten it with another line. We cannot prevent that from happening completely, because of the limits to the cache sizes. The same is true for the shared data caches. Data used by a hardware thread can be accessed by another hardware thread quickly and without fetching it from memory.

There is an additional speedup when we look at the MESI states. The MESI states are used to check what the status is for a cache line. The MESI states are listed in table 2.4. Usually MESI states are used in a multiprocessor environment. If a cache line is modified, then that line is the only valid one in the whole system. But in a multithreaded system, the MESI states are handled a bit differently. When a hardware thread puts a cache-line in Modified state, it puts it in that state for every hardware thread. This means that another hardware thread does not have to acquire exclusive write rights,

before writing to a cache line that has already been modified by a previous hardware thread. Cache consistency is thus strict, by definition, and easy to implement. For more information about the mesi states, we refer to [3].

The MESI states in 2.4 which contain the word "Pending" in the name are used to to support non atomic load and store operations, bus transactions, etc. The state changes do not occur immediately, thus the pending states signify that they are about to change. This is needed if something interferes with the change in MESI state.

When the data is fetched it changes the least recently used (LRU) status of the cache line. This also happens if the cache line is used. This LRU can be a common use parameter, or every hardware thread can hold separate information for every cache line usage. We start with the last option, when a cache line has the LRU status for every hardware thread. This gives the possibility that an hardware thread can remove a line which another hardware thread uses regularly. To bypass this problem we can give we can either give the hardware thread only write permissions on a hardware thread dedicated portion of the cache or use a common LRU. In this architecture a common LRU is used. Every time a line is used the LRU changes and every other hardware thread knows. If the cache is full and a cache line must be removed, it automatically removes the line which is least recently used by all hardware threads.

This is what the purpose of a shared cache in multithreading is. The data is shared, so other hardware threads can access and modify that data as well. If data is used regularly by another hardware thread than the one that fetched it, it stands to reason that another hardware thread wants to use it as well. Or at least this data is important enough to keep it in the cache for a while. Because of this we need to update the LRU and share this LRU with other and all hardware threads. Of course there is always the possibility of cache pollution. Data may be removed by one hardware thread that another hardware thread wants to use. This is inevitable. Even in a single threaded processor this problem remains, because of the limits of the size of the cache.

2.5 Buffering

When all the hardware threads operate sequentially, it is efficient to make sure that they do communicate to each other. In this section we will introduce the pending buffer. The principle is simple. The pending buffer prevents other hardware threads to ask for the same cache line. This means that the bus interface that connects the multithreaded CPU to the memory does not have to do the same request twice. Thus fewer data is transfered and the time that the bus interface does not handle the same second request can be used for something useful instead. We say that the bandwidth (the amount of traffic per time unit) does not get polluted. The memory bandwidth is the amount of traffic to and from the memory per time period. Since the memory has a limited bandwidth, requesting the same cache line twice is a waste of bandwidth; a pollution of the bandwidth.

The pending buffer consists of sets, tags and valid bits as shown in figure 2.3. Every hardware thread has its own group of set, tag and valid bit. The pending buffer is shared between all the hardware threads on the same multithreaded CPU. If a hardware thread has 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 tells the hardware thread that has asked for 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, except for the fact, that it checks if any hardware thread was waiting for the same cache line. If he finds that to be true, he tells that hardware thread that it can continue its work too, if it is his turn.

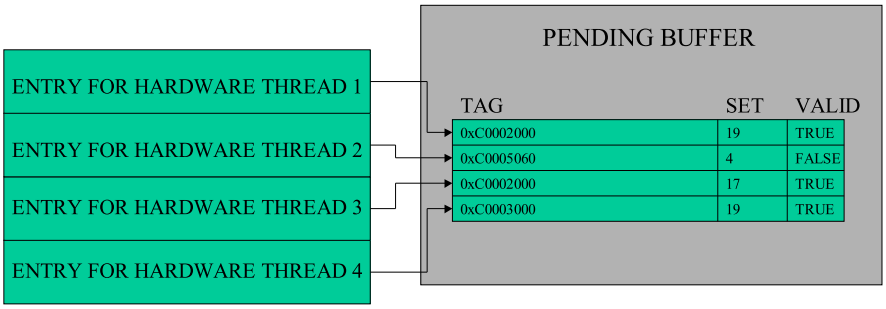

Figure 2.3: An example of a pending buffer. Every hardware thread has a dedicated entry to put his line that is being fetched. We need to look at the entire buffer if we want to see if a line is pending.

The communication between hardware threads is important when there is a copy back situation. If address A is in the cache, but hardware thread 1 wants to put another cache line in the cache on top of the one containing address A. The cache line which contains the address becomes invalid and the entire line is copied back to the main memory. The problem may now occur that another thread wants to access that cache line containing address A, which is supposed to be copied back. It does not find it in the cache because the line is invalid or already overwritten. It can happen that a copy back operation takes more time than the read. The bus interface can handle a read and write request simultaneously. This means that when a copy back request is issued, it can take a read request immediately. For safety reasons it is then better to wait until the copy back operation is fully completed. If we read the line, which is supposed to be copied back, we do not know what the results may be. Therefore a copy back buffer is created, which has the same structure as the pending buffer, as explained above, but not the address that is fetched is stored, but the address of the line that is copied back. If thus a read on that line occurs, the hardware thread waits till the operation is completed (there will be a context switch). Then it asks for the address again. Another possibility is to store the line which is copied back and give it to the second hardware thread if it asks for 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. By the time that that is made clear the copy back operation has already completed.

The TriMedia communicates to the main memory and other CPUs (when in a multiprocessor environment) through bus interfaces. The requests from the TriMedia are

given to the bus interface. The bus interface leads the request to its location. The issue arises in an interleaved multithreaded TriMedia, that there can be multiple requests to the bus interface. Every thread can ask for a different cache line. The bus interface in the TriMedia itself is not multithreaded. This gives us the options of making it possible to do all multiple request in parallel or to make a buffer to sequentialize the requests. There is also a combination of these two. With this option the bus can handle multiple requests, but not as much as the amount of hardware threads, which may be variable. This means that there is still need of a buffer, to sequentialize the requests that cannot access the bus interface just yet. We call this the memory subsystem buffer or mss buffer. If we first look at the option to do all possible requests in parallel. This means that every hardware thread needs its own bus interface. The chip increase is not that significant with a low number of additional bus interfaces, but there is another problem. The bandwidth must be increased. For instance if we have 4 bus interfaces to the main memory. We need to make sure that the memory can handle that with enough ports and four times the bandwidth. The other solution of using the mss buffer then becomes more interesting. This mss buffer consists of entries for all the data that is needed to make the request. Every thread puts the request in his own slot of the mss buffer. The possibilities are also shown in figure 2.4. Here we see a 5 hardware threaded CPU, with three possible bus interface implementations. There is the single connection, the variable connection and the full connected option. The variable connection can have any number of bus interfaces, from more than one to less than 5. In this case there are three. The most important thing when implementing this mss buffer is to make sure that there is no immediate connection between the bus interface and multithreaded CPU with a specific hardware thread active. Because if this was the case a hardware thread could be given information or data that is not meant for that hardware thread. The mss buffer checks all information to and from the bus. This buffer makes sure that the correct hardware thread is woken up, when a request is finished. In figure 2.4, we do assume that the main memory can sustain the increase in bandwidth. The secondary cache has a much higher bandwidth than location at a lower memory hierarchy. Chances are that the secondary cache can handle the increase in bandwidth, but the main memory cannot. The buffers and their properties are listed in table 2.5.

name	*description*	*size*
pending buffer	contains the line which is being fetched	one for each thread, containing tag, set and valid bit
copy back buffer	contains the line which is being copied back	one for each thread containing tag, set and valid bit
mss buffer	contains a request for the memory	one for each thread containing all the information to perform the request

Table 2.2: The buffers and their properties in the interleaved multithreaded TriMedia architecture

All the changes mentioned in this chapter can be found back in figures 2.5 and 2.6.

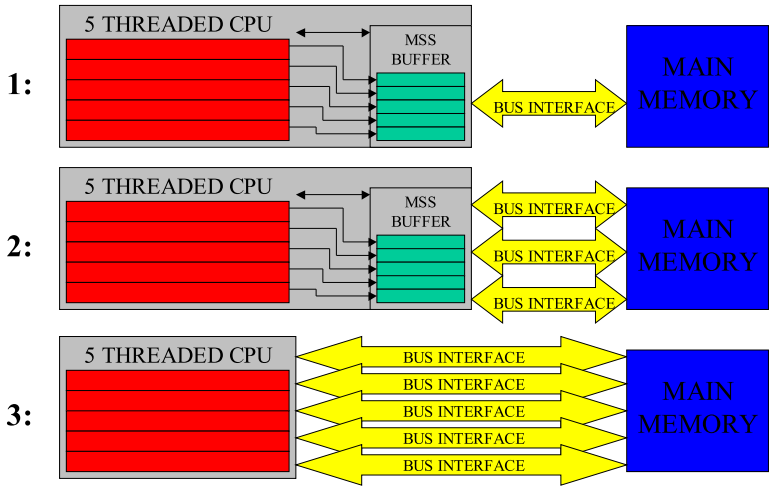

Figure 2.4: Three possibilities of bus interfaces to the main memory in 5 threaded CPU. 1: Single Bus Interface, 2: Variable Bus Interface, 3: Full Connected Interface.

Figure 2.5 is the original TriMedia architecture and in 2.6 we can see the changes and duplications that are needed. We can see clearly that the size will increase. Note that these images are not on scale. In this implementation of multithreading there are 4 threads and the memory subsystem has an entry for every thread. We can see the communication between the various blocks. There will be no request to the memory of a missed block when it is found in the pending buffer. If a request is made for a block that is located in the copy back buffer, the request waits till the copy back is completed.

2.6 Data Cache Controller and Multithreading

In figure 2.7 the flow chart of a data fetch for a store is given. This chart gives the normal procedure for a store to a certain address. One hardware thread tries to change a value of that location. First we check if the cache line containing that location is pending. The tag and set is determined and if we find the same values in the pending buffer, we know that another hardware thread has made a request for the same line. At this point we know that the data is not in the cache and we do a thread switch. If the hardware thread that made the request for the cache line finds that its request has been handled, it tells the thread that waited for the line that it can continue, when it is his turn. Since this cache line can be removed by other hardware threads before it is his turn to use it, the hardware thread goes through the flow the same way as it once started, before it found out that the data was pending. Instead of the pending buffer, the cache line can also be in the process of being copied back. If that is the case, just as with the pending buffer, the hardware thread stalls and switches. It can continue if it is his turn and another hardware thread has signaled that the copy back has been finished.

If the request of the hardware thread for data gets past the two buffers, it can check the cache. If the data is not there, we know that it must be fetched. But before we make

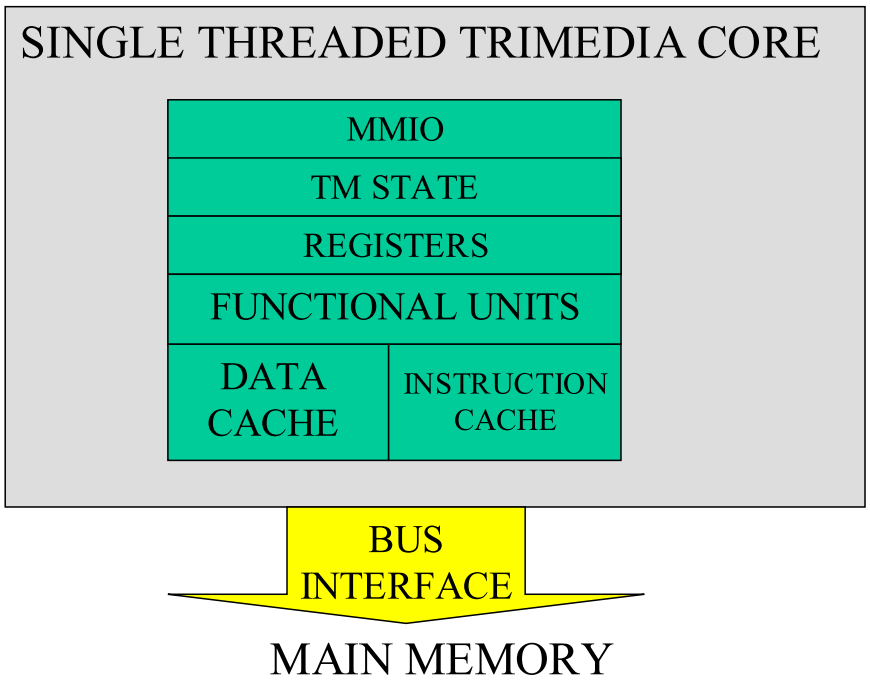

Figure 2.5: The architecture of the original TriMedia

the request and switch context, we set the pending buffer with the correct set and tag of data which needs to be fetched. However, we must not forget the copy back buffer. If the line which is going to be replaced is dirty, we set the copy back buffer as well. Note that the copy back buffer is set using the set and tag of the victim line and not of the line which must be fetched. At this point the hardware thread can make a request for the line and a thread switch will happen immediately after that. When we get the signal from the bus interface that the cache line is ready, we do not have to return to check if the data is in the cache, since the line cannot be overwritten in the meantime. This is because the line is marked invalid from the begin of the request. The other threads will not take that line as a victim line. This implementation however, gives a maximum amount of threads to be used. If the associativity is eight, then no more lines than eight can be fetched in a single set. If we want to bypass this constraint, then we have to increase the associativity or let the threads check for the data again, just as after a pending buffer hit. The possibility of deadlock then also occurs, if every time it appears that the data is not in the cache because another thread has taken that line as victim. The way to handle this problem is just make sure that if all lines in a set are invalid and used as victims to take the first line. To minimize the overhead of this all, we obey the restriction of the maximum of hardware threads. Note that this is a worst case scenario. In real life a lot of application will still run if we increase the number of hardware threads over the maximum.

If the hardware thread finds its requested line in the cache, but not in the Modified state, it needs to do a bus update. The modified state is necessary to ensure that the

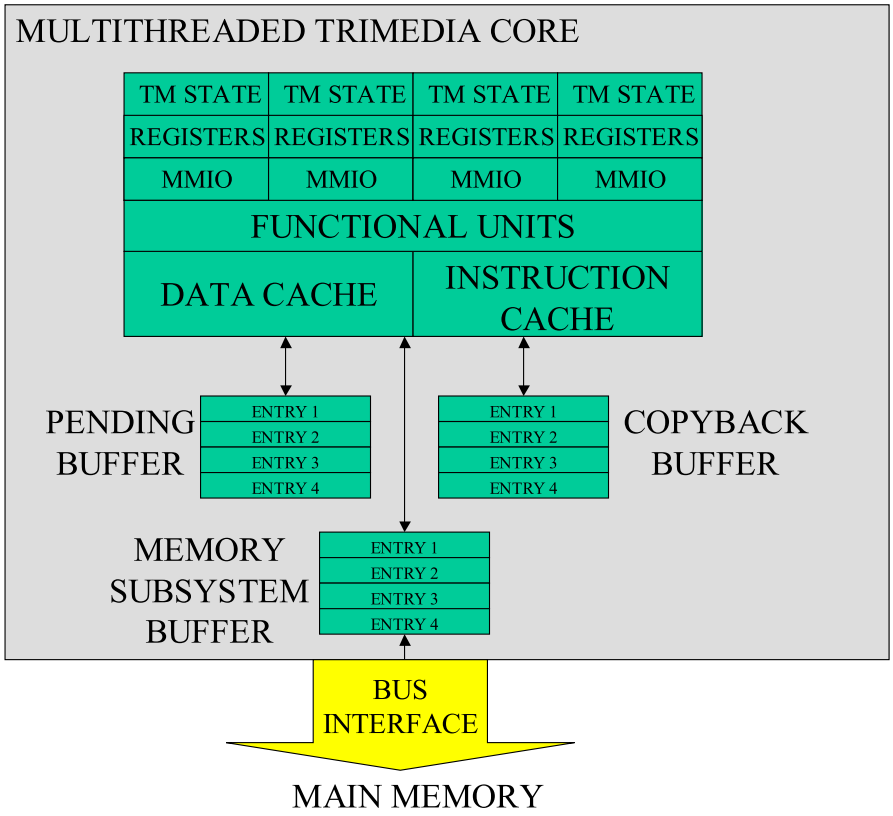

Figure 2.6: The architecture of the multithreaded TriMedia.

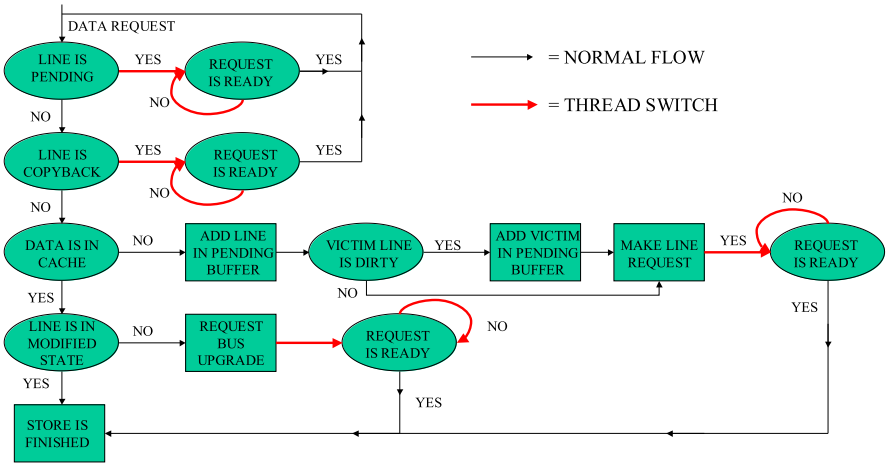

Figure 2.7: The flow chart for a store operation in the interleaved multithreaded TriMedia.

hardware thread has exclusive rights to write and change the values in that line. If it is not in this state, it meant that there is one or more copies of this line in other processors. The bus update makes sure that all other locations with the same line are invalidated.

When the bus update is completed the hardware thread has exclusive rights to modify the line. Thus the line is put in that state and modified accordingly. Just as with the request explained above, we do not need to check if the data is in the cache again.

Other fetches of data are the load instructions. These are different in flow as seen in the figure 2.8. This graph also is used for instruction fetch. The instruction fetch does not need another pending buffer. It can use the same buffer as the data cache since the addresses of data differ from the addresses of instructions. The instruction fetch does not need to check the copy back buffer though, since no copy back or write will ever be done on the instruction cache. But in real RTL the data caches and instruction caches can reside far apart. If they both need to access the same buffer, it will become a bottleneck. The buffer needs to be dual ported for simultaneous acess as well. The checks of the buffers and the cache can be done in parallel.

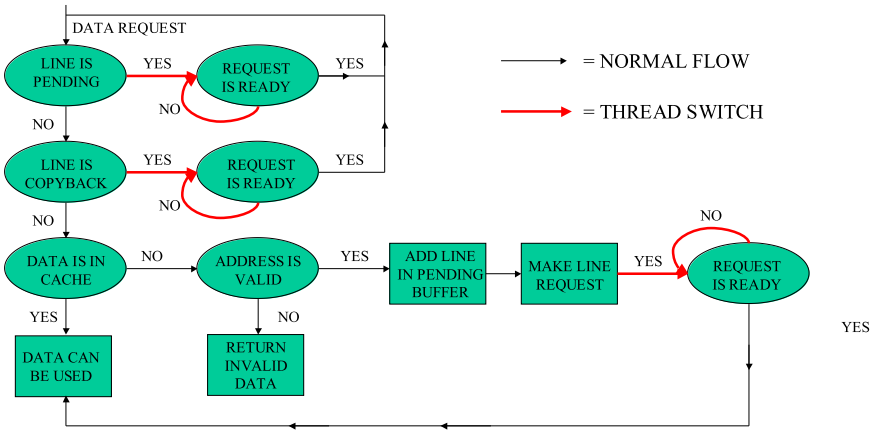

Figure 2.8: The flow chart for a load operation of the CPU in the interleaved Multithreaded TriMedia.

Both store and load operations start the same, with checking the buffers. The difference between the load and store is that the load does not need to write a value and therefore does not need the line to be in mesi-Modified state. If the line is in mesi-Invalid state, it does not count as being in the cache and a cache miss will occur. The store instructions need to have a valid address or a bus error will occur, which will cause the system to hang. The load instructions do not have this restrictions. If instructions are prefetched and executed, it may be that the branch to these instructions is never taken. Thus making the results invalid and never used. In an untaken branch it is quite possible that an address is calculated, which is not in any of the apertures. As long as the result is not used, this is no problem. The value in the register of a load to an invalid address is undefined. If the address of the load is valid then the pending buffer is set and the request is made. Just as a store instruction, the cache waits till the data has arrived.

2.7 Thread Switching Technique

The implementation of context switching in this multithreaded CPU makes use of the round-robin method. If there is a thread switch, the CPU uses the next number in

hardware threads. When the last hardware thread is reached, the next hardware thread will be the first one. This implementation reduces the complexity of architecture and the overhead for thread switching. Do not confuse this with the First In First Out (FIFO) method of the bus interface to handle multiple requests using the mss buffer. Another implementation would be using priorities. Every hardware thread is scheduled with the highest priority of its containing task first. But this approach adds in complexity and overhead cost. The calculation of the priorities must be done correctly, otherwise an important task will never be handled or an unimportant one is always at the head of the queue.

The moment when a thread switch must occur is important for performance issues. The setup of this research was to tolerate long memory latency operations using thread switches. If there is an instruction or data cache miss, there will be a stall time for the CPU. At this moment we should go to another context. The sooner the TriMedia knows that a miss occurred, the better. Because we do not have to throw away the entries in the pipeline stages of instructions that already started to execute. We need to use every cycle of stall time as efficient as possible. If the implementation of the thread switching technique is too slow, it can ruin the performance. The stall time must be significantly larger than the context switch time to increase the performance efficiently. In the worst case scenario, the stall time would be smaller than the switch time and thus deteriorating the performance. Since every hardware thread has an equal importance, they must all be handled evenly. With this in mind we chose to implement a thread switch when there is a first level cache miss. The CPU knows much earlier that there is a level one cache miss than a level 2 cache miss and, as such, reduces the pipeline flush costs. If then a second level cache miss occurs, we already are working with another hardware thread. The request has already been made for the data, be it in the second level cache or not. The advantage of this is quick knowledge when there should be a thread switch. However, there is a condition to make this implementation work efficiently. The cycles it takes for a thread switch to finish, must be equal or less than the time it takes to fetch data from the second level cache. If there are relativly many second level cache misses after a first level cache miss, this can be relaxed a bit. There has been a lot of research to implement a fast thread switching technique. For example [6] proposes a technique that is used in a Named State Processor (NSP). This NSP can switch in 1 cycle, with the help of a context cache. This is extremely fast and such a implementation justifies switching on a first level cache miss if we use that technique so we can switch in 1 cycle.

2.8 Quantum Time Expiration

Unfortunately, thread switching moments do not stop at a long memory latency operation. It can happen that for any reason what so ever, the hardware thread goes into a continuous loop. The reasons is the ll and sc loop which is explained in the next section. Software threads may be executed in parallel, but they may communicate with each other or exchange data among themselves. One hardware thread might go into a continuous loop, while waiting on some sort of feedback from another hardware thread. This will never happen, because that hardware thread is stalled or it is not his time to work. The second hardware thread must be given the opportunity to do his work.

If a software thread has relatively few cache misses, then this thread will be active most of the time. Since we handle every software thread with the same priority, we should give all the hardware threads just about the same amount of time to be active. If we do not solve this difficulty, we might end up with a low priority software thread being active all the time, while a real time application cannot do its job.

To avoid these problems of continuous loops and priorities, we either identify all of them, or give a maximum time that the hardware thread may work. The first option is hard to do. We do not know the priorities of the software threads and we do not know if there is an endless loop. The same instruction is executed, over and over again. But it still remains uncertain if that was not the intention of the software. Even detecting a continuous loop is difficult. If we check, for instance, 100 instructions. If the loop is shorter then these 100 cycles, it is detected. But the loop might be much bigger than that. There is no way to detect that. Thus for all the reasons listed above, we need a maximum amount of cycles a hardware thread may be active. We call this maximum amount the quantum time expiration (QTE). After this amount of cycles the context must be switched. The question now arises, how big must that maximum cycle count be. If we take a too big a value, the performance of the entire system drops greatly. If we examine a hypothetical situation where a loop only takes 5 cycles. The purpose of the loop is to wait for a response of another processor in a multiprocessor environment or of another hardware thread in a multithreaded environment. This means that in a multiprocessor system it takes these 5 cycles, plus the time it takes to transfer data to and from the processors. When in a multithreaded system this loop occurs, the other hardware thread cannot answer because it is not his time to be active. If that hardware thread finally becomes the current thread after the maximum cycle count, it can give the response. It then again takes some time for the first hardware thread to notice that the response is given, because it is now not active itself. It takes some research to figure out the best quantum time expiration. The experiments to acquire some results in this field is given in the next chapter of design space exploration. In figure 2.9 the flowchart for thread switching is shown. First we check if we have reached the quantum time expiration. If so, we do a thread switch and start to count the cycles the hardware thread is active again from another hardware thread. If the instruction is not in the cache, this must be fetched. Thus we do a thread switch to compensate this long memory latency operation. This thread switch makes sure that the counting of the cycles start from zero again. The same holds if we lack all data in the cache to complete all the calculations of that operation. If there is no cache miss, the operation is completed and the flowchart is repeated for the next instruction but the count of the duration of the active hardware thread is raised one cycle to compare with the time quantum expiration.

2.9 Switching During Synchronization Between Hardware Threads

In a multi processor environment the need arises to adjust a memory location without the risk that another CPU changes that location between the read and write of the first processor. One way of implementing this is using two special operations: load link (ll)

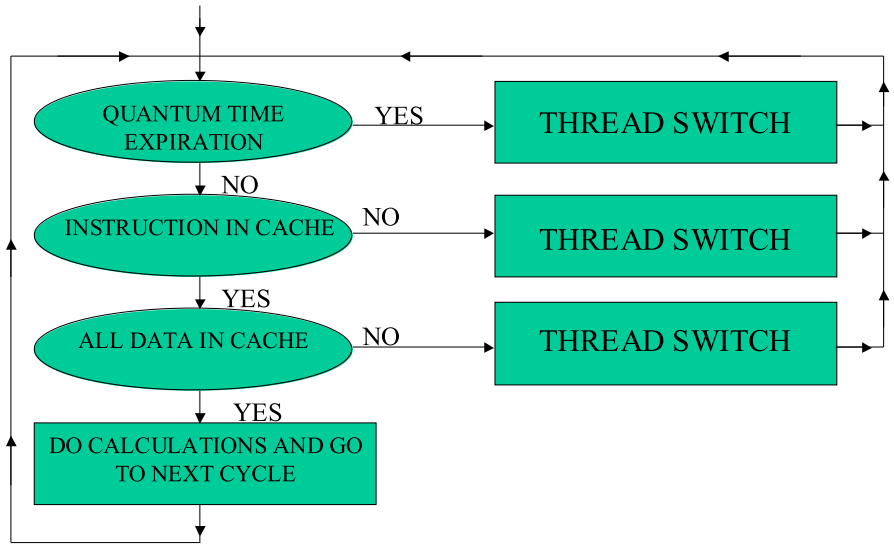

Figure 2.9: The algorithm of the thread switching technique of the multithreaded TriMedia.

and store conditional (sc). The load link and store conditional operations are used to simulate an atomic operation. The load link operation fetches the memory location and stores it in the register. The store conditional stores the contents of the address (which might be changed) back to the main memory. But the sc operation will fail if some other processor touches that between the load linked and the store conditional and the memory location is not overwritten. The processor then tries to do the ll and sc loop again until it succeeds. Possibility for an ll and sc loop is to acquire a semaphore or to use an atomic add. In an atomic add the instructions between the load link and store conditional are used to increment a value which is loaded by the ll instruction. For example to give every CPU their processor number. In the boot code of the processor it is specified that a memory location may contain their processor number. Suppose the memory location is stored in register 64 (r64). Since every processor wants to access that memory location at around the same time, an ll and sc loop, which is described below for the TriMedia instruction set, must be used.

```
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
```

First the instruction address is stored in r68. The next instruction does a load link on the memory location specified in r64 and stores the contents of that location in r65. Now r65 contains a potential processor number. The value in register 65 is then added with one and stored in r66. The fourth instruction is the store conditional that checks the location of r64, which was the location of the load linked value, with the original

value and stores the value of r66 on that location. This happens only if the original value in the memory equals the value of the load linked value in the register. If the store conditional succeeds then a logical TRUE is put into register 67. This means that a number has been brought into the registers and this number is added with one to give to the next processor. The two jump instruction handle the continuation of this all. If the store conditional fails, the jump to r68 is performed. The loop is performed once more until success. If the sc succeeds, the program counter jumps to the label NEXT (not visible in this part of the code).

In a normal ll and sc loop, the CPU waits until the result of the store conditional comes back, to know if the atomic operation has succeeded. However, within a multi-threaded environment, a correct answer of a succeeded store conditional, might become difficult when there is a context switch between the load link and the store conditional. The load linked value can be changed and back again. There is no way of telling to the hardware thread, which started the ll and sc loop, that there is a change in the memory. That hardware thread is in idle state, because of the context switch. For safety issues, every time there is a context switch between load link and store conditional, the sc instruction will always fail. Since the ll and sc loops are small, this has only a small deteriorating effect. The load link and store conditional are used to simulate an atomic operation. An atomic operation cannot be interfered by a context switch.

There is then the issue of a forced switch, because a hardware thread is active for too long. That is why we introduced the QTE in the previous section. The hardware thread should switch when the maximum cycle count, which a thread can be active, is reached. But if that switch moment falls in an ll and sc loop, it gets a bit complicated. The thread can continue a bit further, past the store conditional, and then either switch or wait till there is a long memory latency operation. This all seems quite efficient, but this is not the way. There will always be the possibility of the endless loop. If this loop exist between the ll and sc instructions, which is quite common, the system hangs. Therefore the forced switch is independent of the execution of ll and sc. The ll and sc loops are not that big and the data is in the cache. The damage is minimal.

As soon as a load link operation is started, it sets a bit called "store conditional fail", to FALSE. This bit is part of the hardware thread and is duplicated for every hardware thread. When there is no switch of context before the store conditional happens, the store conditional finishes as in a normal implementation. When there is thread switch between the ll and sc instructions, the "store conditional fail" bit goes to TRUE and it will always fail. Since every start of a load link sets the bit to FALSE to begin with, we do not have to check if we already are in a ll and sc loop if there is a context switch. This reduces the overhead. We take the assumption that there is no ll and sc loop within an ll and sc loop. This is a valid assumption. It would be too complex to implement for an atomic operation to contain another atomic operation.

2.10 Software Enhancements

At this point we only looked at the hardware implementation of thread switching. The most efficient way in increasing performance is using a combination between software and hardware, but we do not know for sure after testing; there might be unnecessary many

switches. In this research the emphasis is put on hardware changes. The significant importance of software changes make it necessary to comment on this subject. The multithreading method here is interleaved. This means that the hardware thread do not operate simultaneously. This has consequences when hardware threads or CPUs are trying to communicate. This happens when all of them are trying to get an independent CPU number to identify themselves. In the bootcode it is thus optimal to have an intentional cache miss after acquiring the CPU number. This gives control to another hardware thread who can acquire his specific CPU number.

In the program code other necessary thread switches may be recognized during compile time. A software addition can remove, in best case scenario, the quantum time expiration completely. For example a software addition is then a specific instruction that forces a thread switch. Since it is dangerous to remove the QTE completely, because of unexpected circumstances, the most efficient way is too increase this quantum time expiration dramatically. A quantum time expiration thread switch is not what we want, because it is not a way to fill in a long memory latency operation, because there is no long memory latency operation that causes the thread switch. In this hardware implementation it is just a safety net for endless loops. The damage is reduced if the cycle costs of the switch are reduced. The damage is removed if there is no forced context switch at all. This can be achieved by software enhancements. An additional operation is added to the instruction set. This instruction enforces a context switch. The operation can be used if the program needs to communicate with other hardware threads in any way. In this case we need to know that the program will run on a multithreaded environment in stead of a multiprocessor environment. This will also solve the problem of endless ll and sc loops.

Besides creating a new addition to the instruction set, the instruction can be implemented by two existing operations. The first operation invalidates a line and the next operation asks for the same line. If this happens in a multiprocessor or multithreaded environment the cache miss is completely useless but it will generate a long latency memory operation and thus a thread switch. The line must however be clean or else it will create a copy back situation which is very costly in cycle time.

Adding an instruction to the known set implies more hardware. It also destroys compatibility. Every program should be first recompiled for other non-multithreaded CPUs. Older programs can run on the new multithreaded environment though, but not the other way around. But this is not recommended, since we assumed programs with instructions who force context switches. The quantum time expiration is set high, because of this. The quantum time expiration here is only used for unexpected events and loops. Thus the endless loops which are more numerous in the old code will run for a long time before the switch is forced. This is a research of hardware, with the idea of no changes in the software and compiler. Therefore the quantum time expiration is set low. This will decrease performance, but does not have the problems explained above.

Another software enhancement is to make the QTE software controllable. It might be difficult to predict the QTE, but when we do know a optimal value, we can set it for that application. This way we can make sure that the value of QTE can be versatile and thus can change if it needs to be.

3 Design Space Exploration

In this chapter the results of the experiments are discussed. First we will set out the simulator and the used programs. Then I will present the design space exploration. Through the benchmark programs the results are shown to provide with information about the performance. We set out the different consequences when changing different parameters. After the three benchmark programs we will provide the chip size increase because of multithreading. In the final section all the results will be discussed.

3.1 Simulated SoC Environment

CAKE is the abbreviation of Computer Architecture for a Killer Experience. CAKE is a multiprocessor architecture[10]. It addresses the need scalable architectures. This means that an interconnection network connects multiple processors and peripherals and other tiles with the same configuration. This is all shown in figure 3.1. We used a CAKE

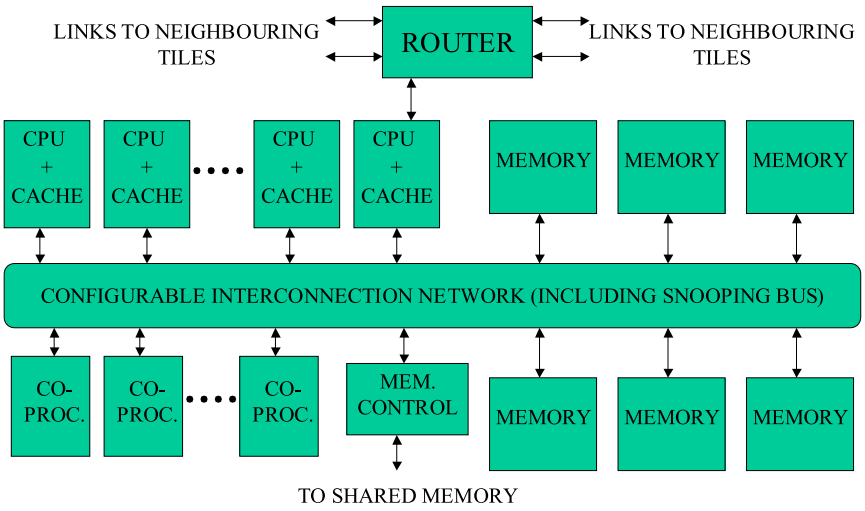

Figure 3.1: Typical architecture of a CAKE tile

simulator to simulate the system on chip (SoC). It had a adjustable number of MIPS and coprocessors. You could simulate multiple tiles. But the important aspect of this simulator for this research was a cycle accurate TriMedia.

Figure 3.1 gives only a broad idea how a CAKE instance can look like. Cakesim is a simulator of CAKE, in which you can set all kind of parameters. Among them are an adjustable number of MIPS, number of TriMedia's and number and type of coprocessors. The amount of tiles you use. The most important thing for this research is the adjustable

amount of TriMedia's. For the purpose of this research we only look at one processor.

The TriMedia asks for data through a bus interface. The bus interface transfers its request to the interconnection network. It is possible that the interconnection network is connected to other locations such as level 2 caches, as depicted in figure 3.1. For the purpose of this research the interconnection network is only connected to the main memory.

The TriMedia in the simulator was changed to be a multithreaded version. All the changes which were discussed were put in the architecture. All the other possibilities of the CAKE simulator were not used. Only one single TriMedia with only one main memory. We made sure that the data cache and instruction caches were shared between hardware threads. All the separate states of the hardware threads reside in the TriMedia block. This way we made sure that from an architectural point of view, there was only one TriMedia. But from the software point of view, there were more. All connections must be implemented correctly to make sure that the hardware threads can communicate with each other through all the buffers. And the connection to the bus interface was made.

Only one bus interface connection was created. The results for more bus interfaces were calculated and not simulated, when running the program with multiple hardware threads and only multiple bus interface. The reason that this was calculated is because of the constraints of the current simulator. With this calculation, which is explained in the next section, we do not have to fix the simulator. Fixing the simulator to be completely multithreaded is a gruesome task, since the simulator is complex and built for the sole purpose of handling a single threaded TriMedia. The disadvantage is that we ignore possible overhead costs because of overhead when having more than one bus interface.

3.2 Applications

The applications that are used on the simulator are:

- pingy
- opt-mpeg2dec
- normal_mpeg2dec

These programs all have different properties. Their characteristics and usefulness they have in testing the performance of the proposed interleaved multithreaded architecture of a VLIW TriMedia multimedia processor are described below. All program were compiled using a commercial compiler for the TriMedia instruction set, which made the performances of the code better than using a open source compiler.

Pingy is a program which was originally created to test cache consistency end coherency for multiprocessor environments. There is one cache line that will be changed on every byte. If there is only one processor, that processor must edit all the values of the cache line. If there are more processors then the workload will be divided evenly among them. In a multiprocessor environment this means that the workload is divided

among all processors which work in parallel. Thus the amount of cycles it takes to complete the program should go down, when the processor count increases. But since it concerns the same cache line, the interprocessor communication kills the benefit of data parallelization. In effect all the processors will do their work sequentially. Only one processor can make their changes. This means that there will be no decrease in cycle count, when using more parallel processors. Any change in cycle count will be caused by the fact that every processor changes the same cache line. Only one processor has exclusive rights, thus the line travels from one cache to another. This takes time. It depends on the implementation of the internal network, how much time that really is. In a multithreaded environment the scenario changes. The program is treated the same way, but the effect is different. The software sees every hardware thread as a separate processor. The workload will thus be divided evenly among them. The difference is that these hardware threads do not work in parallel, but sequentially. There is no decrease in cycle count when the number of hardware threads increases, since they do not operate in parallel. With this program we can illustrate the benefits of the fact that the hardware threads share the same data cache. The cache line does not have to travel from one cache to another. This will give no rise in the cycle count. In an ideal situation this will give a constant amount of cycles it takes for the program to run. Any increase will be the consequence of thread switching overhead.

Opt-mpeg2dec and norm-mpeg2dec are both mpeg2 decoders. MPEG2 is a standard for decoding and encoding moving images, such as in films and animations. Though the standard is the same with both applications, they are implemented differently and use different input streams. These are real applications to research the performance in normal applications. Since the TriMedia is a multimedia processor, we chose for these multimedia application, since these are the most likely applications to be used with this processor type. Opt-mpeg2dec is a program specially optimized to work well on a TriMedia.

Norm-mpeg2dec is as said a MPEG2 decoder. It is not optimized as the opt-mpeg2dec, it is an off-the-shelf version of an mpeg2 decoder.

3.3 Simulation Results

This section describes the results which were made using the simulator of the interleaved multithreaded TriMedia when running the different programs with different parameters. The parameters and their range in which they are changed are shown in table 3.1

The applications are listed in the previous section. By changing the application we can check whether the optimal choice of parameters is dependent of the chosen software program. The design space exploration limits itself to three programs. But only the 2 MPEG2 decoders can be used as real benchmarks for concluding the dependencies of parameters and applications. It is also these two last programs where we changed all the listed parameters. It is unnecessary to do that with the program pingy, because it does not represent a real application.

The three sizes of the data cache that are used are 16 KB, 8 KB and 32 KB. The instruction caches are always twice the size of the data cache, thus they are 32 KB, 16 KB and 64 KB respectively. When the cache size decreases on a normal single threaded CPU,

Parameter	*Range*
Application	Pingy, Opt-mpeg2dec, Norm-mpeg2dec
Data Cache - Instruction Cache Size	16KB-32KB, 8KB-16KB, 32KB-64KB
Quantum Time Expiration	10...500 cycles
Number of Threads	2 and more
Memory Speed	Types 1, 2 and 3

Table 3.1: The parameters of the simulator and the range in which they vary.

the effect will be an increase in cycle time caused by more misses in the cache. Simply because all the data that used to fit in the caches, cannot be stored there anymore. When a line arrives, which cannot fit in the cache anymore, there will be room created for that line in the cache. This means that one line must be copied back to main memory. We call this line the victim line. If that line will be used again when it is copied back, a cache miss will occur because it has been replaced by another line. When the cache was bigger, the line did not need to be copied back, thus saving precious cycle time. In a multithreading environment it is interesting to see whether more cache misses give more opportunity for thread switches to fill up the memory gap. We then have a smaller chip, which might be used in other areas where they need smaller chips, like mobile phones. We might end up with a smaller chip but with a better performance, which will make more advanced real time application possible in that area.

The quantum time expiration is the maximum amount of cycles one thread may be active. The QTE ranges from 10 to approximately 200 cycles with the first two programs. The last program norm-mpeg2dec uses a wider range of QTE to illustrate more clearly what trend the numbers follow. With the help of the experiments the range of the QTE is determined. Every time the QTE is reached, a thread switch is forced. This action has a cost of one cycle. When a normal cache miss occurs, the cost of a thread switch is masked by the time it takes for the previous thread to fetch its necessary data. If we put the value of QTE too small, the unnecessary switches will increase too much. In effect adding to much thread switch overhead costs. On the other hand, a QTE, which is too large, will cause infinitive loops, who wait for response of other CPU's, to be executed too long. Useless code will be executed, when another thread has important work to do. Especially when that thread has a job to get the other processor out of the infinitive loop.

The number of threads is one of the parameters, you tend to think to be the most important in changing. The emphasis is put on two threads in one processor, because of limitations in the current simulator. Two threads may give insight in how adding one extra thread will speedup the program execution. When the number of threads increases, the cycle count should go down in the benchmark programs, not in pingy as we will see later when we discuss the results of this program. The main research idea is how much this amount is and if this decrease in cycles makes it worthwhile to implement multithreading in the TriMedia. When we consider two threads the chip size increase will

be the smallest and the application should have at least two threads. If the application is not multithreaded or has fewer software threads then hardware threads the performance will go down because of the overhead of switching. If there are more software threads then hardware threads, the software threads will wait until one hardware thread is finished with its task.

The memory speed is changed into three types. Detailed information about them is given in table 3.2. Slower memory will give more stall cycles for every request to the main memory. Therefor the multithreaded architecture has more opportunity to be effective in hiding the stall time of the multithreaded processor.

Parameter Description	*Memory Type 1*	*Memory Type 2*	*Memory Type 3*
CAS latency	3	6	12
Row to col delay	3	6	12
Row access time	7	14	28
Row to row delay	10	20	40
Write to precharge delay	2	4	8
Delay btw rw in diff banks	1	2	4
Precharge to activate delay	3	6	12
Any row to any row delay	2	4	8
Memory to CPU delay (mem cycles)	2	4	8

Table 3.2: The different types of memory used with their parameters in cycles

In the next sections the data will be given regarding the applications. In these tests the parameters which were listed above were changed. In the results we could derive the numbers and the relationship of the following items.

- Number of Hardware Threads
- Number of Bus Interfaces
- Primary Cache Sizes
- Efficiency of Multithreading
- Performance

Number of Hardware Threads is the amount of hardware threads that are in the processor. We did the same tests on a single threaded TriMedia, to compare these results with the multithreaded versions.

Number of Bus Interfaces depicts the amount of parallel requests the bus interface from TriMedia to main memory can handle. When there is a cache miss from one thread, this thread is stalled. The request is brought to the bus interface and the data is fetched. However, this takes time. When the cache miss occurs and the request is made to the

bus interface, another thread becomes active. This thread execute as normal, so it can generate a cache miss as well. Now the problem arises, when the previous request of the previous thread is not yet finished. The bus interface is busy and therefor cannot handle this request. The request must be put in a buffer or the bus interface must be expanded to handle this second request. The second solution may be faster, but it creates a bigger area for the chip. The maximum of parallel request is the same as the number of threads. When a request is made, the thread becomes idle until the request is handled. Thus a thread can make only one request at a time. We can also choose to increase the bus interface, but not to the maximum of possible requests. It depends on what the benefits of cycle gain will be, to decide what is the best solution.

Primary Cache Sizes have a key role in the amount of cache misses. A bigger cache means fewer cache misses. The multithreaded architecture has then less opportunity to efficiently use stall time. We need to investigate what the effect of the cycle count is when we increase or decrease the primary cache size. When we decrease the cache size the cycle count will go up, but the amount of additional cycles that are needed to finish the program will tell how efficient multithreading is with more cache misses. On the other side if we increase the cache we get fewer misses and multithreading may become even worse than the original TriMedia, because of thread switching overhead.

Efficiency of Multithreading is already mentioned as the effectiveness to handle stall time of more and less cache misses, but the main idea is how big the cycle difference is with respect to the single threaded TriMedia with the same configuration. It is not necessarily that the multithreaded architecture is better than the single threaded architecture. If the program has a good performance to begin with, the thread switching overhead might even deteriorate overall speedup. We need to investigate what the relation ship is between the performance of the applications and the QTE. On top of that there is the extra size increase that the extra architecture needs. If the size increase for the chip is too high, then it might be more interesting to go for a multiprocessor environment in stead of a multithreaded one.

Performance is the collection and connection of all the items listed above. Performance is not simply the amount of cycles it takes the program to run on the new architecture. There are also the connection between the different items. We need to check if there are correlations between all the items and if these relationships make it harder or easier to implement multithreaded architecture in the TriMedia

If we increase the number of threads, there is the choice to either increase the number of bus interfaces or not. If we increase the number of bus interfaces we get a better performance, but we increase the chip size a bit and we increase the overhead. Furthermore there are more additional problems when we add more bus interfaces. For instance the multithreaded TriMedia may have 5 bus interfaces, but the main memory has only enough bandwidth to handle two parallel requests. In effect, three bus interfaces are completely useless and a waste of overhead. We suggested the use of a mss buffer, which makes it possible to serialize requests to the bus interface. This makes it possible to vary the amount of bus interfaces from one to the number of hardware threads. There is no need to make more than the number of hardware threads bus interfaces, because there are at most that many parallel requests as the number of hardware threads. If we take the hypothetical situation of a multithreaded processor with 3 hardware threads,

it might be that there are never three requests in parallel, because the first one is then already handled by the bus interface. If a bus interface is free, the mss buffer leads the request to that bus interface and back again. If there are the same bus interfaces as hardware threads, the mss buffer becomes obsolete, which reduces overhead and design complexity.

The caches in the TriMedia architecture is quite extensive. It takes up a big part of the chip. If we decrease the size of them, we decrease the size of the chip. The multithreaded architecture may be able to keep still a good performance and then we have a smaller chip with a good performance. When we decrease the cache we get more cache misses, but if we use that stall time because of cache misses efficiently, the chip might have a comparably performance as the big single threaded chip. This might be interesting to develop, because smaller chips are cheaper.

Chip size and complexity are the major costs to prevent the multithreaded architecture to be implemented in the TriMedia processor. If the chip size is too big, it might be convenient and cheaper to create a multiprocessor environment with off-the-shelf TriMedia's. If the complexity of the design is too high it takes too much effort for the designers to implement it functional correct. This too much effort can cost too much time and too much money.

3.3.1 Benefits of Shared Caches

In this section we will present the result, when running the program pingy. First the results are compared when changing the quantum time expiration. Then the results are shown when running this program in a multiprocessor environment. It is not the purpose of this research to compete with more parallel processors, but with only one. But the characteristics of this program make it worthwhile to do this. Thus showing the benefits of a shared cache.

With the the results of the experimental setup we can derive figure 3.2. In this figure the amount of bus interfaces gave only a minimal change in cycle count. To help the visability of the picture we chose to show all configurations with 2 hardware threads in one color, and the configuration with three hardware threads in another color, regardless of the amount of bus interfaces. In figure 3.2 we see that there are two different optimums for the quantum time expiration. If there are 2 threads, the best choice would be a QTE around 80 cycles. But when we increase the number of threads to three, the best way would be to choose a QTE of 60. This is an interesting fact. We also notice that a thread increase will cause a increasing amount of cycle time to complete the program. This is as expected since the program will have to suffer more from thread switching overhead costs. In an ideal world the cost for thread switching would be next to nothing, causing a constant cycle count. This amount would not change, when we change the number of hardware threads.

The interesting part is when we compare this data with the cycle count of a multiprocessor system. This gives us table 3.3. In this table we see all the result when using a normal TriMedia as the cores for a multiprocessor environment. These values can be compared to the multithreaded TriMedia. For the best case scenario we will take the results of a QTE of 70. This value of QTE is based on the fact that for 2 threads the

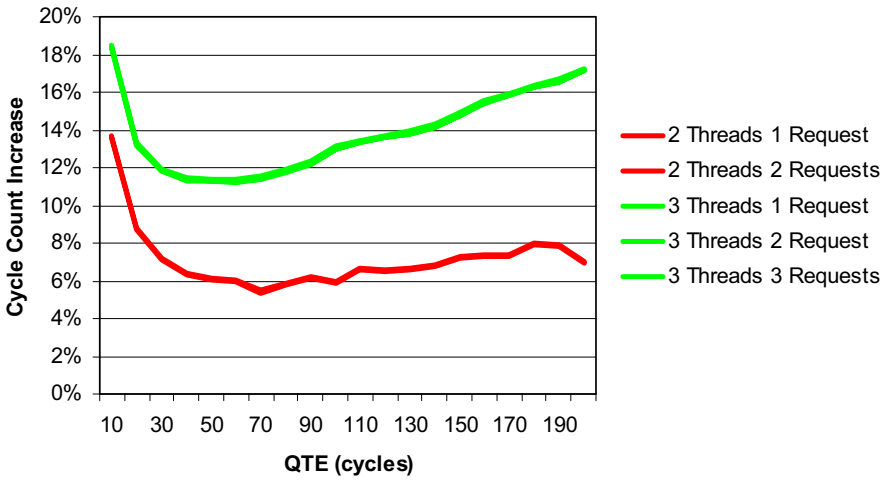

Figure 3.2: The relative increase in cycle count, when using different implementations of multithreading, for the program pingy as a function of the QTE.

optimal value lies around 80 and for three threads it is around 60 cycles. A reasonable conclusion would be that all optimal QTE lie around these values. Table 3.4 gives us an

Number of Trimedia's	*Resulting Cycles*
1	3.093.728
2	13.909.620
3	10.445.603
4	9.171.320
5	7.705.969
6	7.674.526
7	9.302.643
8	8.129.830

Table 3.3: Results of pingy when using a single threaded Trimedia in a multiprocessor environment

idea when the same program is run within a multithreaded Trimedia, without a multiprocessor environment. In this table all possible implementation options, concerning the number of bus interfaces, are taken into account.

The tables 3.3 and 3.4 tell us that the multiprocessor environment stays way behind the multithreaded environment. Even with 8 processors working in parallel, it still cannot

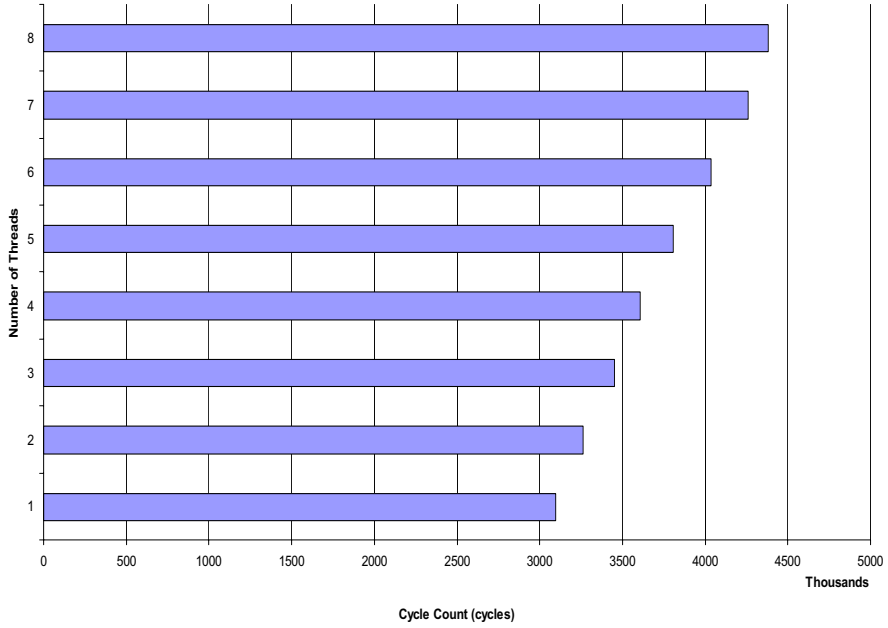

Figure 3.3: Thread switching cycle cost shown with the program pingy for multiple hardware threads

compete with a multithreaded version. This is because there is one cache line that goes from one processor to the next and back again in the multiprocessor environment. The line keeps moving from one data cache to another. But this is only what happens in a multiprocessor environment. If we take a closer look when hardware threads are involved, the story slightly changes. Indeed the cache line goes from one hardware thread to another, but it does not need to change in the cache location. This is the advantage of having a shared data cache. All hardware threads on this cache have equal rights. If one hardware thread has exclusive write rights on a cache line, all the hardware threads on this multithreaded processor have the right to modify this line. This gives a view of the benefits of having shared caches among the hardware threads.

When we look at the amount of cycles it takes for the multiprocessor environment we see that in overall the cycle count is more then the multithreaded ones, but no real trend can be found. This is explained as followed. The line will be changed in all places, but these changes that must be made are distributed evenly among the processors. This means that the total workload remains the same. But because only one CPU can modify the line at a time, in effect the parallel processors work sequentially, as do the processors in a multithreaded environment. But it is the migration that causes the problems when dealing with multiple caches. Every CPU is in a fight with the rest of them for control of the line. How this fight is resolved, depends on the architecture. Can a processor, who can modify the line, change more than one value, or is the line requested by another core

almost immediately after the fetch? When 2 CPUs are involved we see that there is a lot of migration and invalidating. This is handled by the architecture of the simulator. We can see from the result that the behavior of the system will give different result for a different amount of CPU's. The amount of invalidations that are necessary and the amount of cache traffic varies quite a lot. We notice that the amount of traffic does not grow with the amount of CPU's. This phenomena is architecture specific. But we can conclude that there is traffic because of separate caches, and thus the multithreaded TriMedia wins precious cycle time in that aspect.

In fact we must check one multithreaded processor against the single threaded normal processor. In this way we can decide if we get a speedup enough to validate implementing multithreading. The aim was to let the software see every hardware threads as normal processor. Thus the software sees the multithreaded environment as a multiprocessor environment. Thus if the software sees more processors the program code knows this. We can see this by looking at the amount of cycles it takes for a multithreaded environment in 3.3. It increases because of the amount of thread switching that is needed. The workload is divided among all hardware threads. there is no migration of the cache line as within a multiprocessor environment. If there is no migration, then there will be no increase in cycle counts. Furthermore, we stated that the total workload does not change. With this information we can conclude that, in an ideal situation, the cycle count does not change. However we do see an increase in cycles. This is because of the overhead when thread switching is involved. We see that for this program there is almost a linear increase in thread switching cost, when we increase the number of hardware threads.

This experiment shows the benefits of a shared data cache. This means that within the program pingy, in a multiprocessor environment, every cache miss occurs because of the existence of the same line, in modified state, in a different cache. This existence makes th same cache line in all the other caches invalid. All of these misses are reduced and removed by the shared data cache in a multithreaded environment. We can see how big the effect is with this 100% reduction of misses when they occur because of data cache sharing between processors. It would be interesting to know what the benefit or speedup would be when this percentage drops. When not all misses are because of data sharing. In real application data is not shared that much. Thus the speedup of a real application will be less then this application, since a lot of cache misses occur because of the fact that the data is still in main memory. Then also speedup will be gained from the fact that the time when the data is fetched, will be used by another thread to do his work. The question remains what the speedup is when a the percentage of misses because of data sharing decreases.

Number of Threads	*Forced Switch*	*Number of Requests*	*Number of Waits*	*Original Cycles*	*Cycle Reduction*	*Resulting Cycles*
1	43.405	1	0	3.137.880	0	3.137.880
2	45.414	1	96	3.263.665	0	3.263.665
		2	0	3.263.665	2.816	3.260.849
3	48.126	1	109	3.451.197	0	3.451.197
		2	9	3.451.197	3.671	3.447.526
		3	0	3.451.197	4.124	3.447.073
4	50.299	1	140	3.609.708	0	3.609.708
		2	28	3.609.708	5.284	3.604.424
		3	8	3.609.708	6.024	3.603.684
		4	0	3.609.708	6.467	3.603.241
5	53.027	1	155	3.805.864	0	3.805.864
		2	50	3.805.864	5.889	3.799.975
		3	17	3.805.864	6.692	3.799.172
		4	6	3.805.864	6.741	3.799.123
		5	0	3.805.864	7.160	3.798.704
6	56.185	1	179	4.035.236	0	4.035.236
		2	57	4.035.236	6.431	4.028.805
		3	22	4.035.236	7.493	4.027.743
		4	11	4.035.236	7.750	4.027.486
		5	6	4.035.236	7.768	4.027.468
		6	0	4.035.236	8.171	4.027.065
7	59.296	1	204	4.259.921	0	4.259.921
		2	66	4.259.921	7.382	4.252.539
		3	29	4.259.921	8.562	4.251.359
		4	15	4.259.921	8.852	4.251.069
		5	10	4.259.921	9.198	4.250.723
		6	4	4.259.921	9.229	4.250.692
		7	0	4.259.921	9.620	4.250.301
8	60.953	1	215	4.382.033	0	4.382.033
		2	91	4.382.033	7.892	4.374.141
		3	46	4.382.033	9.396	4.372.637
		4	22	4.382.033	9.697	4.372.336
		5	15	4.382.033	9.913	4.372.120
		6	7	4.382.033	9.920	4.372.113
		7	3	4.382.033	9.929	4.372.104
		8	0	4.382.033	10.300	4.371.733

Table 3.4: Results of pingy with a time quantum expiration of 70 cycles, a data cache size of 16Kb, instruction cache of 32Kb, both with an associativity of 8

3.3.2 The Optimized Mpeg2 Decoder

This section deals with the program opt-mpeg2dec. This is an MPEG2 decoder, adjusted to be used with the TriMedia processor in the most optimal way. Since this is a real application which can be used as a benchmark, the results are more important then the previous program pingy. Therefor there will be a lot more tests done with this and the next program. First the results are shown when the simulator is set to different quantum time expirations. Then the effects of a slower main memory and a changes in cache size are researched.

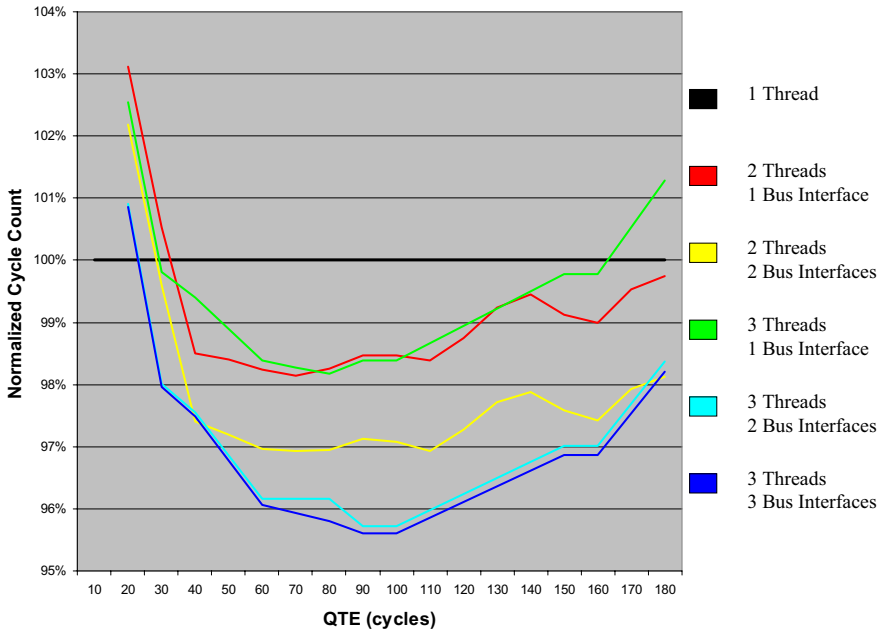

Figure 3.4: The relative change in cycle count, when using different implementations of multithreading, for the program opt-mpeg2dec as a function of the QTE.

In figure 3.4 the results are shown for the program opt-mpeg2dec. Here we see the relative amount of cycles that it takes for the simulator to complete the decoding process of the mpeg2 encoded pictures. Every line represents a different implementation of interleaved multithreading. The amount of hardware threads vary from one (singlethreaded) to three hardware threads. On top of that the amount of bus interfaces to make parallel request are taken into account as well.

The top straight line represent the amount of cycles it takes a normal single threaded processor. This is done for reference. This number is achieved when we set the QTE as such a high value that there will be no forced switch because of it. If we do not do that and we look for instance at the results when using a QTE of 10 cycles and a primary data cache of 16 KB with one hardware thread. The amount of cycles equals 472.763.711. Since the simulator is built to be a multithreaded environment, this amount is too much. The overhead for the thread switches must be deducted. Every thread switch will cost

1 cycle. There are 39.122.493 forced thread switches found. This means that every forced switch will cost one cycle, while it should take zero cycles. There is no need for a thread switch, thus in a single threaded CPU this will not happen. With this in mind we determine the total cost of the program opt-mpeg2dec to $472.763.711 - 39.122.493 = 433.641.218$ cycles. Table 3.5 gives the amount of cycles of a single threaded CPU for the different quantum time expirations and their amount of forces switches because of the QTE violation. In this table we see that the resulting cycles for a single threaded

QTE	*Original Cycles*	*Forced Switches*	*Resulting Cycles*
10	472763711	39122493	433641218
20	452912402	19275213	433637188
30	446271326	12639157	433632169
40	442993997	9358829	433635168
50	441040827	7408838	433631989

Table 3.5: Cycle costs of the program opt-mpeg2dec for a single threaded TriMedia processor, with a cache size of 16Kb and instruction cache of 32Kb, both with an associativity of 8

TriMedia stays about the same. The small differences can be ignored on such a big amount of cycles. In this table it is justified to take the constant value of around 433 million cycles for a normal CPU. If the QTE is taken extremely high, the amount of forced switches will go down to zero and a cycle amount around 433 million will be reached to complete the program. The exact count of this test, when there is no forced switch equals 433.629.256. This means that on a normal CPU it would take around 433 million cycles for opt-mpeg2dec to complete successfully. All the differences in this value and the values we get in table 3.5 is because of the subtle changes that may come when a forced switch is executed.

Getting back to figure 3.4 we see how much multithreading is effective. An important aspect is how much changing the QTE changes the performance. The lines in this figure are created by taking the values of the cycle counts, where possible, and taking the trendline of these points. The trendline is the moving average with a period of two. In general we see that there is an optimal quantum time expiration of around 90 cycles. both increasing as decreasing this number increases the amount of time it takes for the program to complete.In fact the QTE can be chosen so inefficient, that a single threaded CPU outclasses all multithreaded versions. This remark shows that indeed it is very important to accurately set the QTE. This also encourages the development of software additions to force a hardware context switch. The QTE is a major player in affecting the performance. The forced switches need to be removed without ending in an endless ll and sc loop. We expect that adding these changes will be a great benefit for performance, but it does need a new compiler for the programs. This will cost development time and thus money. It still remains to be seen if multithreading is valuable enough to invest in

this.

If we look at the maximum performance boost we see that at 90 cycles it takes a normal processor 433.629.256 cycles. This exact amount is taken from the test when putting the QTE to such a high number that there are no forced switches. If we compare this with the results of a 3 threaded CPU with a capability of handling 2 parallel requests to the bus interface, which takes 415.035.899 cycles. We get performance boost of $\frac{433.629.256-415.035.899}{433.629.256} \times 100 = 4.288\%$. This does not seem much, but we must take a look at the cycles per instructions. the original CPI can be calculated that it took 433.629.256 cycles and 398.854.744 instructions for a single threaded CPU. This gives a CPI of $\frac{433.629.256}{398.854.744} = 1.087$. The multithreaded version did the same program in 415.035.899 cycles. The program code will take about the same amount of instructions. This is not completely true because of the overhead, but if we take this assumption we get a reasonable accurate sense of the CPI. Thus the CPI will become $\frac{415.035.899}{398.854.744} = 1.041$. Here we see that the program speedup is low. But we must realize that this program is a mpegdecoder optimized for the TriMedia. It already has a real good CPI to begin with. This leaves little room for improvement. But improvement is there none the less.

If we look at the effects of the different types of implementation of multithreading we see that there are two lines who are quite similar; 2 hardware threads with 1 bus interface almost equals 3 hardware threads and 1 bus interface. This means that increasing the number of hardware threads is not effective anymore if you keep the limitation that only one request can be handled by the bus interface at the time. The lines that represent 3 hardware threads with 2 bus interfaces and the one with 3 hardware threads and 3 bus interfaces are comparable as well. We can deduct from this fact that adding additional hardware for a third bus interface is not efficient for the performance/chip size ratio. Now it will be worthwhile to investigate if adding more threads and/or bus interfaces is good for performance. If we look at the numbers that are generated, when the QTE equals 90 cycles in the results we see that increasing the number of hardware threads increases the number of forced switches. We already determined that this number should be zero in a ideal world. This number may increase, if the benefit of the additional hardware thread is worthwhile. In this case, if we go from one to two hardware threads with capability of 2 parallel requests (for maximum benefit), we get a difference of 433.629.256 - 421.904.537 = 11.724.719 cycles. If we go from 2 hardware threads to 3 hardware threads, the difference becomes 421.904.537 - 414.561.751 = 7.342.786 cycles. In percentages this means that we first get a speedup of $\frac{11.724.719}{433.629.256} \times 100\% = 2.704\%$ to go from one hardware thread to two, while when adding an additional third hardware thread to the two threaded implementation we get a speedup of $\frac{7.342.786}{433.629.256} \times 100\% = 1.693\%$ This is a huge drop in performance, which is low to begin with. Therefor, at least for this application, it is not worth adding more hardware threads. The additional hardware thread will not give enough benefit to accomodate for the increase in chipsize. Since it is not worth adding more hardware threads, we do not need to concerns ourselves whether the increase in hardware thread count should be accompanied with more bus interfaces. But at least there must be pointed out that we do not need to use the same number as the number of harwdare threads that are available. When we had three hardware threads we deduced that the third bus interface for three threads only gives a minimal extra speedup.

The question rises how much the effectiveness depends on the speed of the main memory. Of course when a cache miss happens, there will be more time for the CPU to wait for the data. To determine how much the benefits will increase, the memory will be made slower. There are 3 types of memories which are used in these tests. The first and fastest type is used in the experiments above. The second and third type vary in parameters which are shown in table 3.2.

If we take the results that were created with the memory type 2 of table 3.2 we get the figure 3.5. In this figure we see the same trend as in figure 3.5. We see that the

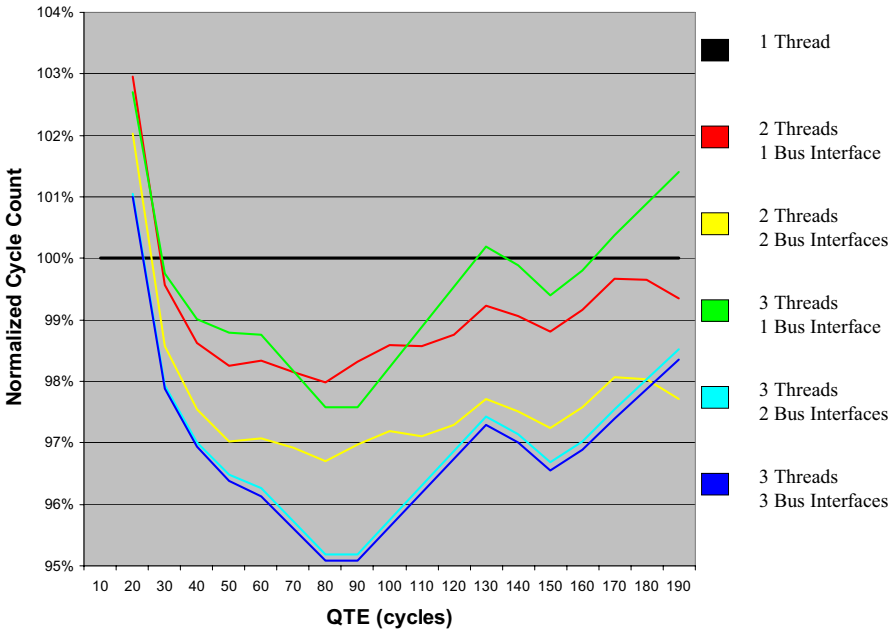

Figure 3.5: The relative change in cycle count, when using different implementations of multithreading, for the program opt-mpeg2dec as a function of the QTE for the slower memory type 2.

optimum lies this time with a quantum time expiration of around 80 and 90 cycles. This is the almost the same as when a faster type of memory was used. The QTE is important for the performance, but apparently it is not dependent on the speed of the memory in this case.

The cycle count of opt-mpeg2dec for a one threaded processor with the fastest memory was 433.629.256. We can compare that with the amount of 434.945.736 cycles, which is the amount of cycles it takes for opt-mpeg2dec to complete when memory of type 2 is used. There is a decrease in speed of $\frac{434.945.736-433.629.256}{433.629.256} \times 100\% = 0.304\%$ This drop in speed is really minimal. This makes it for multithreading not real easier to become more efficient. The amount of cycles it takes with a QTE of 80 can be read from table 3.6

With this table we can compare the change with respect of a slower cache and mul-

Number of Threads	*Forced Switch*	*Number of Requests*	*Number of Waits*	*Original Cycles*	*Cycle Reduction*	*Resulting Cycles*
1	4435186	1	0	439384757	0	439384757
2	4039404	1	793173	426915127	0	426915127
		2	0	426915127	5758574	421156553
3	3831218	1	1310595	424394097	0	424394097
		2	154799	424394097	10413644	413980453
		3	0	424394097	10860678	413533419

Table 3.6: Results of opt-mpeg2dec with a time quantum expiration of 80 cycles and a memory of type 2

tithreading. Again we see that adding the capability of handling a third parallel request to the bus interface is not efficient. The chip size and complexity would increase, while the performance gain is only minimal. This is $\frac{413.980.453-413.533.419}{413.980.453} \times 100\% = 0.108\%$. This performance gain is not worth investing time and money for development.

When we check the performance increase with multithreading when using a type 2 memory configuration, we see that the 1 threaded implementation takes 434.945.736 cycles. This value is acquired by increasing the QTE to such a high value, that no forced switch is generated. Because there is only one processor, there is no interaction between processors. Therefor no endless ll and sc loops of waiting for response of other processors is generated. The best performance is with 3 hardware threads and 2 bus interfaces. We ignore the 3 hardware threads with 3 bus interfaces line, because it was concluded that it would be inefficient to implement that third bus interface. The performance gain is thus $\frac{434.945.736-413.980.453}{434.945.736} \times 100\% = 4.820\%$. This is almost the same as the previous gain of 4.288%. A rise in performance was expected. The gain is the same as with the memory of type 1. This means that the longer idle cycles because of the slower main memory data fetches are not used more effectivly. The CPI with the memory of type 1 had a value of 1.087. Now the CPI has become $\frac{434.945.736}{398.854.744} = 1.090$. This is not a significant rise and thus we cannot expect to have a significant rise in performance. Though we did not expect a decrease in performance. In the next section we will see if this is case specific or that we see a trend. The memory can be downgraded to type 3 and the cache can be decreased to generate more misses.

In figure 3.6 we see the representation of the result of the normalized cycle count of the program opt-mpeg2dec with the slowest memory of type 3. However the cache still remains the same size. Again we see a QTE optimum of 80 cycles. However for the line representing the three hardware threads this optimal value is not clear visible. The optimum seems to be around 120 cycles. This is not a valid conclusion. In the previous figures (3.5,3.4) we see that all lines look about the same but they are on a different hight. The shape of line representing a two threaded implementation is the same as the representing a three threaded implementation. It stands to reason that this is a trend that continues with the change of memory. This is not visible here because there is a lack of results that is cause by the limitations of the simulator. For validity only the

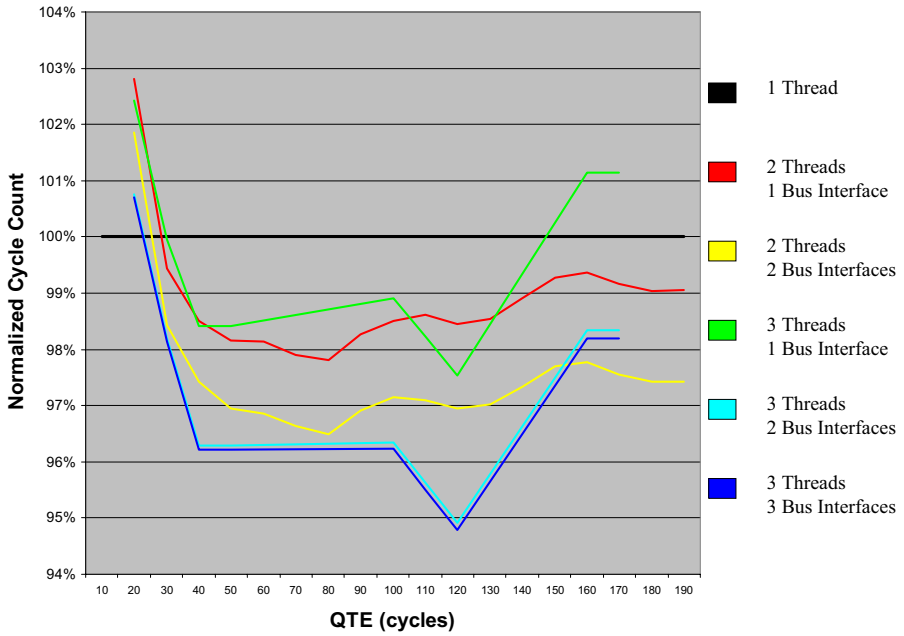

Figure 3.6: The relative change in cycle count, when using different implementations of multithreading, for the program opt-mpeg2dec as a function of the QTE for the slower memory type 3.

result of the 2 threaded implementation is considered.

By testing with a high enough QTE we get a cycle count of 437.578.472 as the result for the opt-mpeg2dec program with this type 3 of memory. This gives a CPI of $\frac{437.578.472}{398.854.744}$ = 1.10. The CPI keeps increasing, as expected, when the memory becomes slower. Since the CPI does not change that much, the performance speedup cannot be that much higher at best. If we look at the increase of multithreading benefit when decreasing memory speed, we are forced to look at the implementation with two hardware threads. The conclusion which can be drawn from these result, can be easily extrapolated when using more threads. In table 3.7 we see the speedup for two threaded implementation with

Memory Type	*1 threaded CPI*	*2 threaded CPI*	*Cycle Count Speedup*
1	1.087	1.054	3.058%
2	1.090	1.060	3.170%
3	1.097	1.061	3.304%

Table 3.7: The speedup of 2 hardware threads with 2 bus interfaces for the program opt-mpeg2dec, when dealing with different memory types

two bus interfaces for the maximal capability of parallel requests. The speedup does increase, but only minimal. The CPI does not increase that much with a slower memory type. The CPI/speedup ratio for the three different memory types are respectively 0.345, 0.334, 0.321. There is an decrease in ratio for slower memory types. This means that the speedup rises more than the CPI. Thus the interleaved multithreading will be more effective when there is a higher CPI. At least for this case. In the next parts we will see what happens when we increase the CPI even more by decreasing caches and using the three types of memory.

3.3.2.1 Effects of Reducing the Sizes of the Caches

When decreasing the cache size, the CPI decreases. In table 3.8 we can see what the values of the CPI are with the different types of memory. This table clearly show that

Memory Type	*Cycle Count*	*Instruction Count*	*CPI*
1	522.497.753	398.854.744	1.310
2	523.815.062	398.854.744	1.313
3	526.447.888	398.854.744	1.320

Table 3.8: The cycles per instruction (CPI) for opt-mpeg2dec and a data cache of 8 KB and different types of memories

the CPI has increased from the previous values. When having a data cache of 16 kilobits, the CPI's lay around the value of 1.090. With this configuration we can check whether the decrease in performance, caused by more cache misses, is somewhat compensated by the implementation of interleaved multithreading. Further we must check if there are any more changes in performance when we drop the cache size to half its original value.

In the figures 3.7, 3.8, 3.9 we see the results when we use a data cache of half the original size. The thing that immediately notices when comparing it to previous results, is that the quantum time expiration optimal value has shifted. Where the previous optimal value was about 90 cycles, while this time it resides around 30 cycles. This can be explained as followed. In a computer program there are normal execution sequences and infinite loops. Then there are two possibilities for a forced QTE context switch to happen. Either it happens during a normal execution sequence, which is unnecessary, or it happens during an infinite loop, which usually should have happened earlier. In this case we have a smaller cache, which means that there are more cache misses. In a normal execution sequence there can be cache misses. However usually all data in an infinite loop is already present in the caches. In effect forced switches in normal execution sequences happen less, because when a cache miss occurs, the QTE counter will be set to zero again. With a decrease in forced switches in normal execution, we have relatively more quantum time expirations in the endless communication loops. These loops need to be broken by a switch as soon as possible. This is done when the QTE goes down, as the optimal value of QTE with a smaller cache points out.

A primary data cache size with 8 kilobytes will produce more misses and thus generate a higher CPI. In the previous sections we already saw that the low CPI prevented the

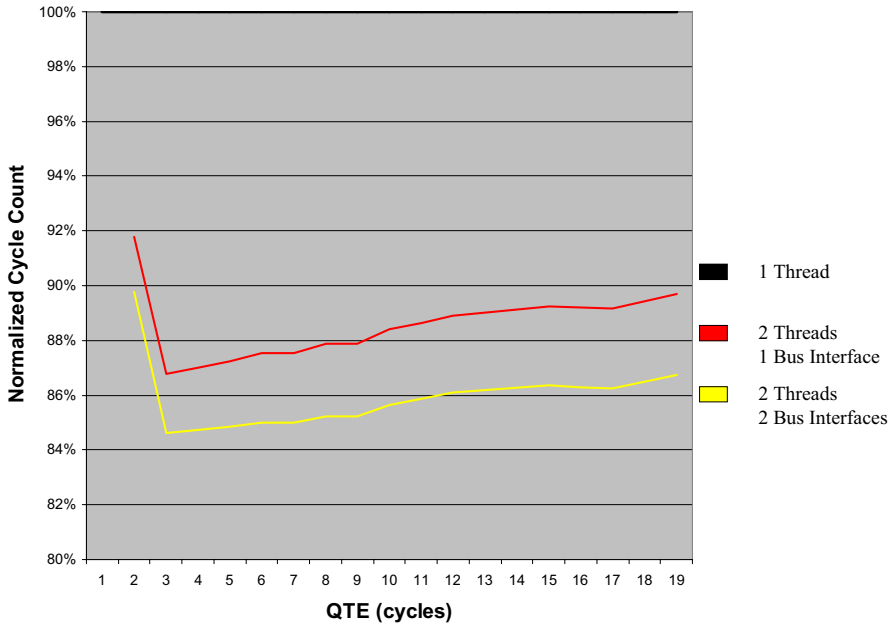

Figure 3.7: The relative change in cycle count, when using different implementations of multithreading with a smaller data cache of 8 KB, for the program opt-mpeg2dec as a function of the QTE.

multithreaded architecture from having a good enough performance boost. With a higher CPI this performance boost can increase. These performance gains are shown in tabel 3.9.

Memory Type	*Original Cycle Cost*	*Multithreaded Cost*	*Performance Boost*
1	522.497.753	442.085.902	15.39%
2	523.815.062	444.125.807	15.21%
3	526.447.888	445.801.396	15.32%

Table 3.9: The performance boost of a 2 threaded Trimedia with 2 bus interfaces and a data cache size of 8 KB.

In this table we see that the performance boost is significantly higher than the previous time with a bigger data cache. It is almost quadrupled. The gain is much more then the difference in cycle count. If for instance we take a look at two threaded Trimedia with two bus interfaces and a memory of type 2. The cycle count for a cache size of 16Kb without multithreading was 434.945.736. With a cache of 8 Kb this comes to a value of 523.815.062. This is an increase of $\frac{523.815.062-434.945.736}{434.945.736} \times 100\% = 20.43\%$. The performance gain for the big cache is $\frac{434.945.736-421.156.553}{434.945.736} \times 100\% = 3.17\%$ when taking

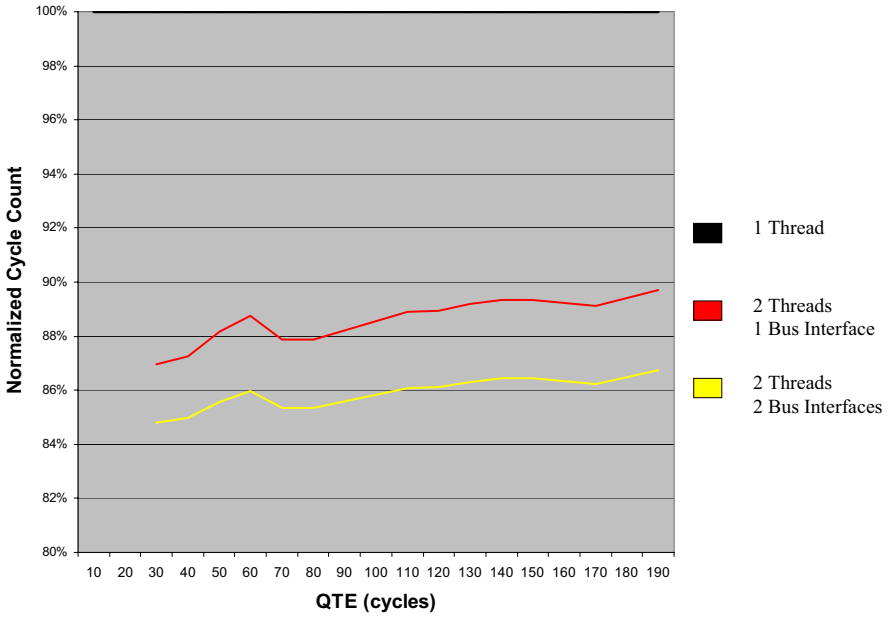

Figure 3.8: The relative change in cycle count, when using different implementations of multithreading with a smaller data cache of 8 KB, for the program opt-mpeg2dec as a function of the QTE for the slower memory type 2.

the QTE optimum. If we take the performance boost from table 3.9 we see a value of 15.21%. This is ralatively a much bigger increase than the 20.43% increase in cycles.

The CPI when using the cache size of 8Kb has a value of $\frac{523.815.062}{398.854.744} = 1.31$, when using memory type 2 and no multithreading. If we do use multithreading this value changes. In this case we choose the implementation of two threads and two bus interfaces, in which the CPI reaches a value of $\frac{444.125.807}{398.854.744} = 1.11$. A significant decrease, It even is comparable with the CPI when using double this cache size and no multithreading. Therefor it might be interesting t use this method to create smaller chips with a good performance.

3.3.2.2 Effects of Increasing the Sizes of the Caches

It is interesting to see what the effect would be if we increase the cache sizes, in stead of decreasing them. This can take away the negative effect of sharing a cache between the hardware threads. In the following test run, we take a 32 KBytes data cache and a 64 KBytes instruction cache. This is double the size we use in the original configuration. The results of this test can be found in figure 3.10. In this figure the most important thing we notice is that the multithreading architecture does not function well at all. Most of the time the 2 threaded TriMedia performs worse than the original single threaded one. When increasing the cache sizes, multithreading does not become interesting for this application. The reason for this is that by increasing the cache sizes, we decrease

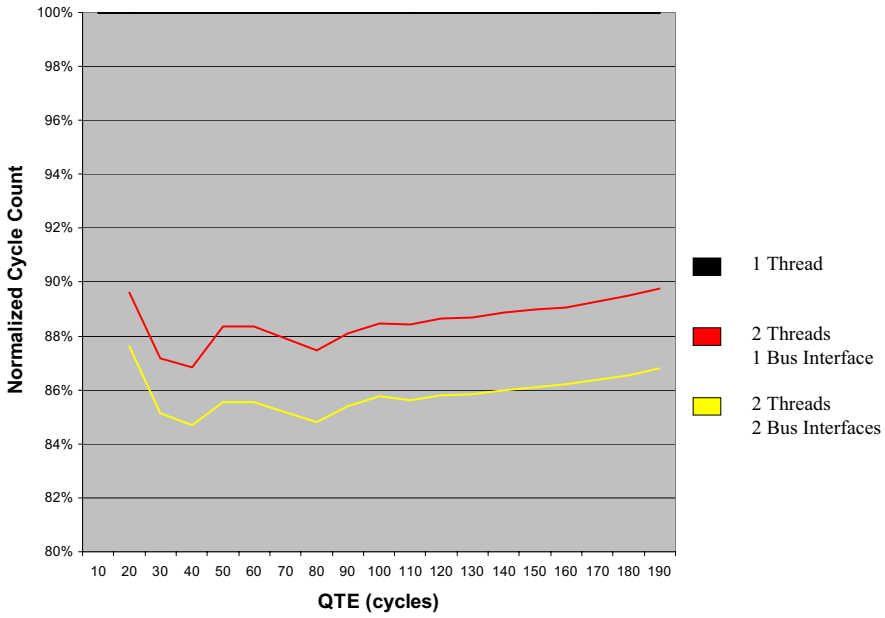

Figure 3.9: The relative change in cycle count, when using different implementations of multithreading with a smaller data cache of 8 KB, for the program opt-mpeg2dec as a function of the QTE for the slower memory type 3.

the number of cache misses. Therefor less opportunity is there for the multithreading architecture to optimize the stall time of the processor. In fact, the overhead of the thread switches is deteriorating overall performance. In this case there are just too many unnecessary forced switches because of QTE violations.

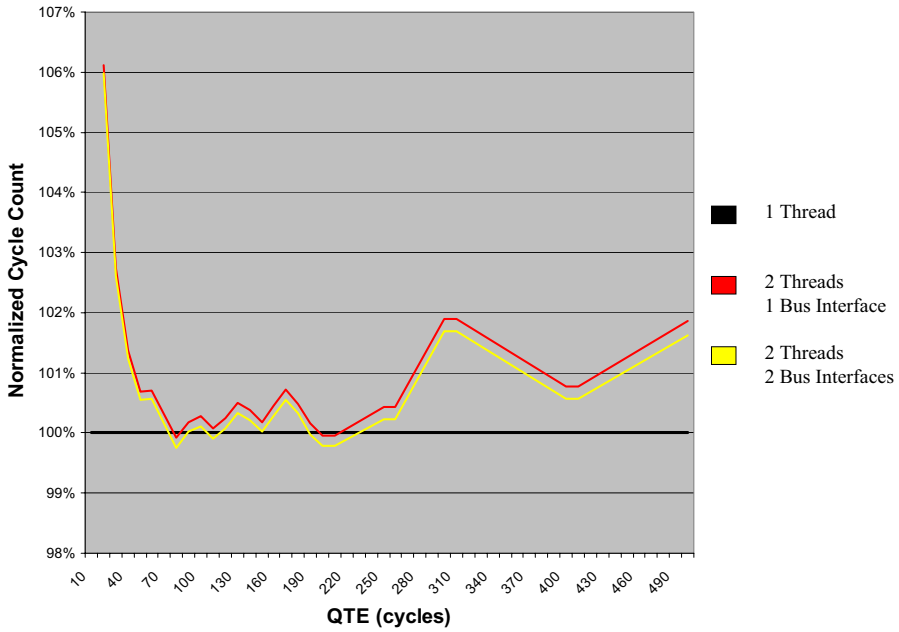

Figure 3.10: The relative change in cycle count, when using different implementations of multithreading with a larger data cache of 32 KB, for the program opt-mpeg2dec as a function of the QTE.

3.3.3 The Non-Optimized Mpeg2 Decoder

The program norm-mpeg2dec is, just as the previous program, an MPEG2 decoder. The most important change with the previous program opt-mpeg2dec is that this program is not optimized. It is just a normal off-the-shelf MPEG2 decoder. Therefore norm-mpeg2dec has a higher CPI. This might give the multithreading technique more occasions to improve the general performance of the program. The same tests as the previous program opt-mpeg2dec will be performed, but we limit ourselves to a two threaded multimedia processor. The reason for this is that the simulator did not produce reliable results for more threads. Though this is a loss for experimental exploration. We can compare these results with the results generated with opt-mpeg2dec.

When using the program norm-mpeg2dec with two threads, a data cache of 16 KBytes and a instruction cache of 32 KBytes, both with an associativity of 8 we can derive figure 3.11. We use type 1 memory for this test.

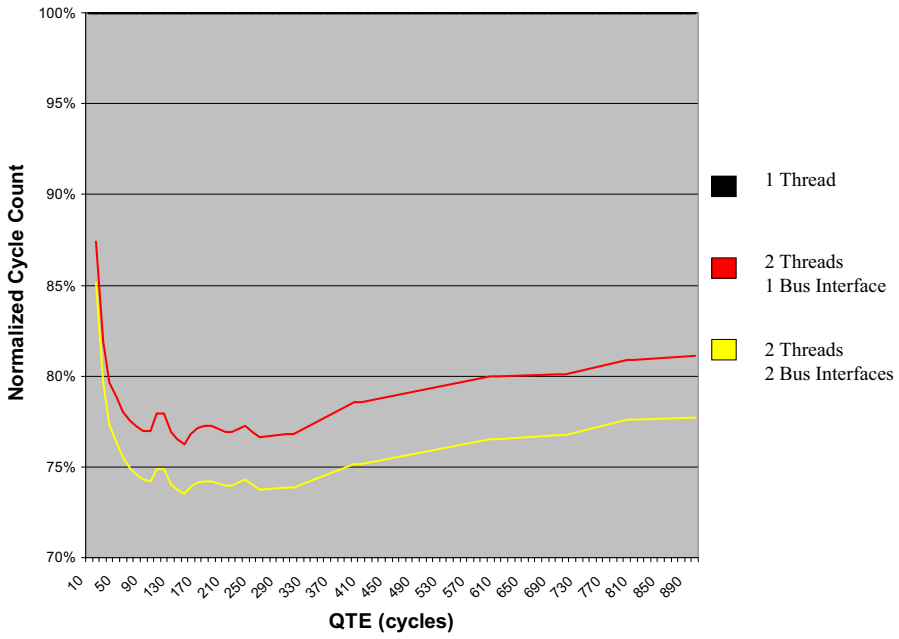

Figure 3.11: The relative change in cycle count, when using different implementations of multithreading, for the program norm-mpeg2dec as a function of the QTE.

In this figure we see the cycle count go down until the QTE reaches a value of 140 cycles. At this point we get the optimal performance boost because of multithreading. When we used the program opt-mpeg2dec with the same setting as used in figure 3.11, the optimal performance lay with a QTE of 90 cycles. This has quite a dramatic effect when deciding if multithreading is a good way to go. We either set the QTE to a hard value, with the cost of more cycles, but with less complexity of implementation. That complexity is what we get when we try to implement a variable QTE. A QTE that can change when handling different programs for better performance. Both options

seem inadequate. These results just make clear that implementing multithreading in only hardware is not that good. The software and the compiler should be adapted to multithreading. In this way the context switches can be given in software and the QTE becomes a less important issue for performance.

At 140 cycles we find that there is a maximal performance boost for the program norm-mpeg2dec. When running the program on a normal single threaded processor the amount of cycles it takes for the program to complete successfully is 639.290.309 cycles. At 140 cycles with a two threaded TriMedia with the capability to handle two parallel request, the cycle count is 469.278.404. This configuration gives us a performance boost of $\frac{639.290.309-469.278.404}{639.290.309} \times 100\% = 26.594\%$. If we compare this with the program opt-mpeg2dec we see an incredible difference. With the program opt-mpeg2dec it was only 4.288%. The performance boost is application dependent, which we already suspected. But the fact remains that the difference is quite high.

This significant increase in performance when comparing both programs can be found in the CPI. Opt-mpeg2dec had a CPI of 1.087. This number gave little room for improvement as described when discussing that program. When we use the program norm-mpeg2dec we get a instruction count of 455.668.232. The CPI of the program norm-mpeg2dec when multithreading is not used is thus $\frac{639.290.309}{455.668.232} = 1.403$. When the optimal multithreading configuration is used we have a CPI of $\frac{469.278.404}{455.668.232} = 1.030$. This is a significant speedup. We see that in this program there was room for improvement and that was filled up with the use of multithreading.

If we change the two threaded implementation, to the architecture that can handle only one request at the time, it takes the program 486.384.389 cycles to complete. The speedup is thus $\frac{639.290.309-486.384.389}{639.290.309} \times 100\% = 23.918\%$. This is relevant, because it shows that without the second bus interface the speedup is still significant. With a instruction count of 455.668.232 this architecture gives a CPI of $\frac{486.384.389}{455.668.232} = 1.067$. Again a very good CPI in comparison with the original single threaded version.

In the next two figures we see the effect of using the slower memory types. In figure 3.12 we use the type 2 memory and in figure 3.13. The first thing that we check is the optimal value of the QTE. In the figure 3.12 we see that the optimum lies at around 140 cycles. This is the same as before, when we used the fast memory configuration of type 1. But this does not hold for figure 3.13. When we use the slowest type 3 memory the optimal QTE lies around 170 cycles. In table 3.10 we see all the results when using the program norm-mpeg2dec with a data cache of 16KB and different types of memory.

The shapes of the figures of norm-mpeg2dec vary from those of opt-mpeg2dec in the way that both programs have the trend to significantly decrease the amount of cycles near smaller values for the QTE. But where opt-mpeg2dec shows a fast increase in cycle count after the optimal QTE value, the program norm-mpeg2dec shows only a slow increase. In other words, the program opt-mpeg2dec has a small range of values for QTE to achieve good performance,whereas the program norm-mpeg2dec has a much bigger range of values for QTE to acquire a good performance.

The number of table 3.10 give quite a good result for multithreading as a technique to improve performance. The previous program opt-mpeg2dec was manually optimize for TriMedia. The program norm-mpeg2dec was not. Most programs are not that optimized as opt-mpeg2dec. This observation make the results for norm-mpeg2dec even

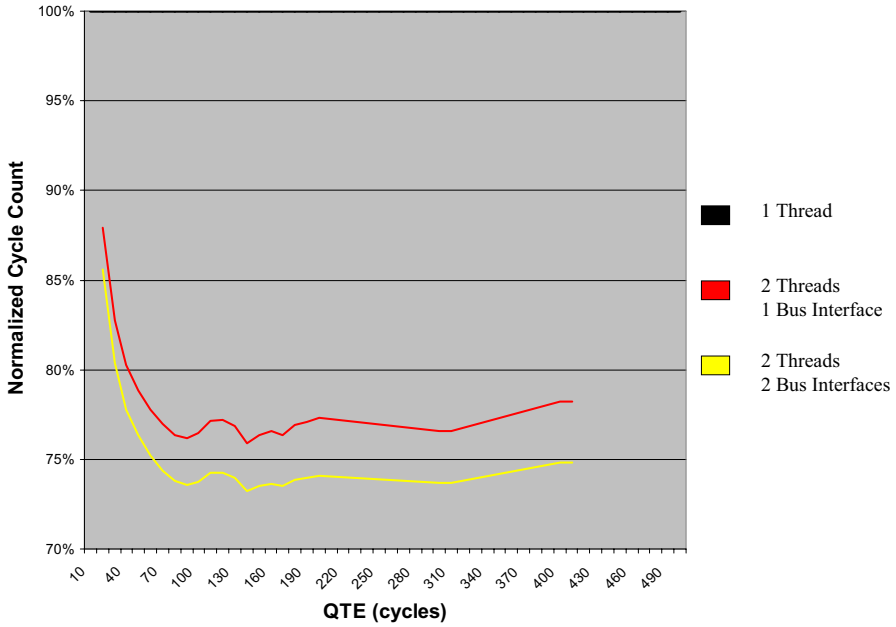

Figure 3.12: The relative change in cycle count, when using different implementations of multithreading, for the program norm-mpeg2dec as a function of the QTE for the slower memory type 2.

Memory Type	*Original Cycle Cost*	*Multithreaded Cost*	*Multithreaded CPI*	*Performance Boost*
1	639.290.309	469.278.404	1.030	26.594%
2	639.698.661	472.086.680	1.036	26.202%
3	640.515.365	470.143.302	1.032	26.599%

Table 3.10: The performance boost of norm-mpeg2dec as compared with 2 threaded Trimedia with 2 bus interfaces. The singlethreaded TriMedia has a CPI of 1.403.

more important. This is an improvement worthwhile of implementing if the costs are not too much. The costs in chipsize increase are handled in the next section.

3.3.3.1 Effects of Reducing the Sizes of the Caches

Just as with the program opt-mpeg2dec we now examine the result when we halve the data and instruction cache in size. This means that the data cache is now only 8 KBytes in size and the instruction cache has a size of 16 KBytes. This decrease in cache sizes will create more misses. In the figures 3.14, 3.15 and 3.16 we see the results of these tests.

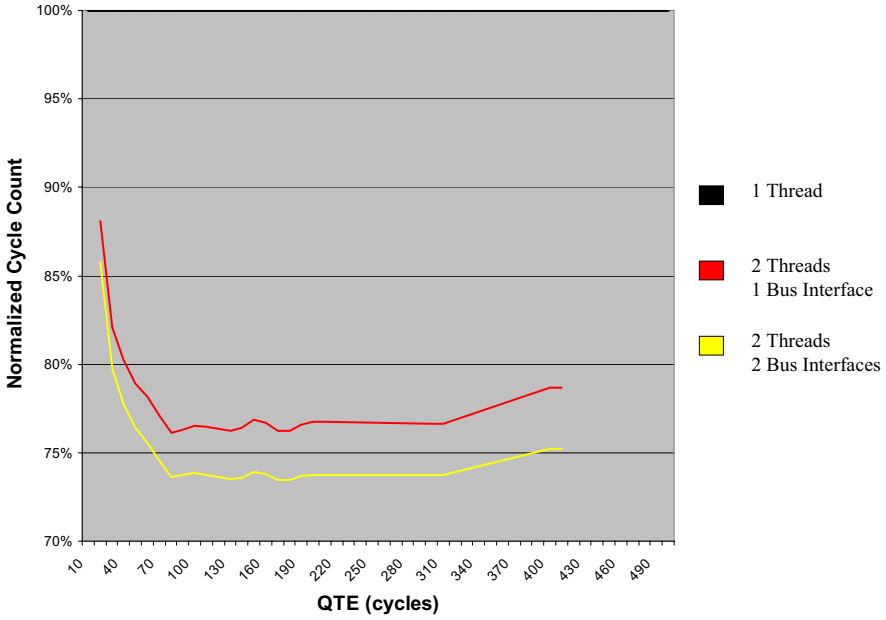

Figure 3.13: The relative change in cycle count, when using different implementations of multithreading, for the program norm-mpeg2dec as a function of the QTE for the slower memory type 3.

We see that the optimal values for the QTE lie around 80, 130 and 70 cycles for using memory types 1, 2 and 3 respectively. These numbers show us that it is almost impossible to predict what the optimal value for QTE will be. The range, however, for a good but not optimal performance, is quite large. When we thus predict a QTE, it is not that critical to be cycle accurate. This was almost the case when dealing with the optimized version of the MPEG2 decoder, the program opt-mpeg2dec.

Now we will examine the performance of the program norm-mpeg2dec with a data cache of 8 Kbytes. We take the optimal values of the QTE as we see in the figures 3.14, 3.15 and 3.16.

First we will take a look at the different CPI. The single threaded TriMedia needs 458.197.029 instructions to complete the program norm-mpeg2dec. Depending on the memory type, only the cycle count changes. A complete list of the CPI and cycle counts is given in table 3.11. When we take the normal memory speed of type 1, the cycle count for the single threaded TriMedia is $\frac{689.056.494}{458.197.029} = 1.503$. This 0.1 higher than the CPI when we have a data cache size of 16 KBytes. As expected an higher CPI. This should give the multithreaded environment more room to give a significant performance boost.

The performance boost of multithreading is calculated when using the optimal QTE for that configuration. We see in figure 3.14, that the optimal value of QTE equals 80 cycles. This configuration gives a cycle count of 547.928.678 when using only one bus

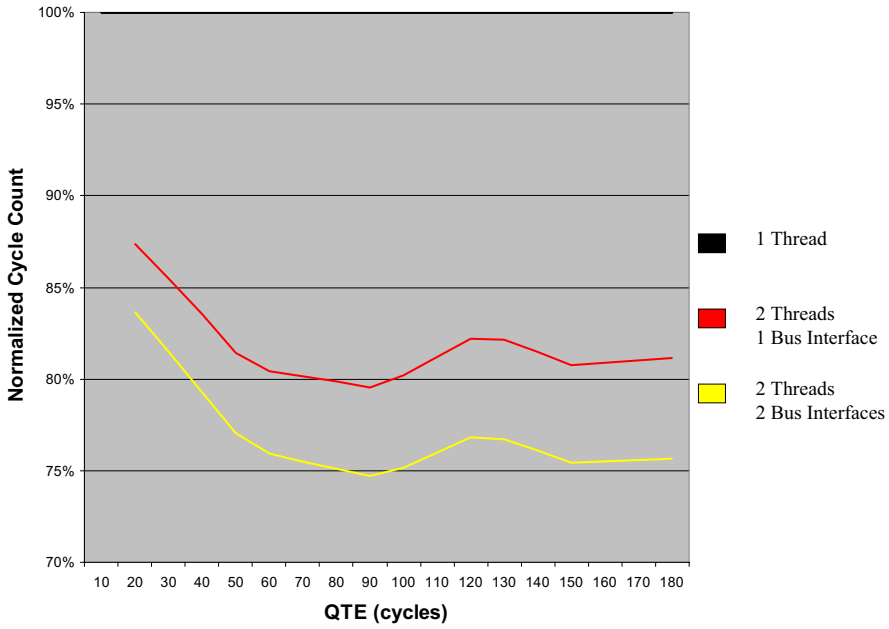

Figure 3.14: The relative change in cycle count, when using different implementations of multithreading with a smaller data cache of 8 KB, for the program norm-mpeg2dec as a function of the QTE.

interface and 514.848.299 when using two bus interface. When we compare that with the original single threaded cycle count of 689.056.494 we get respectively the boosts of $\frac{689.056.494-547.928.678}{689.056.494} \times 100\% = 20.48\%$ and $\frac{689.056.494-514.848.299}{689.056.494} \times 100\% = 25.28\%$. These performance boosts are less than the time we used the data cache of 16 KBytes. Apparently the extra cache misses were not compensated by the multithreading architecture.

Memory Type	*Original Cycle Cost*	*Multithreaded Cost*	*Multithreaded CPI*	*Performance Boost*
1	689.056.494	514.848.299	1.124	25.252%
2	689.464.846	520.912.448	1.134	24.447%
3	690.281.550	517.836.930	1.130	24.982%

Table 3.11: The performance results of norm-mpeg2dec when the data cache has a size of 8 KBytes. The multithreading architecture has 2 threads and 2 bus interfaces

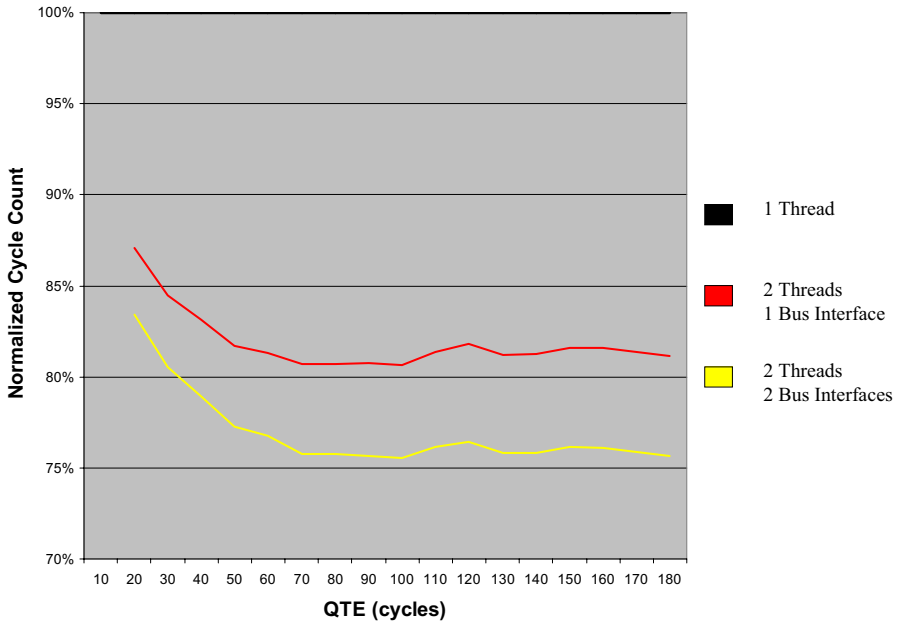

Figure 3.15: The relative change in cycle count, when using different implementations of multithreading with a smaller data cache of 8 KB, for the program norm-mpeg2dec as a function of the QTE for the slower memory type 2.

3.3.3.2 Effects of Increasing the Sizes of the Caches

A final test that is interesting is how much cache pollution is created because of the sharing of data caches. We get the results when we use double the cache sizes in respect to the normal configuration. That is, we use a data cache of 32 KBytes and an instruction cache of 64 KBytes. When no multithreading technique is applied, the cycle count equals 578.582.416 for 453.322.364 instructions. With multithreading we get the results shown in figure 3.17.

In this figure we can see the same shape of figure as all the other test with norm-mpeg2dec with other configurations. The optimal value of QTE is 150 and that gives values of 453.804.120 and 447.873.249 for a 2 threaded CPU with one bus interface and one with two respectively. With a data cache size of 16 KBytes these values were 486.384.389 and 469.278.404. This change in numbers is because of the decrease in cache misses. But not all of them can be contributed to cache pollution. Only those misses that are generated because the second hardware thread uses cache space that the first hardware thread still needs. These misses should disappear when only one hardware thread was used. But when using only one hardware thread, there are still misses, because the data cache can't hold all data that is needed.

The multithreaded TriMedia with 2 hardware threads and a 32 KBytes data cache has a performance decrease of 32.580.269 and 21.405.155 cycles. The first is for one bus interface and the last for two bus interfaces. For one bus interface this is $\frac{32.580.269}{453.804.120} \times 100\%$

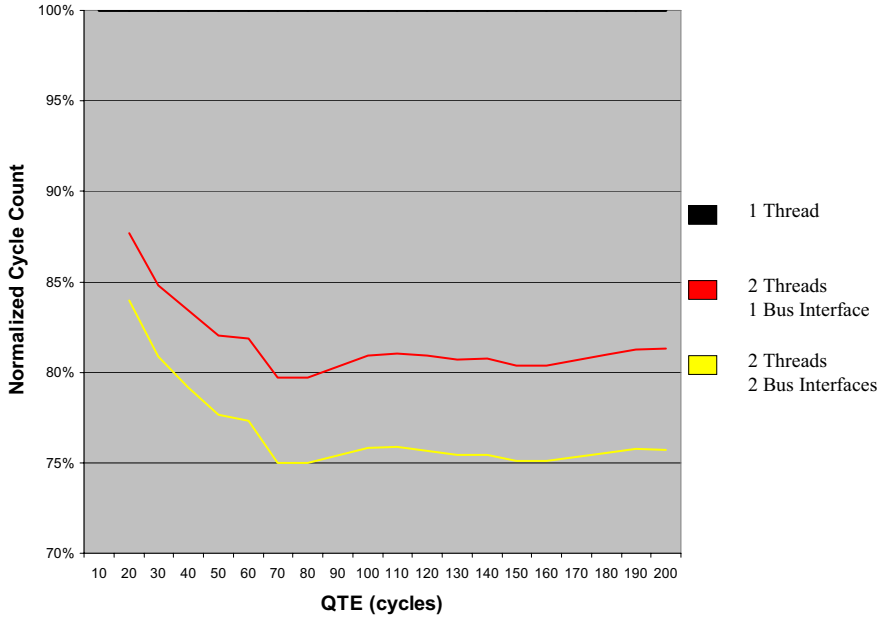

Figure 3.16: The relative change in cycle count, when using different implementations of multithreading with a smaller data cache of 8 KB, for the program norm-mpeg2dec as a function of the QTE for the slower memory type 3.

$= 7.179\%$. When we deal with two bus interfaces the difference is $\frac{21.405.155}{447.873.249} \times 100\% = 4.779\%$. Given the fact that we double the data cache for these differences and that the decrease is not a big percentage, we can conlude that the cache pollution with this configuration is only minimal. If we compare this percentage with the performance boosts, which lie around 20%, we see that indeed they are not that significant. We should even take into accoount that a part of these percenatges are because of cache size differences and not because of cache pollution. The fact remains that we can only conlude that for this program and these cache sizes, the cache pollution is nog significant.

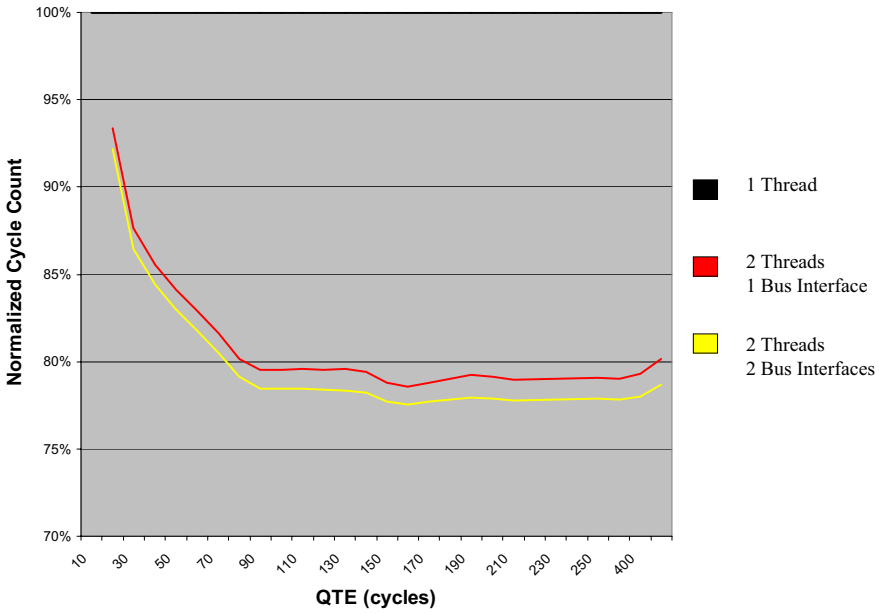

Figure 3.17: The relative change in cycle count, when using different implementations of multithreading with a bigger data cache of 32 KB, for the program opt-mpeg2dec as a function of the QTE.

3.4 Chip Size Increase Cost

Nothing comes for free. Multithreading will cost, among other, a larger chip. To give a idea how much this size will be we will give the percentages of growth. These percentages are calculated by using the number from a real TriMedia (TM3260). These area sizes were provided by Philips Semiconductors. Of course this will only give an indication how much the increase will be. The cost for extra logic that multithreading needs is estimated at a couple percent. The TM3260 has a dual-ported data cache size of 16 KBytes and an instruction cache of 64 KBytes. To implement interleaved multithreading we must duplicate the following items:

- The total register file
- The internal MMIO registers
- The interrupt vector registers
- The bus interfaces (optional)

With these numbers we could derive an increase in chipsize of around 17% per thread. If we want to incrase the number of bus interfaces, that will cost around an additional 4%. These numbers are already rounded up to include the extra logic that is needed for control. With these percentages we can derive formula 3.1.

$$I = 0.17 \times NT + 0.04 \times PR \tag{3.1}$$

In this equation, I stands for the increase in chipsize, NT stands for the number of threads we want and PR stands for the amount of parallel requests we want the multithreaded processor to be able to handle. Keep in mind that equation 3.1 only holds if $NT \in N$ and $PR \in N$.

3.5 Results Analysis

In the previous sections we got the results from the three applications we used. In this section these results will be analyzed even more.

The multithreaded architecture must improve performance, to be it worthwhile to implement it. This seems like a logical and abundant remark, but the tests with the program opt-mpeg2dec point out differently. The tests when running this program showed that multithreading does not necessarily increase performance. The program was optimized for the TriMedia instruction set. This showed in the fact that the multithreaded architecture did improve with the normal cache sizes of 16 KBytes for the data cache and 32 KBytes for the instruction cache. But this improvement was not that much. Overall improvement was about 2.7% for two threaded implementation. This is an improvement for normal cache sizes: data cache of 16 KBytes and instruction cache of 32 KBytes. In the future the caches will probably be bigger and in effect render multithreading completely useless or even worse. This was shown when running this program mpegdec.oo with the double the cache sizes. If we use a data cache size of 8 KBytes, accompanied

by an instruction cache of 16 KBytes, the performance did improve a bit to an overall speedup of around 15%. This showed that even an optimized application can benefit from a multithreaded architecture, but this is only the case if it generates enough cache misses. In this case because of a smaller cache.

But these speedup results were shown to be application dependent. In another mpegdecoder, norm-mpeg2dec, the same tests were run as the program opt-mpeg2dec. These tests gave very different results. As the program was not an optimized version for the TriMedia architecture, it generates more cache misses to begin with. This means that the multithreading architecture could be used more efficiently. When running tests with a data cache size of 16 KBytes, it showed a much bigger increase in speedup of around 25%, as compared with the speedup of 2.7% with the use of the program opt-mpeg2dec. With this program it showed interesting to implement a multithreaded architecture to improve performance. When using a smaller cache size of 8 KBytes it showed that the speedup when using multithreading was slightly less than normal. In this case it still was worthwile to implement it, but it showed that decreasing the cache size is not always benefitial for the performance speedup.

In the program norm-mpeg2dec we examined multiple threads. There we saw that adding more bus interfaces increases the speedup. This is how we would expect it to be. But the interesting part is when we look at the size that increase in speedup is. In both programs we examined the performance when using two hardware threads and one bus interface and two hardware threads and two bus interfaces. With an added bus interface the hardware threads do not have to wait to do their request via the bus interface. Naturally the implementation with two bus interfaces is better and the experiments showed that the performance increase because of a second bus interface is high enough to implement. But when we look at the third bus interface when having three hardware threads in norm-mpeg2dec we see that the third bus interface increase is not that much. In fact, it is even worth considering to leave out the third bus interface. This has the added gain, that the chip size does not increase that much, but even more the overhead cost of adding that second bus interface is greatly diminished.

The last chapter showed that the quantum time expiration (QTE) played an important part in performance gain, when we used a multithreaded architecture. The problem arose that we first said that the QTE should be as high as possible, because every time a forced switch occurred, it is an unnecessary cost. If there is a forced switch, it is either because of the hardware thread is in an endless loop of it is just doing its job for that long. In the first case the hardware switch was probably too late and it should never have entered such a loop. In the second case the hardware switch was unnecessary, the hardware thread could still do useful work. But the optimal value for QTE was relatively low. This makes it clear that we should make software additions. These software additions could for instance call for a hardware thread switch when it knows it enters an endless loop. Therefor the QTE can be set higher and only unexpected rare occasions trigger a forced thread switch because of QTE violation. Without the software enhancements and thus the low value for QTE, there is an additional problem of setting the QTE. The optimal value for QTE is unknown and is dependent of the application and the configuration. Having a variable QTE is only beneficial if that value is known. This can be done through test runs of the application, but this is not a practical solu-

tion, because there are too many applications and configurations to begin with. Another method is too check the optimal QTE during compile time. This is also an unpractical way of solving the problem. If we let the compiler determine the QTE, it is better to let the compiler add some forced thread switches in the code.

The multithreading architecture will make the chip size increase. We concentrated our findings and tests to the implementation with two hardware threads. This will add 17% to the original chip size. If we then want to add the second bus interface, it will take an additional 4% to the chip size.

In the end we need to answer the question if interleaved multithreading is a cost effective and efficient method to be implemented in future TriMedia processors. In this chapter we emphasized the two threaded implementation of a multithreaded architecture. When we choose to implement the two hardware threads in the TriMedia, it is beneficial for the speedup to add the second bus interface. This addition is not that much in cost like chip size and overhead, if we compare it to the benefits that are then gained. Thus if a two threaded TriMedia is implemented, the second bus interface is good addition. For the optimized version of the mpegdecoder opt-mpeg2dec the speedup results were definitely not worthwhile implementing a multithreaded architecture. However, these results showed an optimized version o the program for the processor and thus left little room for improvements. When we analyzed the performance of an unoptimized version of the mpegdecoder norm-mpeg2dec. These results did show a significant speedup and reason to implement a two threaded version of the TriMedia. In the end we have a multithreaded TriMedia with two hardware threads. This architecture has a significant speedup with a program which is not optimized for the TriMedia. Most programs are not that manually optimized as opt-mpeg2dec, thus other programs may exhibit the same speedup as the norm-mpeg2dec. We can only be sure of this, after more tests. The two hardware threads with two bus interfaces give an additional chip size of around 21%. In addition to that, there must be put time and money in development of such a chip. Therefor it is up to the manufacturer to decide whether these costs are worth the speedup.

4 Conclusions

This chapter we will conclude the thesis. First we summarize all the investigations, proposals, conclusions and remarks that were in this thesis. We end this chapter with recommendations for further studies.

4.1 Summary

In this thesis we set out to determine the the efficiency to implement interleaved multithreading in the TriMedia. The TriMedia is a VLIW processor, specially designed for multimedia applications. We chose the TriMedia, because multithreading was not implemented yet. Interleaved multithreading is then a logical option, since simultaneous multithreading is hard to implement in a VLIW processor because of sharing of resources conflicts.

We proposed the architecture of the interleaved multithreaded TriMedia. There are now several hardware threads. These hardware threads contain a TriMedia state. This means that the hardware threads contains all the information that is needed to start computing, from the last moment it went into idle state. There is only one hardware thread active at the time. When there is a thread switch, the original hardware thread switches to an idle state and another thread becomes active.

To make sure that all the information is stored in the hardware thread we need to duplicate some of the original CPU items to the hardware threads. The biggest among these are the 128 registers and the MMIO registers. We do not share the data and instruction cache. These shared caches make sure that the increase in chip size is not too much. Since the caches are a big part of the TriMedia processor, duplicating the caches would be a bad decision, because it is then better to use more single threaded CPUs. The shared cache architecture has the functionality, that the data brought to cache by one hardware thread can be used by another hardware thread.

We propose to use a pending buffer. This buffer makes sure that no two or more hardware threads ask for the data that another hardware thread just asked and is on its way to the cache. Further there is a copy back buffer that makes sure that the copy back of a line in the cache is finished before that line is called again by a thread. The third buffer is optional and dependent of the choice of implementation. Every request for data is fed through main memory or other locations through a bus interface. With multiple hardware threads we can have multiple requests. If we do not want to add more bus interfaces we use the so called memory subsystem buffer to sequentialize the requests through the bus interface.

To utilize the stall time, when data is fetched which is not in the primary cache, we propose to switch to another hardware thread when there is a primary cache miss. We switch to the next hardware thread using the round-robin method. This mechanism

does not hold, because of the existence of endless loop in the code. Therefor there must be a Quantum Time Expiration (QTE). This is a maximum amount of values a hardware thread may be active without primary cache misses. A QTE forced switch is not desired, but inevitable. The TriMedia uses Load Linked (ll) and Store Conditional (sc) operations to synchronize. If there is s thread switch during synchronization, then this synchronization will automatically fail and it must be tried again.

The architecture we propose is one purely based on hardware implementation, but for better performance we need software enhancements. These software enhancements can force a thread switch when that is required in stead of waiting for the QTE to do a forced switch.

We used three benchmark programs to determine the performance of the multithreaded TriMedia. The first program showed the benefits of the shared cache. But it was not a real application. The second program was a manually optimized mpegdecoder. This program showed that improvement with a multithreaded architecture is not as self evident as it may seem. The optimized program had a benefit, but not that much and only with a certain QTE. The third program was an unoptimized mpegdecoder. The results for this benchmark were a great improvement compared to the optimized mpegdecoder. It showed a significant increase of around 25%. This improvement is achieved with two hardware threads. We noticed that adding a second bus interface will give a better performance and at a relative smaller cost. The two threaded TriMedia with two bus interfaces does give significant speedup at relative small chip size increase of about 21%.

4.2 Further Research

The multithreaded TriMedia has proven to give a speedup, but it is recommended to test it with more and versatile programs. But this is not our main concern.

A forced switch because of QTE violation is almost never desirable. To increase the QTE or even delete it, we need to use software enhancements, so the compiler can recognize where a thread switch may optimize performance.

Furthermore it might be interesting to see the effect of a forced thread switch after two ll instructions without a sc instruction in between. It may be able to increase the QTE because of that. Still it remains to be seen how this all will affect the multithreaded CPU in a multiprocessor environment. The difficulties such as synchronization, coherency and consistency.

This thesis was focused on the development of a two threaded TriMedia. Multiple threads are not investigated thoroughly. To make sure that it is efficient we need to know the additional cost of chip size and control logic for synchronization.

Bibliography

[1] David E. Culler and Jaswinder Pal Singh, *Parallel computer architecture, a hardware/software approach*, Morgan Kaufmann, 1999.

[2] Anoop Gnupta, John Hennesey, Kourosh Ghacrachorloo, Todd Mowry, and Wolf-Dietrich Weber, *Comparative evaluation of latecy reducing and tolerating techniques*, Proceeding of the 18th annual international symposium on computer architecture (1991), 254–263.

[3] John Hennessy and David Patterson, *Computer architecture, a quantative approach*, Morgan Kaufmann, 1996.

[4] Jochen Kreuzinger and Theo Ungerer, *Context-switching techniques for decoupled multithreaded processors*, Proc. Euromicro'99 (1999), 248–251.

[5] Deborah T. Marr et al., *Intel technology journal*, Tech. report, Intel Corporation, February 2002.

[6] Peter R. Nuth and William J. Dally, *A mechanism for efficient context switching*, International Conference on Computer Design (1991), 301–304.

[7] Selliah Rathnam and Gert Slavenburg, *An architectural overview of the programmable multimedia processor, tm-1*, proc. Compcon '96 (1996), 319–326.

[8] Gerrit Slavenburg et al., *Tm1000 preliminary data book*, Tech. report, Philips Electronics North America Corporation, January 1996.

[9] Burton J. Smith, *Architecture and applications of the hep multiprocessor computer system*, SPIE (1981), 298:241–248.

[10] Paul Stravers and Jan Hoogerbrugge, *Homogeneous multiprocessing and the future of silicon design paradigms*, Proc. International Symposium on VLSI Technology, Systems and Applications (VLSI-TSA 2001) (2001), 184–187.

[11] D. M. Tullsen, S.J. Eggers, and H. M. Levy, *Simultaneous multithreading: Maximizing on-chip parallelism*, Proc. 20th Annual Symposium on Computer Architecture (1995), 278–288.

Curriculum Vitae

Stephan Suijkerbuijk was born in Bergen op Zoom, The Netherlands on the 25th of January 1980. From 1992 to 1998 he did his secondary education at "'t Rijks". There he studied at the level of VWO and successfully took the exams in the subjects: Dutch, Ancient Greece, English, Mathematics A, Mathematics B, Physics, Chemistry and Economics.
After the secondary education he became a student at the Technical University in Delft. He enrolled in the faculty of Electrical Engineering. There he got his Bachelor Degree in Electrical Engineering at the 26th of September in 2001. He chose to do his Master education at the group of Computer Engineering.
In December of 2002 he started an internship at Philips Research in Eindhoven. He was located in the cluster of Processor Oriented Architectures, which is part of the group of Information Technology. There he assisted the research team in developing a multiprocessor chip, under the supervision of Paul Stravers.
Immediately after that he worked on his Master thesis at Philips Research, on the same location where he did his internship. Paul Stravers and Stamatis Vasssiliadis were his supervisors.