Level one cache performance in a multiprocessor memory hierarchy

Cabrita Alves Martins, C.A.

Award date:
2005

Level One Cache Performance in a Multiprocessor Memory Hierarchy

By
C.A. Cabrita Alves Martins

October 2005

Eindhoven University of Technology (TU/e)
Philips Semiconductors DSP Innovation Center (DSP-IC)

Professor:
Prof. dr. ir. J. van Meerbergen (Philips Research, TU/e)

Supervisors:
ir. H. Dijkstra (DSP-IC)
Dr. ir. M. Bekooij (Philips Research)

Abstract

From the highest level of perspective, the system under study is a multiprocessor system where several processing blocks share access to a level two (L2) cache memory. The audio processing block of the PNX4010 is composed of two Dual REAL RD16025 Audio DSPs. The audio solution for the PNX4010 contemplates the use of level one (L1) cache memories. Cache memory is a small, fast memory unit located between the DSP and the next levels of memory. Dynamic behaviour of caches makes them difficult to apply in real-time DSP applications. This thesis focuses on the L1 cache performance in this context. Trace-driven simulation is a type of simulation where a trace record derived from an application execution is applied to a cache profiler (C model). With this type of simulation, it is possible to explore a wide range of cache configurations, isolate cache events and perform a study at a cycle level in a single processor context. The application traces used in the study were developed in such a way that they do not consider the presence of the cache in the system. The simulations produced accurate figures for the cache performance within a specific context. When the scope increased for real-time execution, factors like scheduling and interrupts caused by external agents should be taken into account. This was not achieved using trace-driven simulation. The decision was to move onto Instruction Set Processor Simulation (ISS Simulation) were the application is under our control and allows extending the scope of the study to a multiprocessor context. The streaming applications are modelled with SDF graphs. The SDF graph can be transformed to an HSDF and it is shown that the HSDF graph can be used to derive the minimal throughput of the system. Simple streaming applications are simulated in the ISS simulator assuming that external sources copy and read blocks of data to and from the data memory.

Acknowledgments

This thesis is a result of my graduation project performed at Philips Semiconductors DSP Innovation Center in Waalre. I would like to express my gratitude to Professor Jef van Meerbergen, Harry Dijkstra, Marco Bekooij and all the people involved for giving me the opportunity to carry out the work for this Master's thesis at Philips Semiconductors DSP-IC. The obtained results would not have been possible without the help, confidence and many hours of their valuable time to have interesting discussions in order to provide me with useful feedback, which helped me to address the problems from another perspective. I also like to thank all the people at the DSP-IC that in a way or another helped me during the course of this past seven months.

Table of Contents

Table of figures

Chapter 1
Introduction

A memory hierarchy is organized into several levels – each smaller, faster, and more expensive than the next level. The levels of the hierarchy usually subset one another. All data in one level can also be found in the level below, and so on until we reach the bottom of the hierarchy. The importance of memory hierarchy has increased with advances in performance of processors. The memory hierarchy is used to overcome the processor-memory performance gap. Consumer electronics devices like smart phones are incorporating all sorts of applications like for example personal information management applications with mobile phone capabilities. All these different applications are mapped onto a heterogeneous multiprocessor system. From the highest level of perspective the system under study is a multiprocessor system composed of several distinct processing blocks: an ARM processor, a Trimedia processor and an audio processing block composed of two DSP processors. The processing blocks share the access to a level 2 cache via a bus.

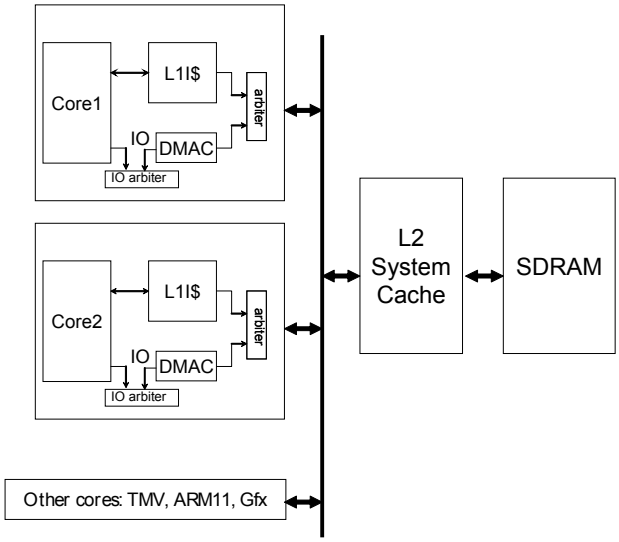


Figure 1 - RD16025C1 Memory Hierarchy PNX4010

The first part of this thesis focuses in the L1 cache performance of the audio processing block. It describes the model of the cache using a stand-alone profiler, which is a C program that mimics the behaviour of the cache. A trace application is applied to the profiler. From the simulation results conclusions can be drawn about the cache architecture and cycle behaviour for audio streaming applications.

The second part extends the scope and describes the modelling and simulation of a streaming infrastructure mapped onto a multiprocessor system. Analysis techniques are used to derive the temporal behaviour of the streaming application, which is described as a Synchronous Data Flow (SDF) graph and it is constructed in such a way that the worst-case arrival times of data can be observed. The guaranteed minimal throughput of the system is derived from the SDF graph using Maximum Cycle Mean (MCM) Analysis. An Instruction set simulator is used to verify the cycle true implementation. From the simulation results it can for example be concluded how the

performance of the application can be improved by applying different mappings of the application.

This work builds on existing programming environments, cache profiler and ISS Simulator available and developed at the DSP-Innovation Centre of Philips Semiconductors.

1.1 Background

While caches are familiar in RISC processors they have only been recently introduced in the world of DSPs. The main reason why caches have not been used in DSP processors is because they bring a fair amount of unpredictability into hardware systems. As DSP applications became more powerful and larger, there is a need to improve the memory system. The use of caches in DSP processors can be motivated partly by cost as a relatively small amount of cache memory can approximate the performance of a much larger local memory at a significantly lower cost. Thus, DSPs can become cost-effective solutions for a much wider range of applications.

Cache memory is a small, fast memory unit located between the CPU and the main storage unit. The cache makes use of the principle of locality; it typically stores the most recently used instructions and data and hence, increases the probability of finding information locally without having access the main memory. Classic concerns when using a cache is to maintain coherence between cache and main memory, and the criteria needed to refresh cache memory as the program is executed. Typically two cache architectures are used: the Harvard, where instructions and data caches are separate; and the Von Neumann, where instructions and data are unified.

The DSP Saturn has a double Harvard architecture with two data memory spaces (X, Y), and one program memory space (P). These three memory spaces are completely independent.
The DSP can perform three different tasks:

- Control the program flow
- Calculate data-memory addresses and access data memory,
- Execute the operations on the actual data.

As these tasks are relatively independent from each other, the DSP architecture is divided into three main blocks to support these activities and it is shown in *Figure 2*.

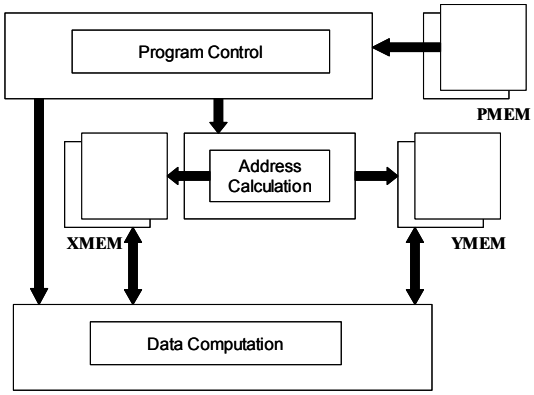


Figure 2 –Block diagram of the DSP

According to [1], two factors reduce the utility of caches in DSP applications: the data access patterns and real-time constraints. DSP applications tend to process large amounts of data and usually DSP applications display very strong locality in instructions accesses. Dynamic behaviour of caches makes them difficult to apply in real-time DSP applications. From the programmer viewpoint, it is hard to know what will be in the cache at the moment his program is executed.

Some caches provide features to help programmers design real-time applications, for example, the possibility to pre-load and lock segments of the cache. Using such a feature, portions of the cache can be loaded with specified blocks of instructions and data, and then locked so that the loaded instructions or data cannot be removed from the caches. Once portions of the cache have been locked portions become deterministic. The trade-off is that the effectiveness is reduced for the remaining instructions and data because the remaining cache size available is reduced.

DSP applications execution is typically dominated by a number of loops, making instructions caches appear attractive. Consumers typically have high expectation about the quality delivered by multimedia devices like DVD-players, audio, and television sets. These devices process data streams and have strict throughput and latency requirements. In order to meet these requirements the system must behave in a predictable manner such that it is possible to reason about its timing behaviour. The use of analytical methods is desirable because simulation can only be used to demonstrate that the system meets its timing requirements given a particular set of input stimuli. The design and programming of these real-time multiprocessor systems should be such that the real-time constraints are met, and the desired audio and video quality is delivered.

For performance and power-efficiency reasons, more than one processor often performs the processing in these systems. In many of these systems the input data is often provided by an external source, which provides a new input sample with a certain periodicity. This source can be an A/D converter and similarly and after processing the data is consumed by a D/A converter. These systems are often designed as hard real-time systems because input samples must be processed in time and output samples must be produced in time, since otherwise the input buffer overflows or the output buffer under-runs.

The analysis techniques that are described in this thesis are applicable for multiprocessor systems that execute applications described as Synchronous Data Flow (SDF) graphs. In [2] the temporal behaviour of the system is derived by constructing a SDF graph of the application that not only models the computation and the communication, but also, the effects on run-time arbitration. The guaranteed minimal throughput of the system is derived from the SDF graph by using Maximum Cycle Mean analysis [3].

1.2 Problem Definition

There are several goals in this thesis:

Trace-driven L1 cache simulation (single processor context):

- Assess the L1 cache architecture.
- Derive L1 cache performance figures.
- Derive the cycle behaviour for a single and multitask simulation.

Model and Simulation (ISS simulation) of streaming applications (multiprocessor system context):

- Shared memory multiprocessor architecture.
- Study alternative architecture.
- Derive predictability of the systems and reason about system parameters.
- Implement test cases to show benefits and drawbacks of both architectures and model of computation.

1.3 Outline of the thesis

This thesis is organized as follows. In the next Chapter, the trace-driven cache profiler is described. Chapter 3 presents a case study; from the simulation results, conclusions and proposals are derived. Chapter 4 describes the ISS simulator. In Chapter 5, the multiprocessor architecture is introduced along with the model of computation that is used to describe the streaming application. In Chapter 6, an example-streaming infrastructure is implemented and simulated. Finally, in Chapter 7 conclusions are drawn and recommendations are given.

1.4. References of Chapter 1

[1] Lapsley Phil, Bier Jeff, Shoham Amit and Lee Edward. *DSP Processor Fundamentals: Architectures and Features*. IEEE Press. 1997.

[2] Marco Bekooij, Orlando Moreira, Peter Poplavko, Bart Mesman, Milan Pastrnak, Jef L. van Meerbergen: *Predictable Embedded Multiprocessor System Design. SCOPES 2004: 77-91]*

[3] F Bacelli, G Cohen, G.J. Olsder, and J-P. Quadrat, *Synchronization and Linearity*, John Wiley & Sons, Inc., 1992

Chapter 2
Trace-driven L1 cache profiler

Caches are very important to embedded computer systems, especially due to the performance gap between microprocessors and memories is continuously becoming wider and the memory plays an important part in the overall performance. Simulation is an indispensable tool in the design of any computer system and can take many forms. Trace-driven cache simulation is frequently used to evaluate memory hierarchy performance and has been especially useful to evaluate cache performance. With this method, a software model of the cache being simulated is driven with a trace of memory references and can be used for system architecture study, determining cache miss-rates, bus traffic levels, or effective memory access times et al.

In Trace-driven simulation, a trace of instructions executed by the processor is recorded in a file and then later interpreted by the simulator. For memory hierarchy studies it is desirable to use long address traces that are accurate representations of real workloads. The software model is written in C and is open for modification – given enough time it can be created with whatever level of detail is needed to obtain results with the desired precision. In contrast, there is no control over the accuracy or completeness of the input trace data. The simulations are repeatable and allow cache design parameters to be varied so that effects can be isolated.

This chapter focuses in the cache profiler developed at the DSP-IC centre and is organized as follows. Section 2.1 describes the trace file format of the trace applications used in the simulations. Section 2.2 explains the simulator set-up and the simulation options. Section 2.3 describes the features modelled in the stand-alone profiler. In Section 2.5, the simulation assumptions are stated. The metrics used in the simulation are presented in Section 2.6. Finally, a small example illustrates the cache simulation.

2.1 Application trace file

A trace data record is a packet of information produced each time a traced event occurs during execution of an application, examples of traced events are a Read and Write. Trace data records contain metrics, which are relevant to a particular event. For example a time stamp, an event identifier and a memory address. The trace must contain enough information to reproduce the application behaviour.

According to [3], the process of generating a trace file can be difficult. In order to generate a trace file that can guarantee that the simulation is accurate several requirements must be fulfilled:

- The simulation must be correct.
- The simulator must also consider the behaviour of hardware components interacting with the processor, including peripheral devices.
- The simulator must take into account the timing of each instruction. This is necessary so that interrupts caused by external agents and timer ticks can be processed at precisely the right moments.
- The software being simulated requires real-time execution.

In the cache profiler, the program reads trace input from stdin. Each line of the trace file contains one record with following format:

[Timestamp][Memory][Type of access][Memory Address]

Exemplified as follows

…
8787 ZM R 011A
8787 YM R 6FF5
8787 XM W 9FF5
…

Where Z refers to the program memory, X and Y to the data memory, R for read access and W for write access. The address is a hexadecimal byte-address between 0 and FFFF. It is important to notice that the DSP processor can issue references to the three memories in the same cycle.

2.2 The stand-alone profiler

The profiler reads the new request arrivals from the trace file. When the request arrives in the profiler it checks the contents of the cache to see whether the corresponding content is present. If yes, the cache is left unchanged, otherwise the reference of the file corresponding to the request is fetched into the cache, which corresponds to a hit miss. If needed some data or instruction lines are removed from the cache according to a predefined removal policy. The simulator behaves in such a way that the output numbers realistically represent the cache events.

Figure 3 illustrates the simulation set-up. Simulation results are determined by the input trace and the cache parameters, which are the input of the simulator in the format of a trace file and parameter file that specifies the cache configuration

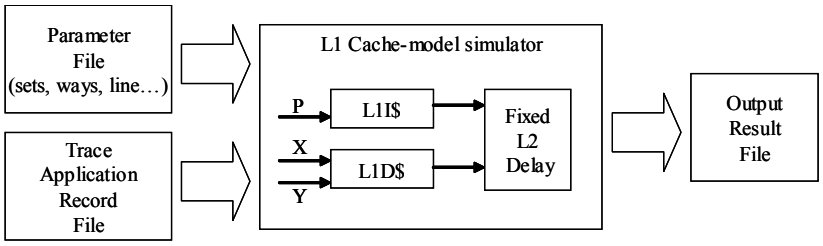


Figure 3 – Simulation set-up

The L1 cache parameters are set with a parameter file, which specifies:

- One or more application traces.
- Output result file
- Cache prefetch
- Number of ways
- Number of sets
- Line size
- Way locking

Through the command line it is possible to setup simulation alternatives. It is possible to explore the cache memory model by defining the data cache shared or separated for X and Y data; set the cache line replacement algorithm for three possible options; specify if the profiler uses one or more trace files, which means that a task-switching scenario is simulated. The profiler can produce extensive debug information. For each address related with a cache event can be associated a label. For example, a label that indicates if it is a read or write miss, the way, the set allocated, etc.

2.3 Cache model in the stand–alone profiler

The profiler models the behaviour of the cache.

2.3.1 Block placement, replacement and identification.

The data block is placed in the cache according to (*block address*) MOD (*Number of sets in the cache*). The cache has a tag address on each cache line that gives the block address. The tag of every cache line that might contain the desired information is checked to see if it matches the block address from the DSP. Also verified are some special bits. There is a way to know that the cache line does not contain valid information, the use of a *valid bit* (V) indicates if the entry contains valid data. In addition, the line also has a *lock bit* (L), which indicates whether a cache line is locked or unlocked, and in the case of the data cache a *dirty bit* (D), which indicates whether the data in the cache differs from its associated contents in the L2 memory. The number of bits for each field is dependent on the size of the cache. For example, a cache with 128 sets and with a line size of 8 words has 7 bits ($2^7 = 128$) in the set index field and 3 bits ($2^3 = 8$) for the offset field. The remaining ones are used for the tag in the tag field. *Figure 4* illustrates the cache set allocation.

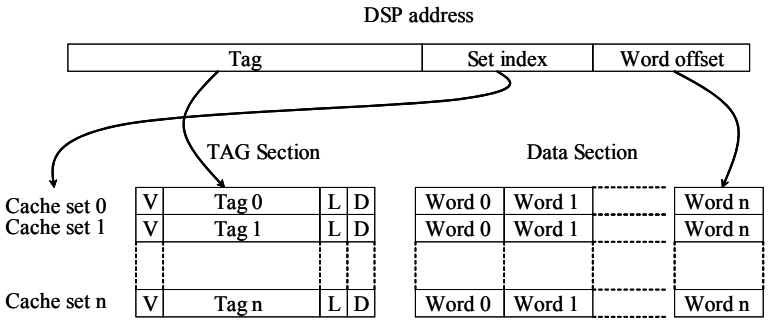


Figure 4 – Cache allocation according to the DSP address

A line replacement algorithm has to be implemented in order to allocate a new cache line if n>1 (i.e. cache is a n-way set associative). Three different cache line replacement algorithms can be set: first in fist out (FIFO), least recently used (LRU) and pseudo least recently use (PLRU). In FIFO the oldest cache line is discarded, this method is easier to implement when compared with the other two. In LRU, accesses to lines are recorded and the line discarded is the one that has been least-recently used. This method relies in corollary of locality: if recently used blocks are likely to be used again, then a good candidate for disposal is the least recently used. It is the most difficult method to implement because it can be complicated to calculate the LRU. An easier method to implement is the PLRU because it is based in a binary decision tree. For example, if

the data cache is four way-set associative, each cache set contains 3-bit PLRU table. Each table entry stores PRLU data and is updated on each read and write access to the associated cache set. *Figure 5* illustrates the PLRU algorithm according to a 4-way set-associative cache.

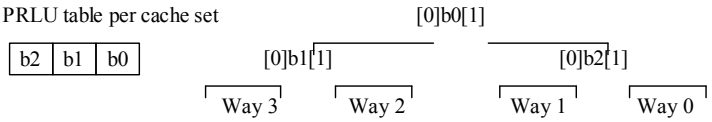


Figure 5 – PLRU cache line replacement algorithm

On a cache hit the values for b0, b1, and b2 are updated in the PLRU table. If for instance a cache hit occurs in way 3, then b0 and b1 in the PLRU table are both reset. Therefore, a PLRU entry indicates the most recently used way. On a cache miss, the least recently used is allocated, in order to do so the information contained by a PLRU entry is inverted.

2.3.2 Memory Management Unit

The behaviour of the cache can be controlled by means of the Memory Management Unit (MMU). The cache MMU allows partitioning of the external L2 memory into segments and for each segment an individual set of attributes can be programmed. A segment is defined by its lower and upper segment address boundaries. The MMU is programmed by application code in order to set an individual set of attributes to the segments defined. The simulator models the integration of a memory management unit for both Icache and Dcache and the effect of this configuration setting can be reproduced. Inside the model the address as read from the application trace is compared to a predefined address range. If the address maps into a segment area, the corresponding segment behaviour is carried out according to the predefined attributes.

The following attributes are modelled and can be applied to MMU data memory mapped segments:

- Division in cached/non-cached memory areas
- Data write mode - The data write mode determines which data write policy is applied to a cached segment. Data can be written according to a write-through or a write-back policy.
- Memory sharing - Memory sharing determines whether an MMU mapped data memory segment is shared or non-shared memory space.
- Locking, flushing, invalidating cache lines and excluding and locking cache ways.
- Program and data hardware prefetching - is the process of loading and temporary storing of the nearest next cache line from L2 memory into a prefetch buffer, in parallel with the data or program of the cache line currently being fetched from the cache (see *Figure 6*). The prefetch buffer stores only a single cache line. Only two prerequisites must be fulfilled to start a hardware prefetch: prefetching is enabled and a cache miss occurs

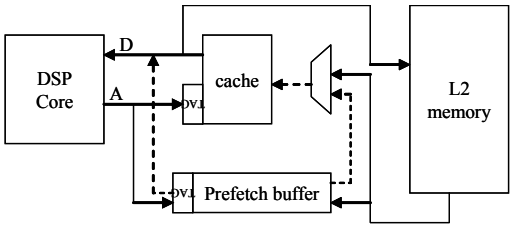


Figure 6 - Prefetching

2.3.3 Write policies

The writing behaviour to an MMU mapped data segment depends on the caching attributes that are applied to the segment. If the address does not match, the record is bypassed. If the address maps into the cached area, the corresponding write behaviour is carried out. *Figure 7* illustrates the functional model used in the simulator.

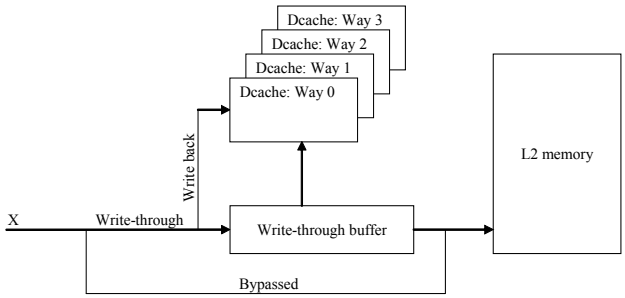


Figure 7 – Write functional model

The write-through buffer consists in a number of programmable entries (one to four), containing a whole cache set. The entry stores the cache address (the TAG and the index part of the logical address) and the corresponding cache line data. The write-through buffer entry is written when new write reference is issued by the DSP, all writes are buffered, regardless of whether they hit or miss in the Dcache. If the write-through buffer is full, the entry is first flushed and subsequently re-allocated to the new write reference. The use of the write-though buffer allows the processor to continue execution as soon as data is written to the buffer, thereby overlapping processor execution with memory updating.

In write-through mode, the Dcache is looked up for data reads and writes. Read misses cause a cache line replacement. Read hits cause the data to be read directly from Dcache and writes that hit update the Dcache. Since the data is simultaneously written to the write-through buffer, the Dirty bit will not be set. Write through is easier to implement than write back. The cache is always *clean* and the next level of memory has the most current copy of data.

In write-back mode, the Dcache is looked up for data reads and writes. Read and write misses cause a cache line replacement. Read hits cause the data to be read directly from Dcache. For writes that hit in the Dcache the dirty bit is set to '1' and the data is not written to the write-through buffer. A victimized cache line is written back to the main memory, only if it is dirty. The use of write back reduces the traffic to the memory traffic.

2.4 Cache performance issues

According to [2], the techniques used to improve the cache performance are categorized in four groups: reducing cache miss rate, reducing cache miss penalty, reducing cache penalty or miss rate via parallelism and reducing cache hit time.

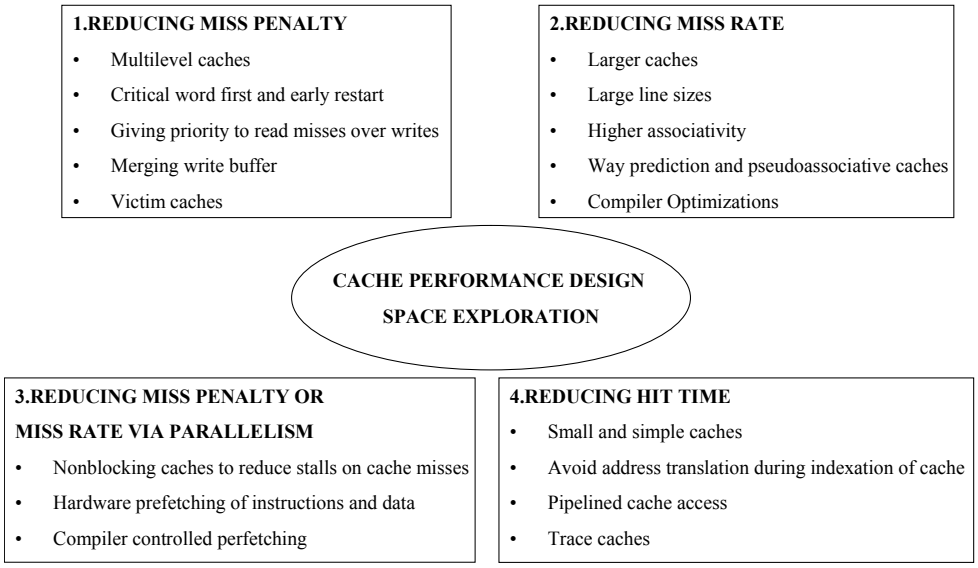


Figure 8 – Cache performance design space exploration

Figure 8 shows 17 possible measures divided in 4 sub-sets. Some of these techniques are addressed in the next with a brief explanation and possible trade-offs.

One method to evaluate cache performance is through the miss rate, which is simply the fraction of cache accesses that result in a miss. Reducing the miss rate is a classical approach for improving the cache performance but it is obvious that the miss rate is not the only component that should be considered. The time that it takes to fetch data from L2 memory upon a miss (i.e. the miss penalty) is just equally important as the miss rate. The hit time or the time that is necessary to access the cache is important because this time can limit the clock cycle rate.

As mentioned in Chapter one, the system being studied has three processing blocks sharing the access to the L2 memory. The miss penalty is dependent on the conflicts on the memory ports of the L2 memory, which relies on misses of other caches. There is no knowledge about the miss rate of other parties and hence, it is assumed an average time for L2 access. The focus is on the L1 miss rate, which is easier to evaluate in this context. The hit time requires a lot of knowledge about the physical implementation, which is out of the scope of this thesis.

The profiler is an open environment, which means that it can be modified. The profiler is an abstract model of the L1 cache that can be modified to incorporate many of the techniques shown above. A trade-off between design time, simulation time, scope and goals of the study confines this work.

The focus of the cache performance is on the miss rate and hence, the techniques that are modelled in the profiler are related with the improvement of the miss rate figure. But not all the

techniques to improve the miss rate are considered in the model, either because there was not enough time to be incorporated (e.g. way prediction) or simply because they are not possible to model (e.g. compiler optimizations). Various techniques shown in *Figure 8* are modelled in the profiler and they are explained in the next section.

In [2] a summary table shows the impact on the cache performance and the implementation complexity for each technique. Table 1 shows an illustration of some of the performance trade-offs to consider: +, - and 0 means that the technique improves, hurts and has no meaning in the factor, respectively. The complexity is ranked from 0 to 3, easiest to the hardest, respectively.

Technique	Miss penalty	Miss rate	Hit time	Hardware complexity
Multilevel caches	+			2
Critical word first and early restart	+			2
Giving priority to read misses over writes	+			1
Merging write buffer	+			1
Victim caches	+	+		2
Larger block size	-	+		0
Large cache size		+	-	1
Higher associativity		+	-	1
Way-predicting caches		+		2
Pseudoassociative		+		2
Compiler techniques to reduce cache misses		+		0
Nonblocking caches	+			3
Hardware prefetching of instructions and data	+	+		2 for instructions 3 for data
Compiler-controlled prefetching	+	+		3
Small and simple cases		-	+	0
Avoiding address translation during index of the cache			+	2
Pipelined cache access			+	1
Trace cache			+	3

Table 1 – Summary of cache optimization techniques

2.4.1 Techniques modelled in the profiler

The techniques shown in *Figure 8* that are modelled in the profiler are:

- Reducing miss rate (Larger caches, larger line sizes, higher associativity).
- Reducing miss penalty (write-through buffer).

- Reducing miss penalty or miss rate via parallelism (hardware prefetching of data and program).

The model that is presented in [2] states that there are three causes for cache misses:

- Compulsory misses – The first access to a cache line is always a miss, so the line must be brought into the cache. These are also called cold-start misses.

- Capacity misses – if the cache cannot contain all the lines needed during the execution of a program, capacity misses occur because of blocks being discarded and later retrieved.

- Conflict misses – Also called collision misses of interference misses. Will occur because of a cache line maybe be discarded and later retrieved if too many blocks map to the same set. Occur when too many lines map to the same set.

The most obvious way to reduce the cache miss rate is by increasing the size of the cache. This will reduce the capacity misses but incurs in a higher cost because the area of the memory is larger. Larger line sizes will reduce compulsory misses because it takes advantage of the principle of spatial locality (i.e. there is high probability that other word in the same line will be needed soon). The trade-off is that increases the miss penalty because larger blocks take more time to be fetched from L2 memory. At the same time, larger lines can increase conflict misses and even capacity misses if the cache is small. There is no benefit in reducing the miss rate if increases the penalty for accessing an L2 memory. Via higher associativity follows the two rules of thumb. The first states that an eight-way set associative cache is for practical purposes as effective in reducing miss rates as a fully associative cache. The second called 2:1 cache rule of thumb states that a direct-mapped cache size N has about the same miss rate as two-way set associative cache of size N/2. Higher associativity comes with the cost of higher hit time, which is the price for using more hardware.

When a cache is in write through mode, a write buffer is needed to reduce the miss penalty, which allows the processor to continue as soon as the data is written to the buffer, thereby overlapping processor execution with memory updating. The cache line present in the write buffer will be flushed when replaced by other line.

Hardware prefetching of instructions and data uses the prefetch of items before the processor requests them. The processor fetches two lines on a miss; the requested line and the next consecutive line. The requested line is placed in the cache and the prefetched line is placed in the prefetch buffer. The prefetch occurs when a cache miss occurs, first it is checked whether the requested cache line is already kept in the prefetch buffer. If so, it will be moved to the chosen victim in the cache. Then, the TAG file is updated. If the requested cache line not kept in the prefetch buffer, it will be loaded and written from the L2 memory into the chosen victim in cache. The potential success of prefetching is either lower miss penalty, or if it is started far in advance of needed, reduction of miss rate. This ambiguity makes this technique be considered in a separate section.

2.4.2 Techniques not modelled in the profiler

The techniques not contemplated are explained briefly and based on the descriptions given in [2].

- Critical word first – the missed word is requested from the memory first and it is sent to the processor as soon as it arrives, allowing the processor to continue execution.

- Early restart – fetches the words in normal order but as soon as the requested word from the block arrives, sends it to the processor in order to continue execution. These last two techniques only benefit designs with large blocks. Given spatial locality, there is more than random chance that the next miss is to the rest of the block.

- Merging write buffer – this technique involves the write buffer and improves the buffer efficiency. With *write merging,* the writes correspondent to different lines are merged in one entry. This optimization reduces stalls because the buffer does not fill so quickly and hence, it is not flushed so often.

- Victim caches – this buffer holds *victim* lines that are discarded from the cache because of a miss. On a miss the victim cache is verified to see if it contains the desired data. This measure is usually more effective with small and direct mapped caches.

- Way prediction - extra bits are kept in the cache to predict the way within the set of the next cache access.

- Compiler optimizations – This measure does not imply hardware changes. This technique uses the principle that software can be rearranged without affecting correctness. Techniques like loop interchange, where exchanging the nesting of loops can make the code access the data in the order that is stored. In this case more sequentially is obtained and hence, spatial locality is explored. Another example is blocking, where temporal locality is explored. This technique is used in dealing with arrays; the blocks within the array are formed in order to maximize the access the data present in the cache.

- Non-blocking caches are used to reduce stalls on misses - allows the data cache to continue to supply cache hits during a miss. The miss penalty is reduced but it requires out-of-order execution CPU.

- Pseudo associative caches - where accesses proceed just as in the direct-mapped cache for a hit. On a miss, however, before going to the next lower level of memory hierarchy, a second cache entry is checked to see if it matches there.

- Compiler-controlled prefetch – the compiler inserts prefetch instructions to request data before they are needed.

- Small and simple caches – One of the time consuming tasks on a cache access is to compare the index portion of the address and compare it with the address. Smaller and simple caches help the hit time.

- Pipelined cache access – pipeline cache accesses so that the effective latency of a first-level cache hit can be multiple clock cycles, giving fast cycle time and slow hits.

- Trace caches – A trace cache finds a dynamic sequence of instructions including taken branches to load into a cache block.

2.5 Cache model assumptions

Assumption 2.1 – *Stalls of program and data cache are added unconditionally*

The stalls of program and data cache are added unconditionally, but in reality some of the miss events can overlap. The model does not take into account that a stall from program and data cache can happen at the same time, and have overlap in time. In this specific sense, the profiler gives worst-case scenario because the numbers of stalls caused by program and data cache misses are summed.

Assumption 2.2 – *Not all the stall events are taken into consideration*

Stalls caused by the use of the write buffer are not modelled. The addresses with the same tag/set value are allocated to the buffer. A write reference cannot be stored in the buffer if the write's tag/set value does not match. Then the buffer has is flushed. At this moment the data is moved to the L2 memory, this is not counted as a stall-giving event. This can increase the stall figure.

A write queue can be placed in front of the write through buffer that allows scheduling of the flushing process with other L2 memory accesses, this it is not modelled in the profiler. This would prevent the stalling of the core because the writes could be stored in the queue while the write-through buffer is still busy with flushing or waiting for L2 access. This can improve the utilization of the L2 bandwidth.

In addition, the use of a write-queue buffer could signify that a data write miss or a data write back does not always have to give a stall. In this case a write is first stored in a write-queue before it finds place in the cache, and a write-back (=a victimized line which is dirty) is first stored in a victim buffer, which is moved to the memory in conjunction with the read buffer. These are all separate processes managed by a cache controller, which only stall the core when there is a conflict. Especially the execution of write backs can be done in background so the core will not be stalled.

The write-backs could be delayed as outstanding writes to L2 memory, which can take place in the background of the replacement progress in some cases. In that case, no stalls will be introduced. In the profiler, the write-backs are added to the miss rate equation.

Assumption 2.3 – *The average number of stall cycles is 3.5*

The number of stall cycles in case of a cache line refill depends on the synchronization between the L1 and L2 domain, it can be 3 or 4 cycles depending if it is an odd or even cycle.

2.6 Cache Metrics

The simulator outputs the number of misses: Prm (program read misses); Drm (data read misses); Dwm (write misses) and Dwb (data write backs). The results are used to compute the miss rates in the following way:

Data miss rate: $$Dmissrate = \frac{Drm + Dwm}{\#cpu\ cycles} * 100 \quad (2.1)$$

Data write back rate: $$Dbackrate = \frac{Dwb}{\#cpu\ cycles} * 100 \quad (2.2)$$

Program miss rate: $$Pmissrate = \frac{Drm}{\#cpu\ cycles} * 100 \quad (2.3)$$

Processor miss rate: $$DSPmissrate = \frac{Drm + Dwm + Dwb + \Pr m}{\#cpu\ cycles} * 100 \quad (2.4)$$

Stall percentage: $$Stall = \frac{(Drm + Dwm + Dwb + \Pr m) * L2\$latency}{\#cpu\ cycles} * 100 \quad (2.5)$$

2.7 Simulation Example

The following example is used to study the model accuracy. Two small trace files identified as *task1* and *task2* are used, the task switch occurs every five CPU cycles. The caches are of 64 words size (64 x 16 bits) with the configuration seen bellow in *Figure 9*. The Dcache is working in write back mode is a dual ported with shared X and Y data. The cache line replacement algorithm being used is the PLRU.

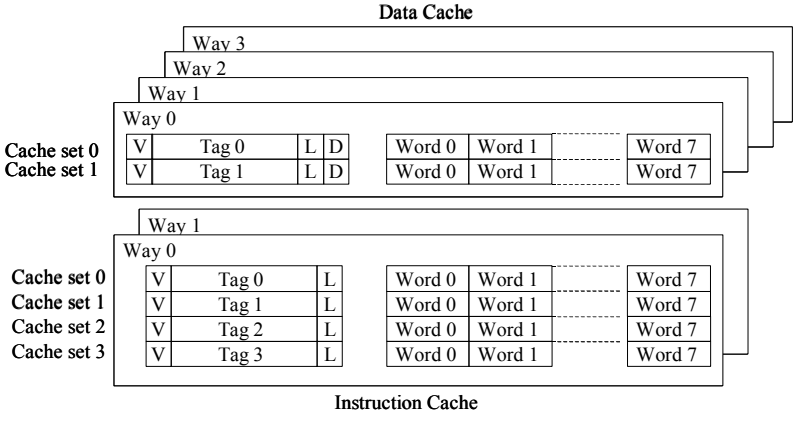


Figure 9 – Icache and Dcache configuration of the example

Figure 10 shows the predicted behaviour for a simulation using the example traces. The profiler is set-up to use a dual ported unified XY cache and the output results for X and Y memory will be summed and denoted with X reference.

task1	event	way	set	word	stall cycles
1 ZM R 9000	miss	0	0	0	3.5
1 XM R A000	miss	0	0	0	3.5
1 YM R C000	miss	2	0	0	3.5
2 ZM R 9001	hit	0	0	1	
2 XM R A0D0	miss	1	0	0	3.5
2 YM R C0D0	miss	3	0	0	3.5
3 ZM R 9002	hit	0	0	2	
4 ZM R 9003	hit	0	0	3	
4 XM W A003	hit	0	0	3	
5 ZM R 9004	hit	0	0	4	
5 XM W A0D5	hit	1	0	5	
6 ZM R 9005	hit	0	0	5	
switch to task 2					
6 YM W C006	miss	0	0	6	3.5
7 ZM R 9006	miss	0	0	6	3.5
7 YM W C0D5	miss	3	0	5	3.5
8 ZM R 9007	hit	0	0	7	
8 YM W CCCC	miss	0	1	4	3.5
9 ZM R 9008	miss	0	1	0	3.5
9 XM R AAAA	miss	2	1	2	3.5
10 ZM R 9009	hit	0	1	1	
11 ZM R 900A	hit	0	1	2	
switch to task 2					

task2	event	way	set	word	stall cycles
1 ZM R EE00	miss	1	0	0	3.5
1 XM R AA00	miss	2	0	0	3.5
1 YM R BB00	miss/wback	0	0	0	7
2 ZM R EE01	hit	1	0	1	
2 XM R CC00	miss	3	0	0	3.5
2 YM R DD00	miss/wback	1	0	0	7
3 ZM R EE02	hit	1	0	2	
4 ZM R EE03	hit	1	0	3	
4 XM W AA03	hit	2	0	3	
5 ZM R EE04	hit	1	0	4	
switch to task 1					
5 XM W CC05	miss	1	0	5	3.5
6 ZM R EE05	miss	1	0	5	3.5
7 YM W BB06	miss/wback	2	0	6	7
8 ZM R EE06	hit	1	0	6	
8 YM W DD05	miss/wback	0	0	5	7
9 ZM R EE07	hit	1	0	7	
9 YM W CACA	miss	1	1	2	3.5
10 ZM R EE08	miss	1	1	0	3.5
10 XM R ACAC	miss	3	1	4	3.5
11 ZM R EE09	hit	1	1	1	

Figure 10 - Cache simulation example

Figure 11 shows the expected events of the cache and has the following numbers: Prm = 6, Drm = 10, Dwm = 7, Dwb = 4. The output of the simulator validates the results as it is shown bellow.

```
RESULTS OF CACHE RUN
. . .
RESULTS OF MEMORY X
. . .
number of X reads        = 10
number of X writes       = 10
number of X read misses  = 10
number of X write misses = 7
number of X write backs  = 4


. . .
RESULTS OF MEMORY Z
. . .
number of Z reads        = 21
number of Z read misses  = 6
. . .
```

Figure 11 – Output results of the cache simulator

2.8. References of Chapter 2

[1] K. Flanagan, B. Nelson, J. Archibald, and G. Thompson, *The inaccuracy of trace-driven simulation using incomplete multiprogramming trace data*, In IEEE International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems. IEEE, 1996.

[2] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[3] Gurindar S. Sohi and Manoj Franklin, *High-bandwidth data memory systems for superscalar processors*, In Proc. of 4th Int. Conf. on Arch. Support for Prog. Lang. and Operating Systems, pages 53--62. ACM, 1991.

Chapter 3

Level 1 Cache profiling

This chapter describes the process of L1 cache profiling using trace-drive simulation. L1 cache profiling has three major goals. First, assess the cache architecture in order to derive a reference configuration that can provide the basis for further performance improvement. Second, derive performance figures for the cache. Third, perform an analysis at a cycle level. This means evaluation of the cache performance as function of execution time.

The starting point of the L1 cache profiling was that there was little knowledge about the application. The applications were developed without taking in consideration the presence of the cache and hence, so are the traces used in this work. The MST-BB Philips department in Nuremberg provided the traces. The manuals and helpful information provided by Nuremberg proved to be important in this study.

While the idea of trace-driven simulation is simple, it is time consuming since the simulation time is proportional to the length of the trace. For real applications, the traces can easily exceed millions of references. The time spent in simulating and the correctness of the evaluation are essential factors. Trace-driven profiling can easily lead to wrong assumptions, results and conclusions. The profiling process is highly application dependent, which means that knowledge about the application is essential. There are several and very important factors to consider when doing the profiling investigation. Among the key aspects are drawing assumptions regarding the applications, defining how to explore the architecture, evaluation of the accuracy of the trace, assessing the results and evaluate what are the sources of uncertainty in case they exist. These issues are developed in this chapter.

The chapter is organized as follows. Section 3.1 presents a study case based in the MP3 decoder and AAC decoder application traces. The following sections describe the knowledge acquired in the study case. Section 3.2 addresses the application and trace analysis. Section 3.3 describes the approach taken in order to explore the cache architecture. Section 3.4 explains briefly the iterative process between the profiling process and the development of the model. Section 3.6 reflects on what to consider when addressing the cache behaviour. Finally conclusions of the study are derived.

3.1 Case study

Two audio application trace files are used in the profiler simulations: the MP3 decoder [1] and the Advanced Audio Coding (AAC) decoder [2]. The MST-BB Philips department in Nuremberg provided the traces. The implementation of the algorithms is out of the scope of this thesis. The overview description of the algorithm presented in this section is based on the user manuals also provided by Nuremberg. The study addresses the goals referred above and describes the process of profiling.

3.1.1 MP3 and AAC decoder application and trace analysis

In the MP3 Decoder three main functions are defined, which are called by the DSP main program: MP3_Init, MP3_Synchronize and Mp3_Process (granule based processing). The

function MP3_Init initialises all static variables of the MP3 decoder and it is only called once at the very beginning of MP3 decoding phase. The initial synchronization is done by function MP3_Synchronize. This function is called repeatedly until synchronization is achieved or the end of the bit stream is reached. After successful synchronization, the function Mp3_Process performs header and side information decoding as well as data processing until the end of the bit stream is reached or an error condition occurs. Mp3_Process performs a granule (sub-frame) based processing and generates per call and per stereo channel 576 output samples (1152 words of 16 bits). The MP3 decoder required memory resources (in 16 bit words) are shown in *Table 2*.

In the AAC Decoder two main functions are defined, which are called by the DSP main program: AAC_Init and AAC_Process. AAC_Init is called once before playing a new AAC file. It may already decode a header found in the input buffer and request more data before calling AAC_Process. The AAC_Process procedure decodes one block of data and produces 1024 stereo output samples (2048 words of 16 bits). The required memory resources (in 16 bit words) for the AAC decoder application are shown in table *Table 3*.

The Data types of the applications are defined, according to [3] the data types present in the traces are:

Input/Output – This part of RAM is used for storage of input/output parameters and streaming data of the application.

Static RAM – This part of the RAM remains unchanged between successive procedure calls. Different procedures inside a package (e.g. speech encoder or speech decoder) may use different static RAM sections.

Scratch Data RAM – This part of the RAM is commonly by the subroutines used of all applications and stores intermediate results. No data is kept between successive calls. Scratch data is located in two different RAM banks, called SCRATCH_A and SCRATCH_B.

Data ROM – The ROM area is used to store constant values or tables.

Program ROM – The ROM area is used to store program.

Stack – in respect to an object allocated on the stack in connection with a subroutine call.

It is possible to track the procedure calls present in the traces by knowing the program address of those calls. This address search is performed in the trace file. This gives a more clear idea about what is present in the trace: what are the functions executed in the trace, how many times they are executed and their duration. The analysis reveals that the MP3 decoder trace contains one call to the initialisation procedure, one to the synchronization procedure and eight calls to the process procedure as shown in *Figure 12*.

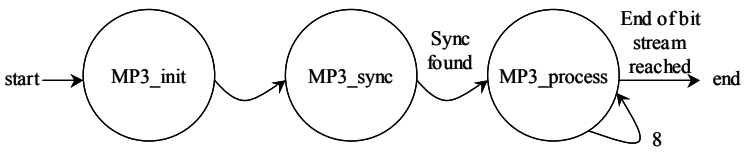


Figure 12 – MP3 decoder flow diagram

The AAC trace contains one call to the initialisation procedure and six calls to the process procedure, as illustrated in *Figure 13*.

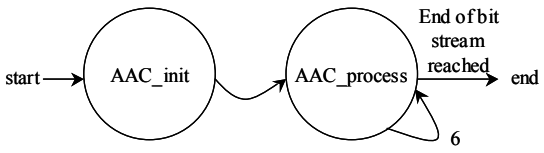


Figure 13 – AAC decoder flow diagram

The memory references present in the traces give some insight about how the algorithms were developed. Using a small program to count the number of reference locations present in the traces, the number for each memory is obtained, the results are shown in *Table 4*. "ZM R" is the number of program reads, which is a number equivalent to the number of DSP clock cycles. The table shows that there is a levelled number between X and Y references which can lead to the conclusion that the algorithm does not explicitly divide X and Y data as in many DSP applications where for example Y data is used for filter coefficients. The same program is used to count the number of different memory references in the traces, *Table 5* confirms the situation. This may be proven to be a decisive factor in the data cache memory model to use and can lead to decisions of using a XY shared data cache model or a separated X and Y model.

An analysis in the trace files reveals the number of samples that are consumed and produced by the algorithms. The MP3 decoder application reads a different number of input data samples in each call of the MP3_Process procedure, as shown in the *Figure 14* below.

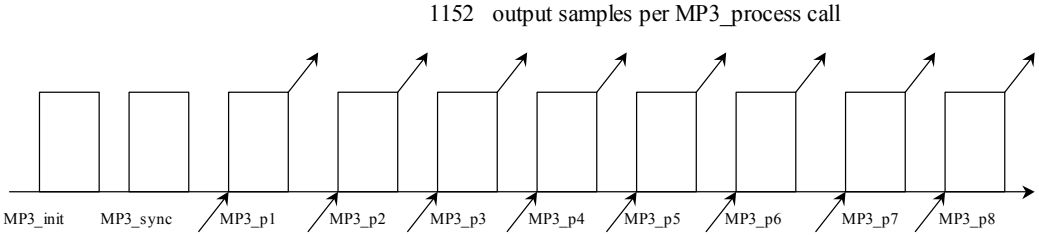


Figure 14 – MP3 decoder trace

For the AAC trace,

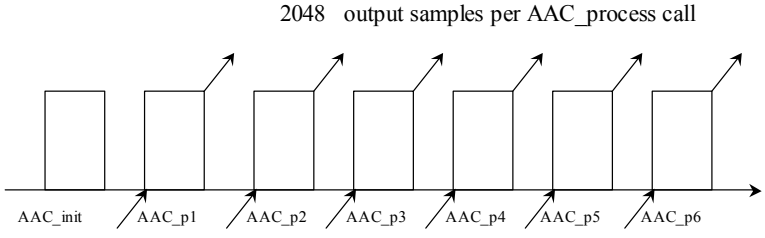


Figure 15 - AAC decoder trace

An inspection of the memory events in the input and output buffer data address range show that:

- The cache lines correspondent to the input data samples are not invalidated before a new block is processed (i.e. before a new process routine execution). The analysis shows that the same data is from the input buffer segment read. The profiler includes invalidation and it is unpractical to solve use it because the number input samples read by the MP3 decoder differs from process call to process call. This number must be estimated because new input samples causes cache misses.

- The output data segment is used for intermediate storage of computation results and it is difficult to know at what point in time the output data block is produced just by looking at the trace. The cache lines corresponding to the output samples are not flushed when the data output block processing is produced. The cache lines corresponding to the output data must be updated in the memory once a frame is produced in the end of each call and the data correspondent to the cache lines must be invalidated. Knowing the expected number of output samples (1152 for MP3 and 2048 for AAC), an estimation of the number of cache lines to be flush can be obtained.

For the input stream, the miss rate is for a worst-case situation estimated.

For the MP3 decoder, 977 new data input samples is the worst-case situation, per routine call.

```
977 / 8 = 122.125              // 8 words is the size of a cache line size assumption
122.125 * 8 = 977 misses       // 8 is the number of frames produced in the trace
977 / 655734 = 0.15%           // 655734 is the number of DSP clock cycles in the trace
```

The estimation of the number of cache line flushes considering a number of 1152 output samples is: 1152 / 8 = 144 cache line flushes.

For the AAC decoder, 1024 new data input samples is the worst-case situation, per routine call.

```
1024 / 8 = 128              // 8 words is the size of a cache line size assumption
128 * 6 = 768 misses        // 6 is the number of frames produced in the trace
768 / 1711445 = 0.045 %     // 1711445 is the number of DSP clock cycles in the trace
```

The estimation of number of cache line flushed considers 2048 output samples.
2048 / 8 = 512 cache line flushes.

The trace does not consider the invalidation and the flushing of the input/output data samples and hence, the program does not consider the interaction with the cache. The conclusion derived from the estimations is that the I/O data stream may not introduce a big uncertainty in the miss rate and in the stall percentage. The estimation must be used to correct the performance miss rate figure result.

For real-time execution, there is the assumption that a periodic source produces (i.e. transfers to the input buffer) samples, and a periodic sink and consumes samples (i.e., reads from the input buffer), respectively. This can have impact in the cache behaviour and is an uncertainty in the analysis. For example, interrupts coming from an external source that produces (e.g. A/D converter) or consumes (e.g D/A converter) must be considered in the real-time execution because data must be supplied and consumed. No interrupt code is present in the trace, the unpredictability of an interrupt could have consequences that are not easy to estimate because it not possible to predict when the interrupt is going to occur, what happens and what is the impact in the cache performance.

The conclusion here is that the applications perform a block read and write operation to data already available in memory, and is not involved in data IO by interrupt.

From the analysis of the application traces, the following conclusions are stated:

Application trace 3.1: *there is no explicit X and Y data, i.e. the applications uses both with a similar weight*. This factor influences the memory model.

Application trace 3.2: *the MP3 trace has duration of 655734 DSP clock cycles correspondent to 3.27 ms of CPU time (200 MHz). Each call to the main routine reads a varying number of samples and produces 1152 samples.*

Application trace 3.3: *the AAC trace has duration of 1171445 DSP clock cycles correspondent to 5.85 ms of CPU time (200 MHz). Each call to the main routine reads a 1024 input samples and produces 2048 samples.*

Application trace 3.4: *the uncertainties related with the I/O stream can be estimated.* The estimation showed that the impact from I/O stream is not significant.

Application trace 3.5: *the trace does not contain interrupt code and does not consider the interaction with the cache.* The application was developed not taken into account the presence of the cache. The simulation assumes that the data is already available in memory.

3.1.2 Cache architecture exploration

The first decision about the cache architecture respects the data memory model. Therefore, it is important to confront different cache memory models. For the study case the following cache models are confronted:

- XY shared cache memory space (double ported memory).
- Separated X data cache and Y data cache memory (one port memory).

Table 6 shows the results of the simulations using the two models for fixed total sizes of 8Kw and 16Kw. The results show that a data cache model with a shared XY data has a lower miss rate when compared with a separated X and Y data cache model. The double-ported data memory model achieves better performance for several cases when the size is the same.

The conclusions from the results is that in case of separate X$ and Y$, a write to X can never be read by Y (and vice-versa) so it gives a miss. This can be done in shared XY, and the current applications exploit this feature, which was confirmed by the application trace analysis for the MP3 or AAC case and this is the reason for the higher miss rate. For the following simulations a share XY$ data model is used.

Profiling supports the decision on parameters for the cache architecture. The three basic parameters of a cache are the size, the line size, and degree of associativity. In cache designs, the problem often resides in the choice of these three parameters. In order to derive the reference cache architecture some basic parameters are kept fixed and the focus is on these three parameters. It is not reasonable to analyze at this point every possible cache configuration and the focus is on the cache basic parameters. For the simulations, the cache line replacement algorithm is the PLRU and the MMU is programmed in order to set write-back policy for all memory range. In order to get data on the miss rate, a stand-alone trace profiler processes the traces over a wide

range of cache configurations. In order to simulate several cache configurations in a single run, a script file programmed in perl is used. The program and data cache have the same configuration for each data point. The results are shown in *Figure 51-56.*

The following observations can be derived from the *Figure 51 and 52:*

- The miss rate decreases with bigger the cache line. This is the result of less compulsory misses. We can conclude that there is locality applications
- Decreases with the bigger cache sizes. This is the result of less capacity misses.
- The increase of the cache associativity has a bigger impact with a cache size of 8 Kword (I$=D$). This is the result of less conflict misses.
- A direct-mapped cache size 8Kw has about the same miss rate as two-way set associative cache of size 4kw for cache line size of 2, 4 and 8w. For cache lines bigger than 8w, the miss rate of a direct-mapped cache size 8Kw is higher than a n-way set associative cache of size 4kw. This is the result of the increase in conflict misses.
- The flat line indicates that the application fits in the cache and hence. This is the result of compulsory misses.

From figures *Figure 53-6*

- The data cache contributes the most to the overall miss rate percentage (i.e. the data cache misses are dominant).
- The misses of program and data cache are added unconditionally, but in reality some of the miss events can overlap. The model in the profiler does not take into account that a stall from program and data cache can happen at the same time, and can overlap in time.

3.1.3 Reference cache architecture

A working assumption for the cache architecture is chosen based on the profiling results and a physical implementation study, which is out of the scope of this thesis. This working assumption leads to an 8 Kw size 4 way-set associative for Dcache and 8 Kw size 2 way-set associative for Icache. The cache line size is 8 words (8×16 bits). The cache architecture is shown in *Figure 16*.

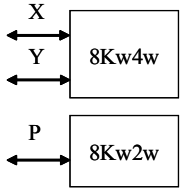


Figure 16 – Reference cache architecture

The conclusions for the basic cache architecture configuration:

Cache architecture 3.1: *The model for the data cache is a shared X and Y data double ported memory.*

Cache architecture 3.2: *Instruction cache is a 8Kw size 2 way-set associative memory and the data cache is a 8Kw size 4 way-set associative double ported memory with shared X and Y data. The cache line size is 8 words.*

3.1.4 Algorithms dynamic behaviour

The performance impact of each procedure from both algorithms is verified. The simulation is done with the reference architecture. It is possible to track the procedure calls present in the traces by knowing the program address of the process calls. This address search is performed in the trace files. By knowing this, it is possible to know the duration of each process computation. By doing post-processing in the debug data produced by the profiler it is possible to filter the miss rate for each main procedure of the applications.

The results are shown in *Table 7-8 and Figure 57-58* and they show clearly that there is a peak in the miss rate at the start of the execution, which is the consequence of a cold start (i.e. compulsory misses). The cache is empty when the initialisation procedure is executed in and noticeable is the dynamic behaviour because there is a significant variation in each process execution, which visible by the fluctuation of the miss rate.

The worst-case execution time of an application is dependent on the input data and sources of dynamism present in the algorithm (e.g. condition statements, data-dependent iterations). The highest and lowest miss rate value per process computation of the applications is:

- MP3 decoder – highest value 2.08%, minimum value 0.89%
- AAC decoder – highest value 2.60%, minimum value 1.99%

One conclusion is that the worst-case execution time of a task can be estimated with the worst-case value of the miss rate assuming a fixed penalty.

3.1.5 Impact of data types

The profiler produces debug information of the cache events. With the knowledge about the memory structure of the application, post-processing is executed in the debugged data. The addresses and events are filtered and is possible to know the influence of each application data type in the data miss rate figures. The results are shown in *Table 9-10 and Figure 59-60*. The x-axis of the figures denotes the segment data type and range. In the MP3 application the static data and data Rom are dominant in the overall MP3 miss rate results. In the AAC application, the output buffer and data Rom are responsible in the overall AAC miss rate results..

A possible recommendation for cache operation configuration optimisation:

- For MP3, lock in one of the cache ways data referent with the static data
- For AAC, lock in one of the cache ways data referent with the data ROM

This is achievable by configuring a specific address range (i.e. MMU model inside the cache) to be locked in some way of the cache.

3.1.6 Performance as function of execution time

The simulation is performed in a task-switch context. Every time a miss occurs, the period is extended to 3.5 cycles. When a task switch occurs, the entire data and instruction cache are invalidated and this incurs in extra stall cycles (1 cycle per cache line). Furthermore, the dirty lines present in the cache are flushed and updated in the memory. It is possible to use the profiler in order to obtain information on the number of dirty lines present in the cache when a task-switch occurs. The number of number dirty lines in a task-switch shows that approximately 70% of the data cache lines have to be flushed.

In the first simulation the task switch occurs every 1 Mcycles of execution time (cpu cycles + stall cycles), The DSP miss rate is 1.9%. In the second simulation The task switch occurs every 1 Mcycles of execution time (cpu cycles + stall cycles), The DSP miss rate is 3.5%. This is the result of the cache cold starts (i.e. the cache is empty), the tasks have to be initialized each time a task-switch occurs and this as impact in the cache miss rate. *Figure 17* shows the result.

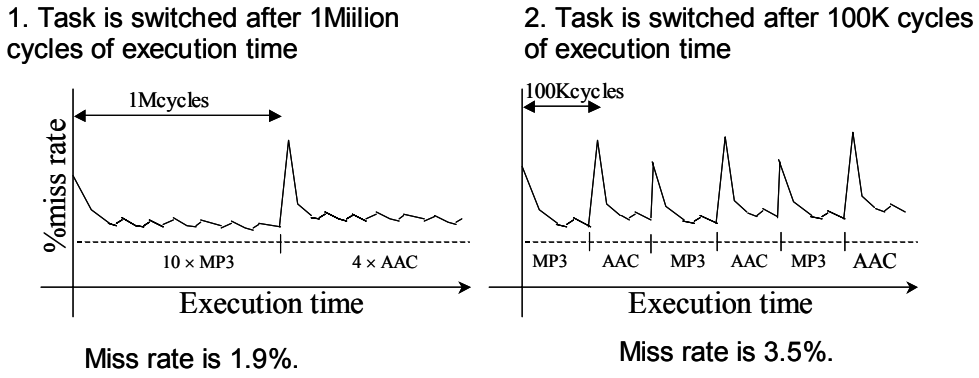


Figure 17 – Task-switching simulation as function of execution time

There is a problem in this simulation that affects its accuracy. In order to perform this simulation, the trace files were concatenated in order to have longer duration (1M cycles execution time), which means that the same data is being used. Still there are important details that are illustratable from *Figure 62*. Notable are the peaks when a task switch occurs and the fluctuation of the miss rate during the execution of the tasks.

It is clear that the effect on the miss rate is small for a budget period of 5ms, which is proven to have no significant impact on the miss rate value. If the budget for each task decreases, the miss rate will increase as a consequence of more frequent cold starts.

In Round Robin the approach is at a frame granule level. The tasks may be chosen to execute once or more, for example task one may execute 2 times while task 2 executes just once, or the opposite. The approach is easier and the controlling task just needs to verify if the tasks have the necessary requirements to execute. The execution will stop when a process routine has finished executing the number of times that was predetermined to run.

3.2 Application and trace analysis

It was shown that the cache profiling process is highly dependent on the application trace running in the system and that there is difficulties to obtain knowledge about the application itself. There was no prior knowledge about the application because it was not developed at the DSP-IC department. The study is done without access to the code and there is no knowledge about the behaviour of the algorithm. It is very important to have a certain degree of knowledge about the application because this could mean less time spent on simulation and more importantly, avoid wrong assumptions and conclusions. Hence, it may be required to perform several analyses in the trace in order to obtain that data.

The basic knowledge starts by the duration of the trace, it may happen that the application trace file only contains a few milliseconds of simulation and this may not represent the worst-case execution of some application algorithm. The designer must know what main procedures are in the applications and at which point they are executed. By knowing this, it is possible to know the duration of each procedure execution and make individual performance estimations in each of the procedures. Some insight about the algorithm behaviour can be derived from this analysis. In streaming applications a number of samples are consumed in each execution of the application and a number of samples are produced. Knowing the number of samples it is possible to quantify the impact of I/O streaming in the overall cache performance. A similar idea is to verify the impact of each of the data types of the application. An application has predefined number of data memory map segments. Knowing the memory requirements of the application and the boundaries of the memory map, the profiler produces debug information in order to verify the performance of each data segment in the overall cache performance.

The DSP processor is able to issue up to three memory references per processor cycle. The situation does not happen in every cycle but it is important to analyse the frequency and quantify the memory references present in the traces. Usually Y data in DSP applications are used for example for filter coefficients and this usually means that the occurrence of different X data references is much higher. This can be a decisive factor in the data cache memory model to use and can lead to decisions of using a XY shared data cache model or a separated X and Y model. Optionally, it can lead to the decision of a smaller cache memory size for Y and a bigger for X.

The analysis leads to a number of conclusions that could be very useful in the analysis and limit the time spent in doing simulations and drawing conclusions about the boundaries of the analysis.

3.3 Cache architecture

The simulation of trace applications in the profiler supports the decision about the cache architecture configuration. Choices about the memory model and the cache configuration parameters are looked into. The process can be divided in two, first to derive a cache reference configuration and secondly the optimisation of the configuration.

The most logical step is to start by the cache memory model and evaluate the cache parameters that dominate the cache performance, the cache size, the degree of associativity and the line size. Options to consider in the memory model are for example to use a separated memory for X and Y data with different sizes and perhaps configurations, or a shared XY data memory with one or two ports. The decision-making is naturally the result of the trade-off between performance and cost (i.e. area and power consumption). The basic cache parameters (size, associativity, line size) should be the main focus of analysis when assessing the cache basic architecture. This means that other parameters should remain fixed for a wide range of cache simulations with these three

changing parameters. Other configuration options may be in a later stage explored. For example, using cache way locking to keep data that is frequently used in one of the cache ways should be used for optimization evaluation of the cache performance.

3.4 Debug information produced by the profiler

Detailed information on the operation of the cache model can aid the analysis; this is supported by the profiler and it can be modified and adapted in order to help the profiling process. More, the debug information produced by the profiler can be subjected to post processing in case it helps the study. For example, in order to evaluate the impact of each data types (see section 3.1.5) the profiler produces debug information that associates an address with a cache event (e.g. read or write miss). The profiler produces this debug information, and post processing can be used in order to find how many misses are associated with a certain address range. First by filtering the addresses corresponding to an address range (equivalent to a memory segment from a certain data type) and then counting the number of misses associated with that range. Then, knowing the number of DSP cycles that are present in the trace it is possible to calculate the miss rate of each data type.

3.5 Cache performance

The miss rate figure is independent of the cache penalty and consequently these are two separate components developed in the model. The execution time is dependent on both miss rate and cache penalty, which makes it more complex to model. The percentage of stall cycles is dominated by the penalty caused by the fetching data to the L2 memory and in this model is considered to be fixed. But there are other factors that cause stalls as seen in the case study. The model in the end should accurately mimic the cache behaviour and give detailed debug information because they are essential in the development, accuracy and correctness of the conclusions of the study.

The focus of the cache performance is on the miss rate. The first goal is to derive a miss rate figure and the second is to address real time execution. The assumption of a fixed penalty when a miss occurs is due to the fact that the cache performance of other processing blocks is not known. Their cache performance will influence the latency but because there is no information available at this stage the latency is kept static. The miss rate figure gives insight about the stall percentage of the processor. The performance as function of execution time allows investigating for example if the application has dynamic behaviour. It can give the idea if the miss rate has for example behaviour with bursts, their duration and peak values. Important is to derive the execution time for certain granularity or the response time of a task that is running concurrently with one or more other tasks. In this case scheduling must be addressed.

The execution order of a program influences decisively the program miss rate figure. If the program contains many jumps to sub routines means a higher number of misses or by the contrary, if the program is relatively sequential means a less number of misses. The impact is often even bigger in the data because the data is not usually accessed sequentially and this means that the probability for a higher number of misses is higher.

The cycle behaviour is difficult to evaluate. First, it requires that the trace represents accurately the run of an application in the processor and it should contain all the interaction with the hardware peripherals and must take into account the timing of each instruction. This is necessary

so that interrupts caused by external agents and timer ticks can be processed at precisely the right moments. Secondly, the stall events must be modelled correctly in the simulator in such a way they happen at the exact point in time as they would in the implementation. The stalls caused by fetching a cache line from the L2 cache, the cache invalidation in case a task switch occurs or the input data samples have been consumed, flushing the cache in case new data has been produced et al. This is in the hands of the designer and with time all the stall events can be incorporated in the model.

The question that must be answered is until what extent (limits) we can assess accurately the cache performance based in trace-driven simulation and the profiler described in chapter two. The answer is depends on several factors:

- Does the profiler accurately mimic the cache behaviour?
- Does the trace application reflect the worst-case behaviour of the application?
- Does it accurately reflect the application running in the system?
- Does it consider the behaviour of hardware peripherals interacting with the processor?
- Does it contain the interrupts cause by external agents?

If the trace reflects accurately the application running in the system in a worst-case situation, the cache configuration can be explored in order to derive an accurate miss rate figure. If the model considers all the stall events at the correct point in time we can model the performance as function of time.

However, if the answer is no and the trace does not represent accurately the application running in the processor we must clearly point to what must be investigated further and what are the points of inaccuracy. Is possible to estimate with a certain degree of accuracy the results produced by this uncertainty? If yes we can give a fairly accurate number for the miss rate. The cache behaviour as function of time is more difficult to assess because there are many more factors to consider. For example the interaction with peripherals, interruption of execution, consumption of data produced and real time execution.

3.6 Cache Behaviour

The execution time of a task depends on number of processor stall-cycles, which depends on the miss rate and the miss penalty. The cache miss rate is dependent on the state of the cache after task switch (in case more than one task is running in the processor) and the input data values. The cache miss penalty depends on conflicts on memory port, which depends on misses of other caches. The execution time of a task depends on number of processor stall-cycles, which depends on the miss rate and the miss penalty. The cache miss rate is dependent on the state of the cache after task switch (in case more than one task is running in the processor) and the input data values. The cache miss penalty depends on conflicts on the next level memory port, which depends on misses of other clients (see *Figure 1*). In order to study real time execution other factors must be taken into account.

3.6.1 Supply and consumption of data samples

It is assumed that this system processes data streams where the input data is provided by an external source, which provides a new input sample with a certain periodicity. This source can be an A/D converter and similarly and after processing the data is consumed by a D/A converter into an analogue signal. This last situation is illustrated in *Figure 18* with the MP3 decoder serving as example.

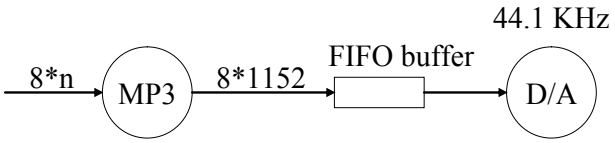


Figure 18 – Consumption of data by D/A task

These systems are often designed as hard real-time systems because input samples must be processed in time and output samples must be produced in time, since otherwise the input buffer overflows or the output buffer under-runs.

The real time execution is considered for the MP3 decoder in the following way.

The MP3 decoder:

- MPEG-1, Mono 44.1kHz, 64 kbit/s
- 8 calls to Mp3_Process
- Each call generates an output block of 1152 samples (16-bit, stereo)
- 576/44.1kHz = 13 ms per call, *8 = 104.5 ms real-time

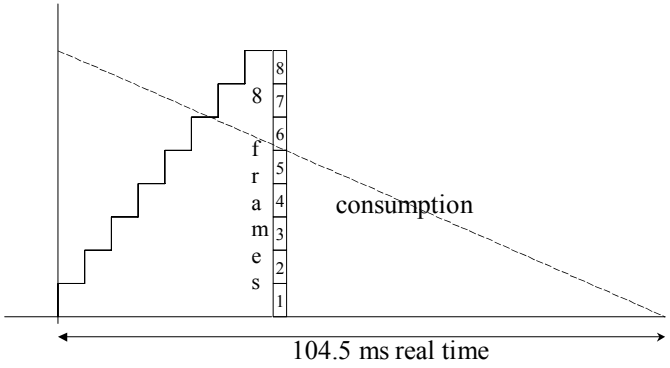


Figure 19 - Real time execution for MP3 decoder

The MP3 trace produces 8 blocks of 2×512 samples in a certain amount of execution time. Those samples have to be consumed from a buffer located in the memory. The D/A converter consumes one sample at a strict frequency from the output buffer, this is not considered in the trace and hence, the D/A task is not considered in the execution. This task may cause the interruption of the execution of the program running in the processor and may be another source of stalls in the execution. Furthermore, *Figure 19* shows that the processor will be inactive most of the time since execution time for producing the 8 blocks are likely to be much less than the 104.5 ms of

real time. We want to increase the load of the processor by running two or more tasks in parallel and consequently the scheduling of tasks be taken into account.

3.6.2 Execution time and Response time

The definition of actor is introduced. An actor is a task with well-defined properties. An actor has a worst-case execution time and in this time consumes a fixed number of data from every input and produces a fixed amount of data on every output. During every execution of the actor the same amount of tokens are consumed and produced.

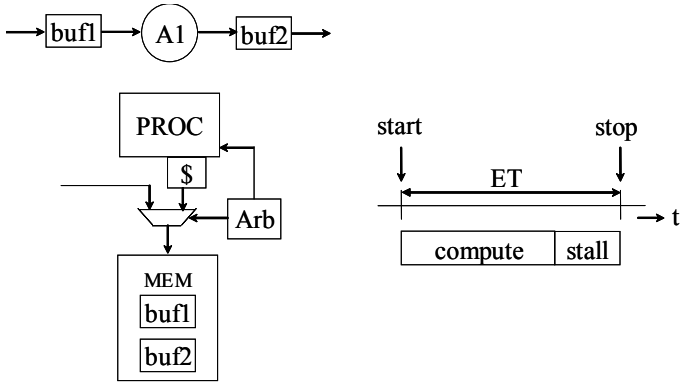


Figure 20 - Execution time of an actor

It is possible to assign more than one actor to a processor. Actors assigned to the same processor are executed successively. The notion of response time (RT) is defined for an actor, as described in *Figure 21*. The data-dependent response time of a task that is mapped onto a processor and receives input data is defined as the time it takes from when the task is enabled (i.e. when it has all necessary data to compute) until it is finished. The task may have to wait for the processor to get free before it can start and it can be interrupted several times during its response. This waiting time depends on which actors can be enabled at the same time. Not only the task's processing time, but also the waiting time and interruption time must include in the response time.

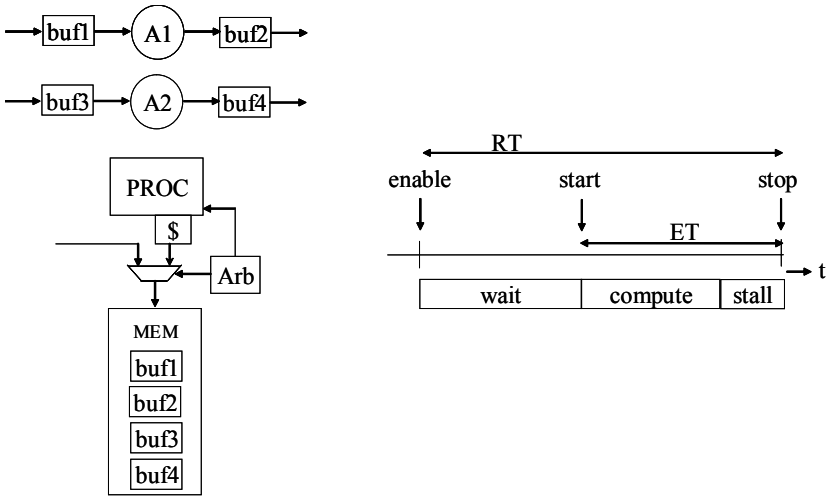


Figure 21 - Response time of an actor

3.6.3 Scheduling approach

The scheduling approach also plays an important role in the system analysis. Various methods have been developed to tackle the problem of making run-time choices. Examples are Round Robin and Time Division Multiple Access (TDMA). In Round Robin tasks are checked consecutively, if a task is enabled, it is executed immediately. TDMA uses a fixed period (time wheel) of which each task gets a predefined time slot (see *Figure 22*). During this time slot, a task has the opportunity to execute until the end of the slot. When the task is not enabled, because it has no data, the processor is idle for the remaining time in the slot. TDMA requires pre-emption to allow the execution of tasks to span over multiple time slots. The power of this method is that the execution of tasks (or jobs) can be decoupled, as if they were running on separate processors: if two tasks do not depend on each other's data they cannot influence each other's timing. Pre-emption, however, needs additional hardware and introduces task-switching overhead.

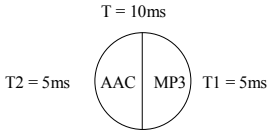
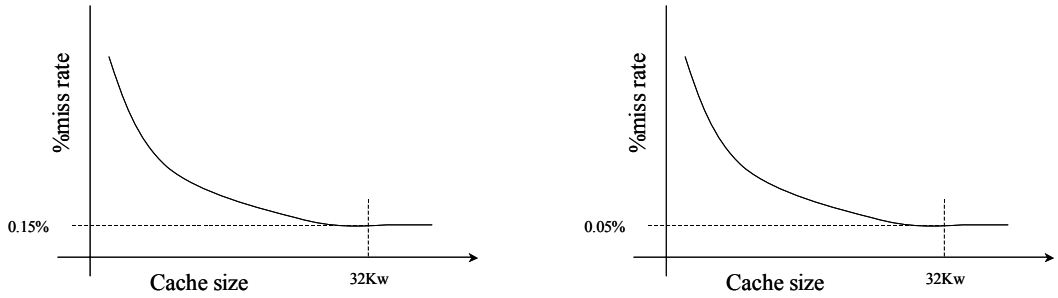


Figure 22 – Time wheel TDMA

3.7 Results, Conclusions and Proposals

3.7.1 Cache architecture

The Figures show the saturation point of the miss rate as function of the cache size for the two applications. The value of the miss rate shown in the graph refers to the input stream data, which was estimated in section 3.1.



The increase of the line size improves the miss rate figure; this is shown in appendix A.

A reference configuration for the cache was derived from the trade-off analysis between performance, power consumption and cost. This is out of the scope of this thesis.

The cache reference configuration is:

- Data cache – double ported memory, 8Kw size, 4-way set associative and 8 words for the cache line size.

- Instruction cache – one ported memory, size of 8Kw, 2-way set associative and 8 words for line size.

3.7.2 Miss rate

The value of the miss rate is accurate. The value refers for a specific granularity; there are two uncertainties that can have an impact in the value:

- I/O stream data (estimated).
- Interrupt code (the traces derived from a simulation were data input samples are available in memory)

The MP3 decoder for a granularity of 8 frames and for the reference architecture configuration has a value of 1.5%.

The AAC decoder for a granularity of 6 frames and for the reference architecture configuration has a miss rate of 2.3%.

The values shown above include the impact of I/O stream data calculated in section 3.1. The values have little impact in the miss rate performance.

3.7.3 Processor Stall percentage

The processor stall percentage for the reference cache configuration and for the MP3 decoder is $1.5\% \times 3.5 = 5.25\%$

The processor stall percentage for the reference configuration and for the AAC decoder is $2.3\% \times 3.5 = 8.05\%$

The processor stall percentage value is restricted by the assumption that the latency is fixed.

3.7.4 Dynamic behaviour

The variation of the miss rate values for process execution in both algorithms is derived from the simulation described in Section 3.1.3. The worst-case behaviour of the application is dependent on the input data and sources of dynamism present in the algorithm (e.g. condition statements, data-dependent iterations). The highest and lowest value per process computation is shown below

- MP3 decoder – highest value 2.08%, minimum value 0.89%
- AAC decoder – highest value 2.60%, minimum value 1.99%

3.7.5 Task-switching Execution

For a task-switching simulation of the MP3 trace and AAC trace, where each task as a budget of 1 million cycles of execution time, the miss rate is 1.91%. For a task budget of 100Kcycles the miss rate increases to 3.5%

3.7.6 Trace-driven simulation

Evaluating the cache architecture through trace-driven simulation seems to be a good solution; it allows exploring wide range of cache configurations and memory models. Using trace-driven simulation, it was found that there are certain properties that must be known in order to improve the analysis (eg. input-invalidate/output-flush/IO-handling/data memory map). These items will be included in the applications when it is adapted to run in a cache based DSP. An extensive trace application analysis helped this process and it allowed estimating uncertainty that exists in the trace regarding I/O data stream. In this system context, the miss penalty is dependent in other cache miss rate and it is assumed fixed and hence, the focus was on the miss rate.

Difficult to evaluate real time execution:

- production and consumption of I/O samples by strict periodic sources not considered ⇒ I/O by interrupt .

3.7.7 Recommendation

A possible recommendation for cache operation configuration optimisation:

- For MP3, lock in one of the cache ways data referent with the static data
- For AAC, lock in one of the cache ways data referent with the data ROM

This is possible in the profiler by configuring a specific address range (i.e. MMU model inside the cache) to be locked in some way of the cache.

3.7.8 Proposal

Based on the findings, a decision was made in order to move to ISS processor simulation. The goals are stated in Chapter 1.

3.8. References of Chapter 3

[1] "*MP3 Decoder Package User Manual*", Dietmar Gradl, Dietmar Lorenz, Cliff Parris, 11-May-2004.

[2] "*AAC Decoder Package User Manual*", Alex Stenger, 18-Mar-2004.

[3] "*Software Creation Process Annex D26_1 - R.E.A.L. Assembler Coding Standards*", Peter Meyer, Arthur Tritthart, 19-Aug-2004

Chapter 4

Instruction Set Simulator

Simulation is recreating an environment in enough detail that desired effects of a real system could be observed. Instruction-Set Simulation is simulating a processor at the instruction-set level. Instruction-set simulation is a type of simulation that is detailed enough to run executable programs intended for the machine being simulated. In this simulation environment it is possible to extend the scope of the study to a multiprocessor context. The applications are under our control and can be adapted in such a way that the applications can be simulated as the assumption stated in Section 3.6.1.

This chapter introduces the simulation environment used for processor simulation of the Adelante RD 1602X family of DSPs. The environment is targeted at embedded applications that require data processing, for instance, telecom and medium to high-end audio applications. The simulation produces simulation results that are *bit-true* and *cycle-true* accurate. Cycle-true means that values are produced at corresponding times and bit-true means that a value produced in simulation is bit-for-bit identical to the corresponding value produced in hardware.

The chapter is organized as follows. The first Section gives an overall description of the most significant features of the development environment, how to develop, debug and simulate an application in the environment. Section 4.2 describes the hardware resources applicable in the experiments described in later chapters.

4.1 Integrated Development Environment

4.1.1 Basic Design Flow

Figure 23 illustrates the typical design flow to create an executable that is developed for the Saturn core.

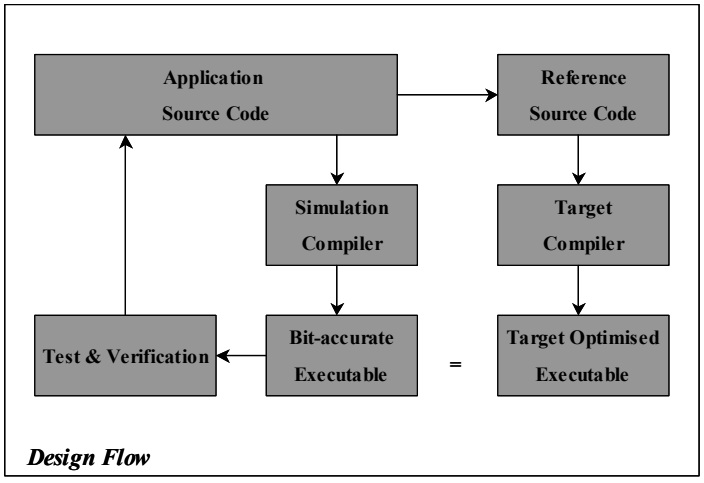


Figure 23 – Design flow

The environment enables to test and verify the DSP application. The Simulation C-Compiler converts the DSP-C code into ANSI C source code and produces a bit-accurate executable using a host compiler. Running this executable on a PC emulates the behavior of the DSP code on a Saturn core, enabling to functionally test and verify the code. The final version of the application source serves as the reference source for the target C-compiler. The target C-compiler converts the reference source into a Saturn optimized executable.

4.1.2 Development flow

The development flow is show in the *Figure 24* bellow.

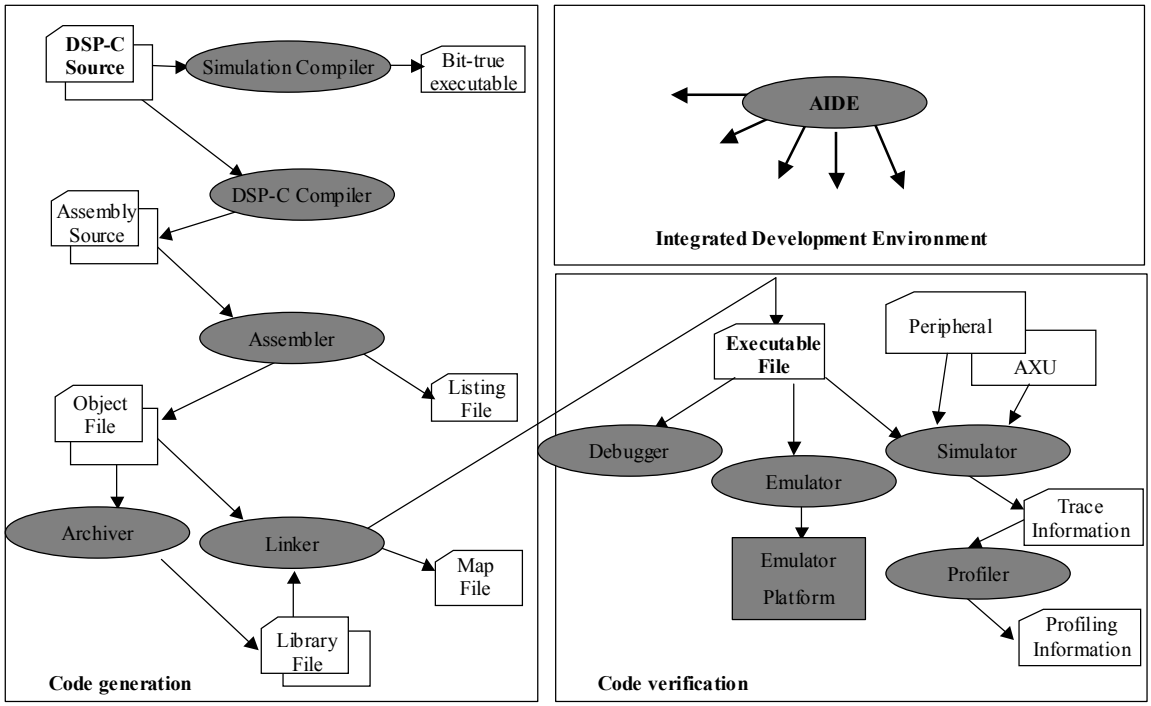


Figure 24 –Development flow

The Integrated Development Environment (Atmosphere IDE) developed at the DSP-Innovation Centre of Philips Semiconductors is particularly suited for the Saturn DSP-core. The Atmosphere IDE is part of the Saturn Atmosphere tool chain. Atmosphere IDE is a graphical user interface integrating all Saturn tools: C compiler, assembler, linker, simulator and the emulator. It has a Graphics User Interface (GUI) that consists of a number of windows, which is divided in the category of general windows and the core specific debug windows. The IDE is a GUI front-end that can interface with multiple simulator engines, also referred to as core DLLs.

The Atmosphere IDE enables build, develop, debug and simulate DSP applications. Applications are project-based and can be comprised of C and/or ASM source files. DSP cores can be loaded as simulator engines or core-DLLs into the IDE, allowing for multi-core simulation. Peripherals can be loaded as peripheral DLLs, enabling simulation of a DSP sub-system in any possible configuration. Atmosphere IDE is specifically designed for the Windows operating system. It takes advantage of the Windows multitasking capabilities, to continue working on other applications while running a simulation session in the Atmosphere IDE.

The simulator loads and runs C/ASM translated executables on the simulated DSP cores. While debugging, various debug windows enable comprehensively analyse and troubleshoot the application. To simulate and debug an application, a number of powerful integrated functions are at disposal. Several variants of memory and register breakpoints can be set to halt program execution for evaluation purposes. The Atmosphere IDE supports profiling, which in order to optimise and detect performance, bottlenecks in the source code.

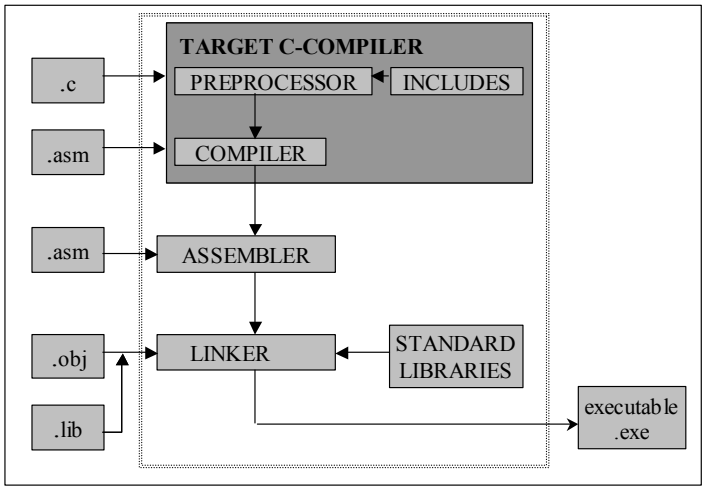


Figure 25 – Simulation compiler

The design flow to generate an executable designed for the Saturn core is shown in *Figure 25*. Normally an executable file *(.exe)* is created based on C- and/or assembly coded files, after it is compiled, assembled and linked by the respective tools.

4.1.3 Summary of IDE main features

The Atmosphere IDE debugging features are broad. It allows source code level, assembly code level and clock cycle level debugging. Event level and asynchronous clock cycle level debugging in a multi-core environment. During simulation, it is the possibility of stepping into, over and out of functions, subroutines and macros. The values of variables can be examined and changed. The memory and register output can be redirected on reading and/or writing and setting read and/or write protection to memory cells. It supports the memory paging mechanism applicable to Saturn DSP-cores. It is possible comprehensive troubleshooting during assembling, compilation and building of applications by relating error and warning messages to their corresponding source code line. A summary of the IDE features is given:

- Integrated Development Environment (IDE) for Saturn DSP cores, integrating a C compiler, an assembler, a linker, a simulator, an emulator and a profiler in a project-based environment.
- Usage of configuration files to consistently describe a DSP-core's memory configuration.
- Multi-DSP core simulation and inter-core communication.
- Program and register breakpoints in several variants, including predefined conditions and actions.

- Instructions, operations and cycles timing counters.
- Importing and exporting of instructions and data, in various formats, from and to external files.
- Special support for expanding and stepping in assembly macros.
- Profiling to support the optimisation of source code.
- Multiple formats for displaying memory and register values.
- Application Program Interface (API) supporting the connection of peripheral user-DLLs.

4.2 Supported Hardware Resources

4.2.1 DSP subsystem

The Saturn has a double Harvard architecture: the DSP has two data memory spaces (X, Y), and one program memory space (P), as shown in *Figure 26*. These three memory spaces are completely independent. The DSP can perform three different tasks:

- Control the program flow.
- Calculate data-memory addresses and access data memory.
- Execute the operations on the actual data.

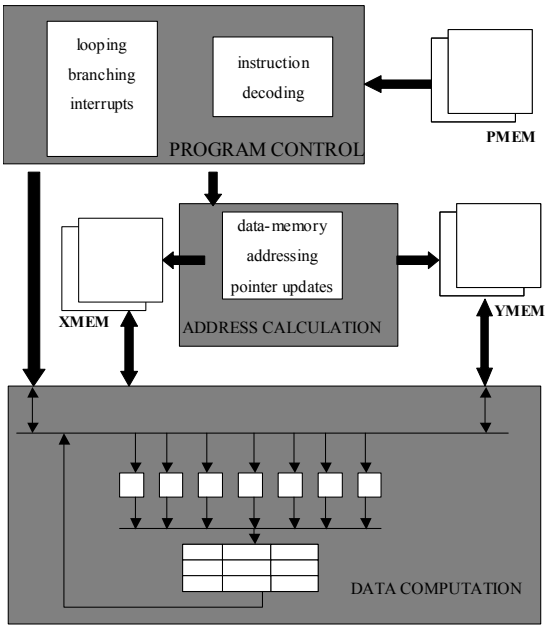


Figure 26 – Saturn DSP block Diagram

The Program Control Unit (PCU) takes care of instruction decoding and of the instructions that can influence the program flow. These are instructions like branching, looping, and interrupt

handling. The Data Computation Unit (DCU) contains the actual data path. In this unit the logic and arithmetic operations are executed. The Address Calculation Units (XACU, YACU) handle the addressing of the X and Y data memories. The Saturn DSP subsystem, illustrated in *Figure 27*, also contains a DMAC (DMA Controller) and an MPU (Memory Paging Unit). Configuring these blocks should be done via the configuration file.

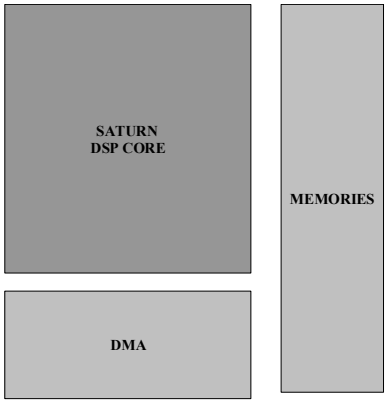


Figure 27 – DSP Subsytem

The simulator provides configurable memory simulation. By default, the simulator does not provide any memory mapping specific to the device or processor. Adding memory blocks using the simulator configuration file can simulate all the memory blocks in Program, Data, and I/O space. For example, for on-chip memories, the simulator takes care of the number of cycles required to access memory depending on wait states specified by the simulator configuration file.

4.2.2 Peripherals

Usually, the DSP is attached, through the I/O-bus, to one or more hardware peripheral blocks. A number of relevant hardware peripherals are also available as software peripheral. The DSP code can cooperate with one or more peripheral blocks, enabling simulation and debug of the application. The peripheral blocks are implemented in software by dynamic link libraries (DLL files) and can be added to the simulator. A DLL is a small, executable program that, when needed, is called on by a running application and loaded to perform a specific function, such as transfer data to a file. DLL files are dynamically linked with the program that uses them. Dynamic links help the programs to use resources, such as peripherals. DLLs are used by applications to store functions and data required for proper operation. Adding a peripheral software module to the simulator is best done via the configuration file. The most efficient way to run a simulation is to make use of command files.

The peripherals that are used by the simulator and that are relevant for this thesis are:

- The Direct Memory Access Controller (DMAC) allows (for) fast data transfer between memory and a peripheral without the intervention of the DSP core. For the data transfer one of the available DMA channels can be used.

- Data cache (Dcache) is used to reduce the effective memory access time. It is a 4-way set-associative dual-ported 4 KW SRAM, it has a cache line width of 8 words (128 bits). Uses the PLRU cache line replacement algorithm. The Dcache features an integrated MMU, the MMU operates in the resulting address space of the paging done by the MPU. The MMU allows partitioning of the main data memory into a maximum of 8 segments.

For each segment an individual set of attributes can be programmed. The Dcache supports sharing of X and Y memory segments. The Dcache can be controlled through 8 cache management instructions (CMI). They can be issued from the DSP core or from an external DMA host. The Dcache supports cache line locking and cache way exclusion. The Dcache supports two programmable write-policies: write-back and write-through mode.

- Instruction cache (Icache) is used to reduce the effective memory access time. The Icache is a 2-way set-associative single-ported 4 KW SRAM. The Icache has a cache line width of 8 words (128 bits). The Icache uses the PLRU cache line replacement algorithm. The Icache features an integrated MMU. The MMU allows partitioning of the L2 program memory into a maximum of 8 segments. For each segment an individual set of attributes can be programmed. The Icache supports hardware prefetching from the L2 program memory. Hardware prefetching caches the requested cache line, and simultaneously stores 'near' cache lines in a prefetch buffer. Prefetching can be used to reduce the percentage of cache misses. The Icache can be controlled through 4 cache management instructions (CMI). They can be issued from the DSP core or from an external DMA host. The Icache supports cache line locking and cache way exclusion.

- The Performance Profiling Unit (PPU) provides data that can be used by a programmer to gather information on the real-time behaviour of a cached application. The information can be used to enhance cache behaviour and speed up execution times, by improving the DSP program structure.

- The Memory Paging Unit (MPU) – Paging is done by MPU unit of the DSP core itself. It is used to increase the memory space available by using a paging mechanism.

- The Test Input Output (TIO) enables communication between the DSP's I/O space and an external host.

4.3 Profiling

The environment can be used to measure the runtime performance of application code. For this purpose, the profiler generates profiling data, based on an executable file *(.x)* and at least one binary trace file *(.bin)*.

On instruction level the generated profiling data indicates, for each instruction:

- The number of times an instruction is executed.

- The average number of clock cycles consumed by an instruction.

- The memory start address at which an instruction is stored. If paging is used, also the page number is indicated.

- For an instruction containing a function call, also the number of clock cycles consumed by the call is indicated.

On function level, the profiler accumulates the instruction level data. The profiling data generated on function level, indicates, for each function:

- The total number of clock cycles consumed by a function (both in- and excluding calls).
- The relative number of clock cycles consumed, as a percentage, of the total number of clock cycles consumed by the program (both in- and excluding calls).
- The memory address range at which a function is stored. If paging is used, also the page number is indicated.
- The number of times a function has stalled.
- The average number of clock cycles wasted during each stall.
- A description describing the reason or cause of a stall
- The number of times a function is called (both in- and excluding calls).
- The minimum number of clock cycles consumed per function call (if a function is called multiple times; including calls).
- The maximum number of clock cycles consumed per function call (if a function is called multiple times; including calls).
- The average number of cycles consumed per function call (both in- and excluding calls).
- The percentage of function lines that were executed at least once, during a program run.
- The total size of a function, indicated as a number of (16-bit) words.

The Saturn Atmosphere generates static and/or dynamic profiling data accordingly. All profiling data is represented in tabular form. The profiling tables generated are listed below.

- *File information table* - lists general information about the profiling process, such as the used input files and the generated output files and the version of the profiler.
- *Program information table* - static and dynamic profiling information on program level. The table lists the name of the executable, the number of program executions, the average number of clock cycles consumed by the program, the percentage of the program lines that are executed during a program run and the size of the program as a number of 16-bit words (*size*).
- *Static function information table* - lists for each function contained by the program, the function name, the size as a number of 16-bit words and the start and end address of the function in memory.
- *Function information table (excluding callees and interrupts)* - dynamic profiling information on function level. The table lists for each function, excluding callees, the

function name, the size of the function as a number of 16-bit words, the percentage of the function lines that are executed during a program run, the number of times the function is called, the average number of clock cycles consumed per call, the total number of clock cycles consumed per call, the relative number of clock cycles and the start and end address of the function in memory.

- *Timing information table (calls, IRQs and traps (in/excluding interrupts))*1 – dynamic profiling information on function level. The table lists for each function, including callees, the function name, the average number of clock cycles consumed per call, the relative number of clock cycles, the total number of times the function is called, the minimum and maximum number of clock cycles consumed per call.

- *Assembly instruction information table* - dynamic profiling information on function and instruction level. The table lists for each function, the name of an instruction label, the name of a function, the assembly instruction(s) contained by the program/function, the number of times an instruction or function is executed, the average number of clock cycles consumed by an instruction, the number of times a function has stalled, the average number of clock cycles lost during each stall, the stall reason, the memory address range occupied by the function and the memory start address of the memory occupied by the instruction.

4.4. References of Chapter 4

User and Reference Manuals:

- Saturn Atmosphere Getting Started Manual
- Saturn RD16024 Programmer's and Reference Manual
- RD 16025 Technical Reference Manual
- Saturn Atmosphere IDE Manual
- Saturn Atmosphere Dynamic Peripheral Manual
- Saturn Atmosphere Peripherals API Manual
- Saturn Atmosphere profiler API Manual

Chapter 5

Processor Architecture, Model of Computation, communication & Scheduling

The communication between two processes running in different processors is examined. A Synchronous Data Flow is introduced in which computation, communication and run-time arbitration is expressed. It is possible to transform SDF graph in a HSDF graph. The most important properties of an HSDF graph are shown and an example illustrates how to derive the minimal throughput of the system.

The chapter is organized as follows. First the processor template will be introduced. In Section 5.2 the terminology is explained and the communication between processes running in different processors. Section 5.3 introduces the formal model.

5.1 The Processor Template

The processor template that is available in the DSP-Innovation Centre of Philips Semiconductors is shown in *Figure 28*.

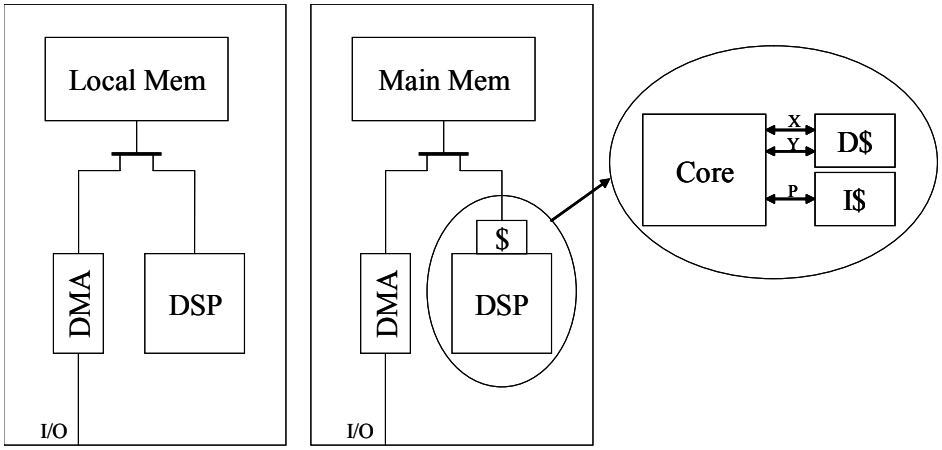


Figure 28 – Processor template

This template is used for a homogeneous multiprocessor system and represents a tile. The cache can be added as an application specific execution unit (AXU DLL) and in this case the access to the memory incurs in extra cycles. There is an arbiter to access the memory that gives priority to the DMA. In the tile there is a direct memory access controller (DMAC) capable of transferring data between the memory and a peripheral connected to the tile.

5.2 Actors and communication

Tasks with well-defined properties are called actors. One of the properties is that an actor can't be blocked internally and thus has a worst-case execution time. In this time, it consumes a fixed number of tokens from every input and produces a fixed number of tokens on every output. A token is defined as a container in which a fixed amount of data can be stored. During the execution of the actor, the fixed amount of tokens are consumed and produced.

Two processes can only communicate with each other if a connection exists between them. The connection between two processes is done via a FIFO buffer, either located in the data memory or added as a peripheral. When two processes are on different processors and the communication is done via a shared memory, the FIFO is implemented in the data memory. Optionally, the communication is done via a DMA channel connecting a peripheral FIFO to two processors.

5.2.1 Communication via shared memory

In a multiprocessor system where the communication between actors is done via shared memory the multiprocessor architecture in the simulation environment corresponds to *Figure 29*.

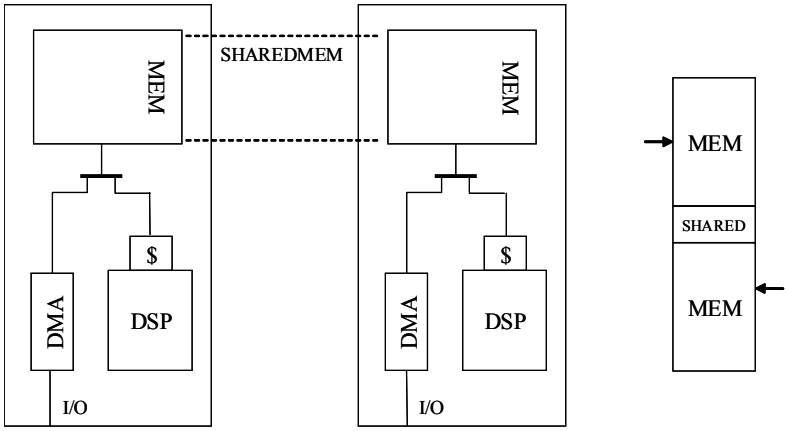


Figure 29 – Communication via shared memory

Each processor has is own port to access the main memory. The shared memory location is where the communication between actors takes place. The shared location may be defined with different memory locations for each of the processors.

5.2.2 Communication via DMA

In a multiprocessor system where the communication between actors is done via FIFO buffer the multiprocessor architecture in the simulation environment corresponds to *Figure 30*.

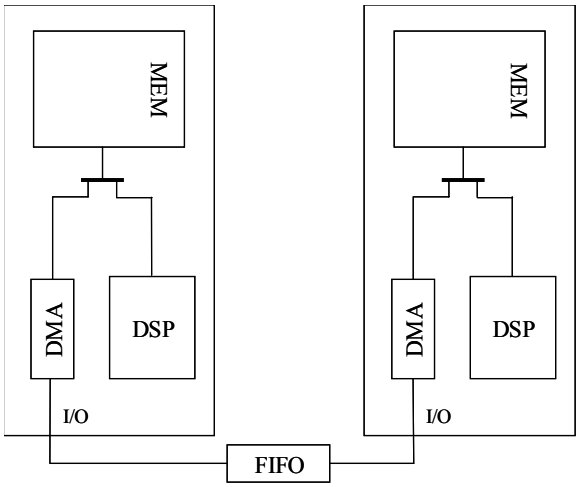


Figure 30 – Communication via DMA

This optional architecture approach uses a FIFO buffer to connect the two cores using a DMA channel.

5.2.3 C-heap protocol

The FIFO buffers needed for the communication between the actors are implemented in software using the C-heap protocol. *Figure 31* exemplifies how the protocol works.

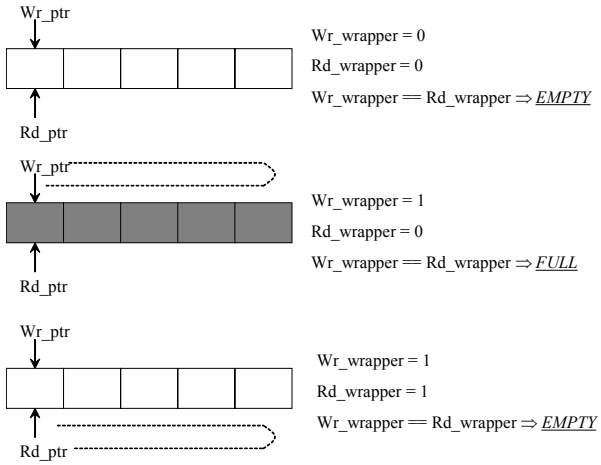


Figure 31 – C-heap protocol example

The FIFO is implemented as a circular buffer, when a pointer reaches the last position it is reset and points to the first position. The variables wrappers are used to verify if the buffer is full or empty. A comparison is performed between the pointers in order to verify if there is space or data available in the buffer, depending if the action is to write or read.

5.3 Model of computation, Communication and Scheduling

This section is based on the work described in [1][2][4]

5.3.1 Synchronous Dataflow

The SDF graphs are used in this thesis to derive the worst-case temporal behaviour. An example of an arbitrary SDF graph is depicted in *Figure 32*. The nodes in an SDF graph are called actors. Actors have well-defined input/output behaviour because they produce and consume a number of tokens. A token is depicted in the figure as a black dot. If more than one token is present on an edge then the number of tokens is specified next to the dot. An actor is enabled after a predefined number of tokens are available on every input of the actor. The number of required tokens is specified at the head of the data edge of every incoming edge of the actor. The number at the tail of an edge denotes the number of tokens an actor produces. Actors with internal state are modelled in an SDF with a self-edge, like the self-edge of actor A4 in *Figure 32*. The self-edge has one initial token such that the next execution cannot start before the previous execution of the actor is finished. The Response Time (RT) of an actor is the time interval between the time that the actor is enabled and the time that the actor finishes its execution. The response time of an actor can depend on the values in the token that are consumed by the actor.

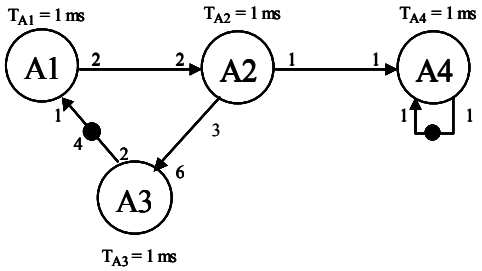


Figure 32 - SDF graph

5.3.2 Homogeneous Synchronous Dataflow and monotonic execution

An SDF graph can be transformed into a Homogeneous Synchronous Data Flow (HSDF) graph on which analysis is performed. An algorithm, which transforms any SDF graph into an HSDF graph, is described in [3]. The HSDF graph obtained after transformation of the SDF in *Figure 32* is shown in the *Figure 33*.

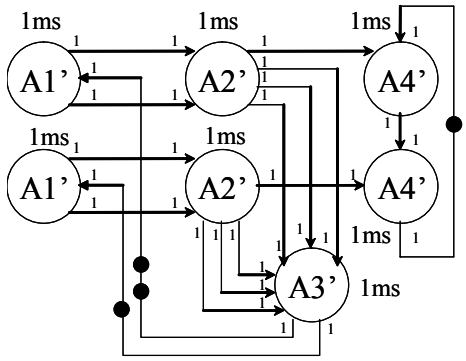


Figure 33 – HSDF graph obtained after transformation of the SDF in Figure 32

An HSDF graph is a special case of an SDF graph in which the execution of an actor results in the consumption of one token from every incoming edge of the actor and the production of one token on every outgoing edge of the actor. After it is started, an actor in the implementation finishes its execution within the WCET.

An HSDF graph can be executed in a self-timed manner, which is defined as a sequence of firings of HSDF actors in which the actors start as soon as they are enabled. In the case that the HSDF graph is a strongly connected graph and a FIFO ordering is maintained for the tokens, then the self-timed execution of the HSDF graph has some important properties. A FIFO ordering is maintained if the next execution of the same actor cannot start before the previous execution is finished.

The properties of the self-timed execution of HSDF graphs are:

- The HSDF is deadlock-free if there is on every cycle of the graph at least one initial token.

- The execution of the HSDF graph is monotonic, decreasing actor execution times result in non-increasing actor start times.

- A HSDF graph will always enter a periodic regime. More precisely, there exist a $K \in$ ˢ₂, an $N \in$ ˢ₂ and a $\lambda \in$ ᴾ₂, such that for all actors $v \in V$, given $k > K$ the start time $s(v, k + N)$ of actor v in iteration $k + N$ is described by:

$$s(v, k + N) = s(v, k) + \lambda.N \tag{5.1}$$

Equation 5.1 states that the execution enters a periodic regime after K executions of an actor in the HSDF graph and the time one period spans is λ.N. The number of firings of an actor v in one period is denoted by N. Thus, λ is equal to the inverse of the average throughput measured over period. Hence, a periodic regime will be entered and consequently a simulation can be stopped after the first period of the periodic regime.

The Maximum Cycle Mean (MCM) [3] of an HSDF, which is equal to λ, is given by equation:

$$MCM(G) = \max_{c \in C_G} CM(c) \tag{5.2}$$

The Cycle Mean (CM) of a simple cycle c in the HSDF graph G is given by equation:

$$CM(c) = \sum WCET(v) / d(c) \tag{5.3}$$

In this equation *d(c)* denotes the number of initial tokens on the edges in a cycle *c*. The WCET of actor *v* is denoted by WCET(*v*). The MCM of an HSDF graph can be derived with a pseudo polynomial algorithm [5][6].

The worst-case start-times of the actors during the transition state as well as the steady state can be derived by simulation. During this simulation, all actors must have an execution time equal to their worst-case execution time. The start-times observed during this simulation are equal to the worst-case start times of the actor due to the monotonicity of the HSDF.

Since the HSDF graph is monotonic, decreasing actor execution times result in non-increasing actor start times. The reason is that a shorter response time of an actor results in an earlier production of tokens, and an earlier production of tokens results in an earlier arrival of tokens. An earlier arrival of tokens cannot result in a later enabling and start of an actor. Due to monotonicity tokens will arrive later during self-timed execution of the SDF graph than in the implementation, given that the response time of an actor in the implementation is smaller or equal than the response time of the actor in the SDF graph.

5.3.3 Timing analysis example

In [1] the temporal behaviour of a system is derived by constructing an SDF graph of the application that not only models the computation and the communication, but also the affects of run-time arbitration. The example given in [1] illustrates the analysis techniques that are used to derive the temporal behaviour of a system. The example uses a simple multiprocessor architecture (see *Figure 35*) and the streaming application is represented by a HDSF graph shown in *Figure 34*.

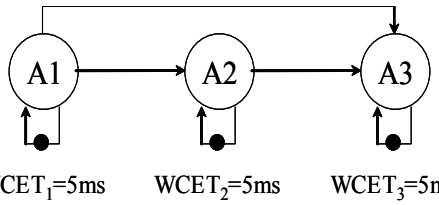


Figure 34 – streaming application described as an HSDF graph

The streaming application is executed in the multiprocessor architecture comprised of two processors and each one has an instruction memory. The bus arbiter grants access to the data memory and the execution time of the actor takes into account the worst case time to access the data memory.

.

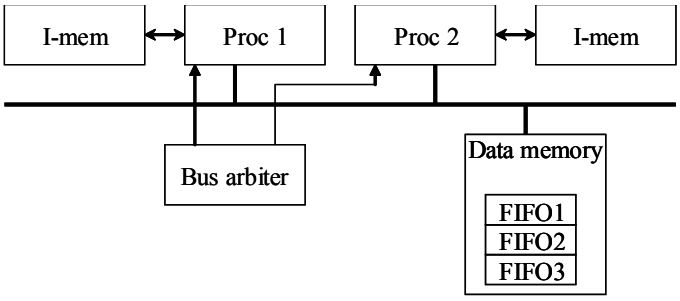


Figure 35 – multiprocessor system with centralized data memory

Assume as example that actors A1 and A3 are executed on processor 1 and actor A2 on processor 2. The actors that are executed in the same processor are invoked in an infinite while loop. When the actor is invoked, it checks if there are tokens available in each of its inputs and if there is space available in each of the outputs, otherwise returns. The Round-Robin arbitration of the actors A1 and A3 is modelled in the Worst-case response time (WCRT) of the actors. The WCRT of an actor is the maximum interval between that the actor is enabled and that the actor finishes its execution. Given Round-Robin arbitration, the WCRT of actor A1 is the sum of the WCET of A1 and A3 actors (the same is for A3). The implementation-aware HSDF given RR arbitration for actor A1 is shown in *Figure 36*.

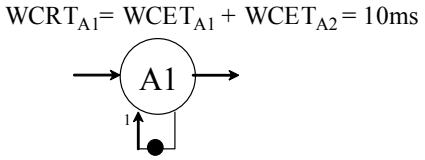


Figure 36 – Implementation-aware HSDF graph of A1 given RR arbitration

In the example it is assumed that at most two tokens can be stored in the FIFO (represented with two dots). The minimal throughput of the system is equal to the throughput during self-timed execution of the HSDF graph in *Figure 37*, given that all actors have a response equal to their WCRT. The MCM of the HSDF graph in *Figure 37* is determined by the WCRTs of the actors on the cycle A1, A2, A3, and the number of tokens on the edges of this cycle. Hence the MCM of this HSDF graph is (10 + 5 + 10)/2 = 12.5ms, and the minimum throughput 1/12.5ms=80 tokens/s.

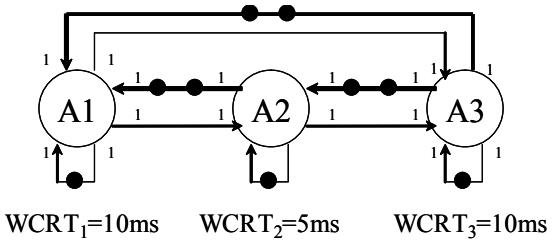


Figure 37 – implementation-aware HSDF graph

The minimum throughput can also be verified by self-timed execution of the HSDF graph. All the state changes of the HSDF graph is depicted in *Figure 38*. The figure shows that the first time

two states are identical is at $t = 50$ms. The state $t = 50$ms is equal to the state $t = 25$ms. The period between those two states is 50-25 = 25 ms and is equal to λ.N. It is shown that N=2 (The number of firings of an actor v in one period) so λ (MCM) is equal to 12.5ms.

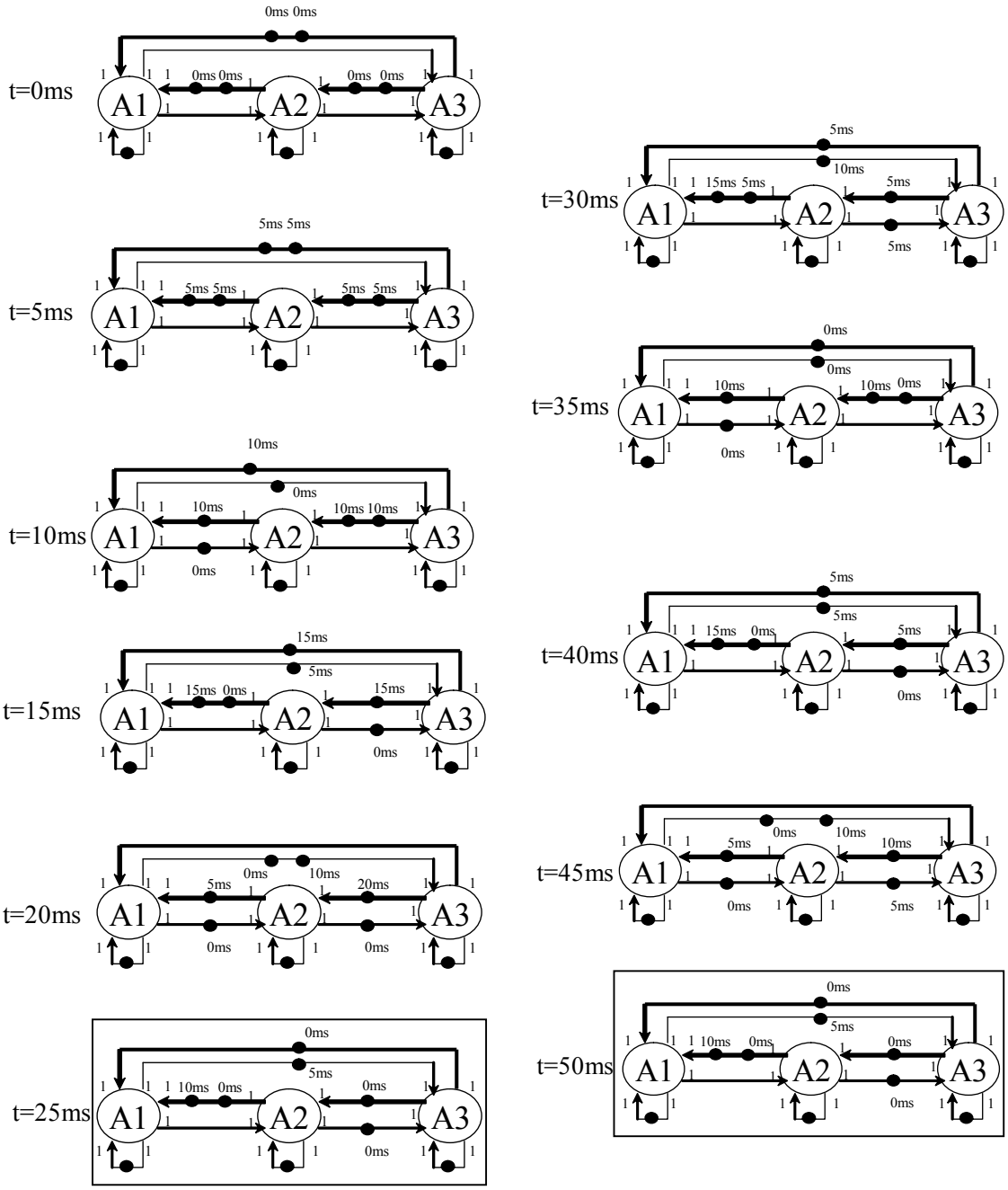


Figure 38 – The state changes of an HSDF graph from the example

5.3.4 Interaction with the environment

The MCM of the HSDF graph in *Figure 37* equals the inverse of the guaranteed minimal throughput of the system. This throughput is obtained without taking into account interaction with the environment. For systems that do interact with the environment it should be guaranteed that no data is lost due to overflow or underflow of the buffers between the system and the environment. A typical example of such a system is a system in which the input data is provided by a strict period external source like an A/D converter and consumed by a strict period external sink like a D/A converter. In [2] it is proven for this example that if the source and sink actor execute strictly periodically during self-time execution of the HSDF graph then the input buffer of the system will not overflow and the output buffer of the system will not underflow. In order to verify whether the FIFOs at the input and output of the system do not overflow or underflow it is needed that the strict period source and sink are modelled as actors in the HSDF graph as is shown in *Figure 39*.

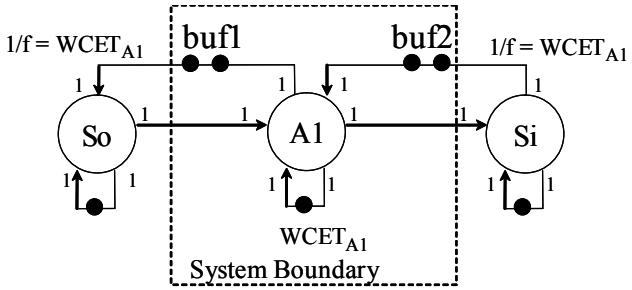


Figure 39 – HSDF graph that is used to prove that, given a strict periodic source and sink, the FIFO buffers at the input and at the output have sufficient capacity.

The source and sink actors are given a $WCET_{A1}$ equal to the length of period of the A/D's and D/A's strict periodic clock. The self-edge with one initial token ensures that the next execution of the source and sink actor cannot start before the previous execution is finished. If the source and sink actor execute strict periodic it is guaranteed that, in an implementation of the system, the FIFO between source and the system never overflows and the FIFO between the system and the sink never underflows. The reason is that, due to monotonicity, tokens will not arrive and depart later in the implementation than during a simulation run in which all actors have an execution time equal to their worst-case execution time. If tokens do not depart later than during the simulation run, then this results in the same or less tokens in the FIFO between the source and the system. If tokens do not arrive later than during the simulation run, then a greater or equal number of tokens are in the FIFO between the system and the sink in the implementation.

5.3.5 Other arbitrations

In [1] it is shown that an implementation-aware HSDF can be constructed for other arbitration policies. An arbitration policy is predictable if the maximum interval can be defined between the moment in time that an actor is enabled and the moment in time that the actor is invoked by the arbiter (*Figure 21*). Examples of other arbitration policies are TDMA and RMS if a period T is invoked by a timer in the system. After invocation the actor polls if there are sufficient tokens in its input FIFO before it executes. The actors can be considered independent because they do not block but return if sufficient tokens are available. The value of T_{wait} is equal to the period T of the timer because the maximum time between the arrival of a token such that the actor is enabled and

the invocation of the actor is equal to period *T*. The interval T_{proc} for TDMA is equal to the WCET of the actor in the case that the WCET of the actor is smaller than *T*.

The implementation-aware HSDF graph for an actor that is executed on a processor with TDMA arbitration or RMS is shown in *Figure 40*.

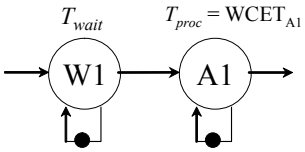


Figure 40 – implementation-aware HSDF if TDMA or RMS is applied

In [1] an example is given. *Figure 41* shows the implementation-unaware SDF graph. Actors A1 and A2 produce two tokens per execution and actors A2 and A3 should be executed 2 and 4 times respectively. The multiprocessor architecture is shown in figure 8, actors A1 and A3 are mapped into processo1 and actor A2 is mapped to processor 2.

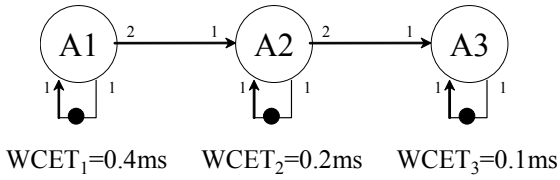


Figure 41 – implementation-unaware SDF graph

If RMS scheduling is applied, actor A1 is invoked with a period of 1ms, A3 with 0.25ms and A2. This results in the HSDF shown in *Figure 42*.

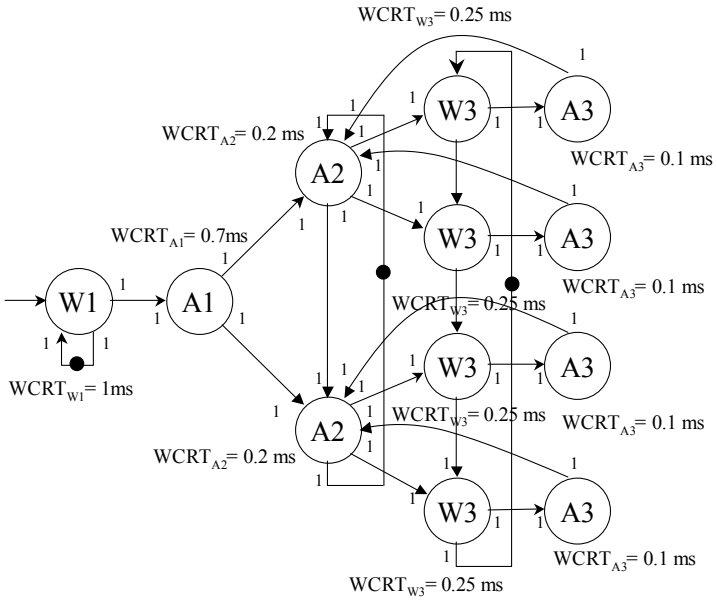


Figure 42 - HSDF graph obtained after transformation of the implementation-unaware SDF of Figure 41

This last figure serves to illustrate that the complexity in the analysis of the HSDF graph grows rapidly, a data flow program (SDF simulator) is required to compute the MCM (and hence the minimum throughput) for several FIFO buffer sizes.

5.4 Conclusions

The analysis techniques presented in this chapter showed that the minimal throughput of the system is derived with maximum cycle mean analysis (MCM). The minimal throughput of the system is equal to the throughput during self-timed execution of the HSDF, given that all actors in this HSDF graph have a response time equal to their WCRT. The analysis also showed that a strict periodic source (e.g. an A/D converter) and a sink periodic can be included as actors in the HSDF graph of the application.

In order to use this type of models, an upper bound must be derived on the execution time of an actor in the target processor. It is clear by the results obtained in chapter 3 (regarding the dynamism of the algorithms MP3 and AAC) that the execution time of an actor must be done in such a way that considers the dynamic behaviour.

The analysis of more complex graphs and schedules requires the use of a data flow simulator to compute the MCM (Throughput), FIFO capacity and simulate a Dataflow graph.

5.5 References of Chapter 5

[1] Marco Bekooij and Jef van Meerbergen, "*Timing analysis of data driven hard-RT multiprocessor systems*", Submitted to date 2006.

[2] Marco Bekooij, Orlando Moreira, Peter Poplavko, Bart Mesman, Milan Pastrnak, Jef L. van Meerbergen: *Predictable Embedded Multiprocessor System Design. SCOPES 2004: 77-91]*

[3] S. Sriram and S.S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, Marcel Dekker, Inc, 2000

[4] Marco Bekooij, Sonali Parmar, J. van Meerbergen, *Performance guarantees by simulation of process networks, SCOPES 2005*

[5] E.L. Lawler, *Combinatorial optimization: Networks and Matroids*, Holt, Reinhart, and Winston, New York, NY, USA, 1976.

[6] J. Cochet-Terrasson, G. Cohen, S. Gaubert, M. McGettrick, and J.-P. Quadrat, "Numerical computation of spectral elements in maxplus algebra", in *Proc. IFAC Conf. on Syst. Structure and Control*, 1998.

Chapter 6

Simulation of Streaming Applications in the ISS Simulator

This chapter describes the simulation of simple streaming applications mapped onto a single and multiprocessor architecture using the simulator environment available at the DSP-IC centre. The streaming applications are modelled with the graphs introduced in chapter 5 and the processor simulation environment introduced in chapter 4.

The goal is to try to investigate the predictability of the two multiprocessor architectures introduced in chapter 5 (*Figure 29* and *Figure 30*). In addition, to identify possible bottlenecks of the system and problems in the simulation environment that could introduce inaccuracy in the study. This chapter assumes a streaming infrastructure where the input samples are provided and consumed by external sources. The actors are simple computations; unfortunately there was no time to implement a similar set-up with real applications. The initial idea was to start with small examples and move towards a real case study. Still, the simple examples explained here can be related with case study presented in chapter 3.

The chapter is organized as follows. In Section 6.1, the inclusion of a strict periodic source in the simulation environment is described. In Section 6.2 a streaming application is simulated in a multiprocessor architecture. At the end conclusions are stated.

6.1 Single processor simulation

6.1.1 Streaming application with strict periodic source and sink

Figure 39 (Chapter 5) shows an HSDF graph that is used to prove that, given a strict periodic source and sink, the FIFO buffers at the input and at the output have sufficient capacity. In order to the FIFO buffers at the input and at the output of the system not overflow or underflow, the source and sink actors are given a frequency of 1/WCET of the actor mapped onto the processor. The FIFO capacity is two tokens; the expected timeline of the execution of the three actors in the HSDF is shown in *Figure 43*.

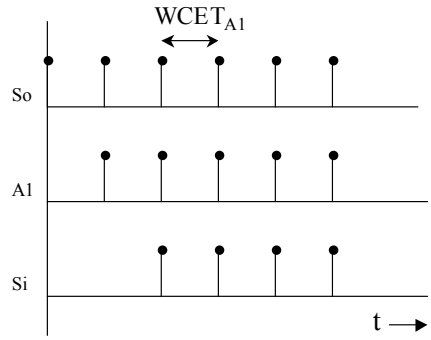


Figure 43 – timeline representing the execution of the HSDF graph of figure 5.12

It is easy to verify that the minimum throughput of the system is 1 token/$WCET_{A1}$.

6.1.2 ISS simulation of periodic source

For better understanding of this experiment it considers only the source actor. The analyses are analogous for both source and sink and hence, here it is explained an application that has a periodic source actor and an actor A1 executed in the DSP processor. The peripheral that supplies the samples is added in the simulation environment, the architecture is illustrated in *Figure 44*. The TIO (source actor) is connected to the DMA in one side and to a file in the other. The TIO supplies a sample at a programmable frequency; it requests the use of the DMA channel at that frequency and transfers a sample from the file to the FIFO buffer in the memory. When a predetermined number of samples (corresponding to a token) are transferred to the buffer an interrupt is produced. If the actor is executing at this point, the program will be interrupted and the processor will execute the interrupt routine.

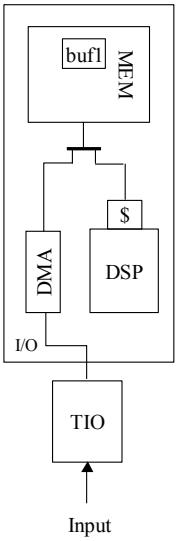


Figure 44 –Processor connected to peripheral TIO

The DMA is not autonomous, which means that the processor has to program the number of samples that are to be transferred to memory and the target location. The DMA has priority over the processor when accessing the memory, which means that the processor is stalled when a simultaneous access to the memory happens. The SDF graph can be represented as shown in *Figure 45* assuming that buffer1 has the space to hold 2 tokens. The actor DMA triggers the execution of A1.

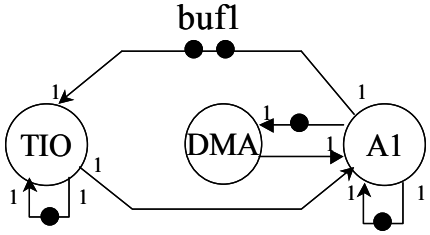


Figure 45 – SDF graph

The actors represented in the graph perform the following actions:

- TIO – reads one sample from the file (the input stream) at a pre-defined frequency. After it transfers a number of samples equivalent to a token through a DMA channel to the buffer in memory it generates an interrupt.

- DMA – it is invoked when the interrupt is activated (i.e., it is the interrupt routine). It updates the FIFO buffer write pointer, verifies if there is space available in the buffer. If there is space available reprograms the DMA, otherwise returns.

- A1 – The actor running in the DSP verifies if there is data available in the buffer, if yes executes and updates the read pointer, if not it returns. Further, it verifies if the DMA is active, if not calls the DMA routine.

The frequency of the TIO is programmable and consequently, it can happen that the TIO is programmed to read more than one token while the A1 actor is executing. The WCET has to be reviewed considering the number of samples that are transferred.

The Worst-case execution time of the actor A1 is

$$WCET_{A1} = Exec\_timeA1_{withouth\_stalls} + Stall_{Mem\_access} + Time_{program\_DMA} + N\_Tokens \times Stalls_{DMA\_access}$$

$$Stall_{Mem\_access} = Miss\ Rate \times Miss\ Penalty \times Exec\_timeA1_{withouth\_stalls}$$

$$Stalls_{DMA\_access} = number\_of\_samples \times 1clock\ cycle$$

In order to ensure that the FIFO buffer in memory does not underflow or overflow the TIO has to be programmed with the frequency equal to number_of_samples/$WCET_{A1}$. The time line illustrates the execution of the actors.

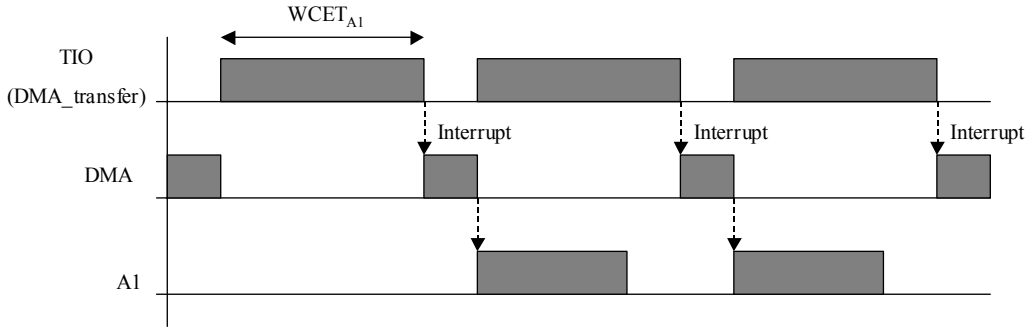


Figure 46 –Timeline execution of actors

6.1.3 Dynamic behaviour of A1

The timeline shown in *Figure 46* assumes that the execution time of actor A1 is fixed, which in streaming applications is not true (this was verified in the study case presented in chapter 3). The MP3 and AAC decoder showed a dynamic behaviour because the execution time changes according to the input samples values. This can have consequences in the execution and it is

illustrated in the *Figure 47*. The figure shows that it may happen that actor A1 is still executing when the interrupt routine is triggered and this will make the input buffer underflow because the DMA transfer will be stopped. In order to restart the DMA transfer, actor A1 has to verify that there is no more data in the buffer and has to call the DMA routine to reprogram the DMA.

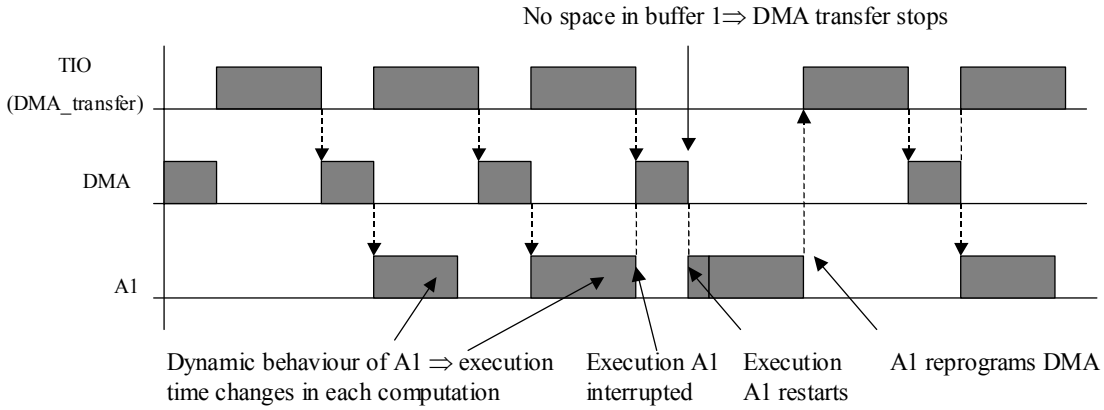


Figure 47 -– timeline execution of actors when problem occurs

The case study presented in chapter 3 showed the dynamic behaviour of the applications. It was verified that the execution time changes in each main process execution. With trace-driven analysis it was verified the dynamism but there is no knowledge if the application traces used in this study represent the worst-case execution time.

6.2 Multiprocessor Simulation with communication via shared memory

Figure 48 illustrates a streaming application that is mapped onto a multiprocessor architecture. Each tile includes a cache with the reference configuration derived from the case study in chapter 3. The execution is done in Round-Robin fashion in case more than one actor is mapped onto a processor. It is not possible to simulate TDMA in the simulation environment because that requires the implementation of a Kernel to save the status of the tasks upon a task-switch and to give a fixed budget for each actor in the application. The time factor was an important constraint of the work and it was not possible to evaluate the TDMA situation. In this experiment actor A1 and A2 are mapped onto processor 1 and actor A3 is mapped to processor 2.

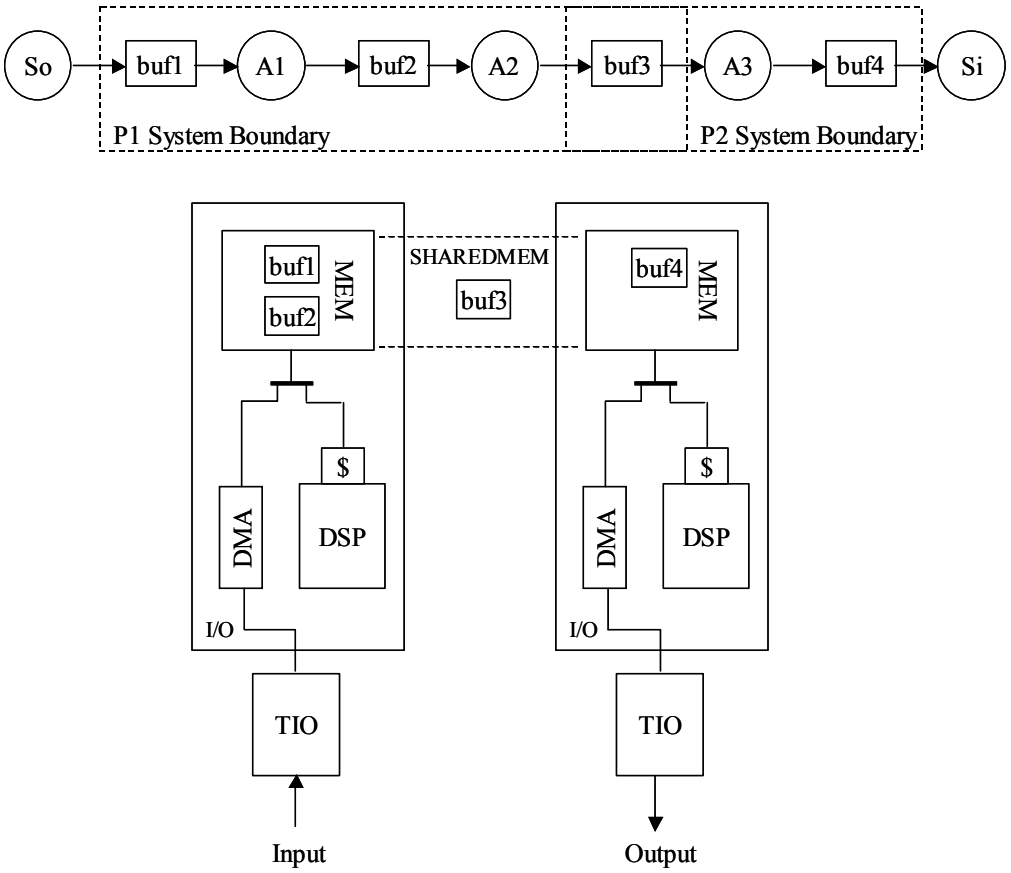


Figure 48 –Streaming application mapped onto multiprocessor architecture

6.2.1 Data coherence between cache and memory

The flush and invalidate cache management instructions have to be included in the program code of the actors in order to ensure data coherence between cache and memory. The actions to ensure data coherence are shown in *Figure 49*, where invalidation and flushing of the data is done in the end of the execution of the actors.

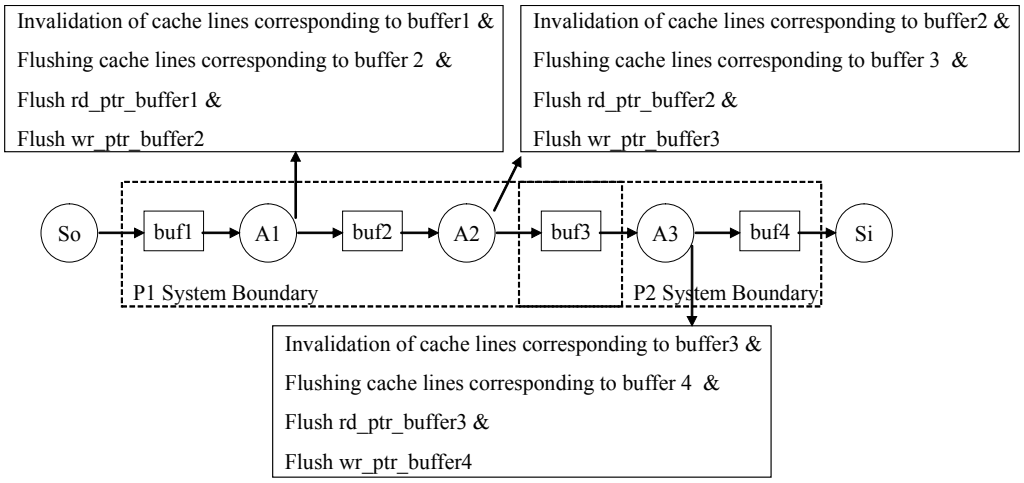


Figure 49 – Maintaining data coherence between cache and memory

In this experiment actors A1, A2 and A3 are identical and they consume a token with the same data size. Assuming that the WCET of the actors is the same we can model the application with implementation-aware HSDF. *Figure 50* shows the HSDF graph and the excepted execution timeline.

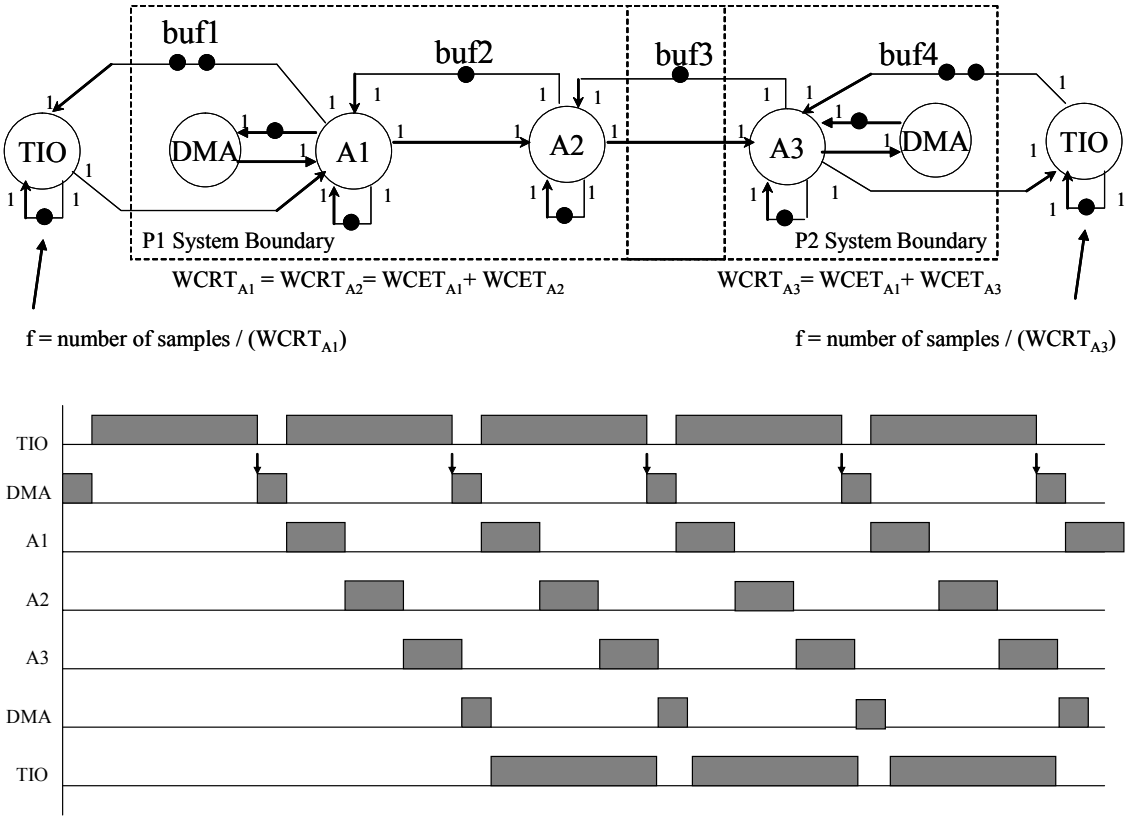


Figure 50 – HDSF graph of the streaming application and timeline execution

There is limited amount of parallelism that can be achieved in this simple application due to the data dependencies between the actors.

6.2.2 ISS simulation

The ISS simulation of the application shows the parallelism described above. More important, it shows that the response time of the actors in the simulation is smaller than the response time of the actor in the HSDF graph. The implementation-aware HSDF graph models the actors in a Worst-case response time yielding a minimum throughput. The response time of actors A1, A2 and A3 in the simulation is lower or equal to the response time in the implementation-aware HSDF graph, because the DMA access to the memory does not cause the processor to stall in every access.

6.3 Multiprocessor Simulation with communication via shared memory

This study was not successful because there was no time to finish a running simulation with the set-up described in section 5.2.2.

6.4 Conclusions

From the experiments described above two bottlenecks are identified:

- The DMAC is not autonomous and must be reprogrammed each time a number of tokens are copied to a buffer in memory. The actions of these task were described in Section 6.1.2.

- The DMA transfer has priority over the processor, which means that the processor will be stalled each time a sample is copied to the FIFO buffer in memory.

The case study presented in chapter 3 section 3.1 showed that the performance execution of an audio streaming application is more dependent of other types of data (i.e. static, scratch and ROM data) when compared to I/O stream data. The conclusion is derived by the fact that the I/O data impact in the cache miss rate is considerable lower than the others. knowing that, the probability of the number of processor accesses to the main memory is much higher than the DMA.

6.4.1 Recommendation

Based in the last analysis, it seems plausible to consider that the processor should have priority over the DMA for a number of cycles. For example, in ten cycles, the DSP has nine cycles of priority and the DMA has one. This could improve the execution time of the actor running in the processor.

Chapter 7

Conclusions and Future Work

Trace driven analysis proves to be a good method to evaluate L1 cache performance; the results obtained are accurate within the context explained in chapter 3. It proved to be a good method for the architecture exploration and it may be possible to improve the cache performance with the recommendation given in chapter 3.

ISS simulation allowed recreating the environment with enough detail that the events of the real system could be observed. In the example described in chapter 6 it was possible to consider:

- Interrupt handling
- Scheduling
- Interaction with the cache
- Supplying and consumption of samples by external sources.

With the ISS simulator is difficult to analyse throughput and latency. With this level of detail it becomes hard and time consuming to analyse complex graphs. This also relates with variation of system parameters, for example modifying buffer sizes. The use of a data-flow simulator for simulation of dataflow graphs should be considered for exploration of more complex graphs. It would allow studying a streaming application that is represented in a SDF graphs in a much less time. To consider is the trade-off between accuracy and simulation time.

For large and complex tasks, which may contain many input data-values dependent conditions, it is difficult and impractical to obtain a tighter bound on the WCET. Although it is possible to verify the dynamism of each process execution with trace-driven, it is not possible to know if the traces available represent the worst-case execution. Still a plausible way to estimate the execution time is with the ISS simulator, running a real-time application for a long period in order to derive this value.

In [1] a design flow to obtain a mapping of soft-RT applications is presented where the value WCET is not considered. The study considers the fact that the WCET is complex to obtain and it may be too pessimistic. The deadline miss probability of real-time applications mapped onto multiprocessor architectures is the focus of the work. The design flow identifies the bottlenecks in the mapping of the application on to a multiprocessor system. The bottlenecks are removed by adaptation of the FIFO buffer capacities or an increase in the time slice for TDMA arbitration in order to not increase the end-to-end deadline miss probability. The design flow allows that in a few optimisation iterations to reduce the miss probability of real-time applications.

Programming in the ISS simulation environment proved to be time consuming and affected the goals of this thesis because the applications are developed in a mixed of assembly and C and this affected decisively the progress of the work. The available framework, based on real-time applications (streaming kernel and decoders) and a network manager (setting applications and communication between tasks) was not open for modification and did not consider the use of cache and interrupt handling.

For the study of the alternative architecture a FIFO buffer dll was developed. Unfortunately due to the short remaining of time was not possible to complete the implementation and have a successful simulation. The simulation environment is flexible but it required more time and

knowledge than the time left during this master project. The simulation of an alternative architecture was not successful and therefore the comparison between architectures did not take place. Still it is possible with the available tools to explore ISS simulation with network based FIFO communication. The small example was intended to illustrate that situation.

For the current processor architecture, the study of an autonomous DMAC should be considered; it would to improve the execution time of the applications running in the processor. The example described in Chapter 6 explains the identified bottleneck. The simulation environment does not support pre-emptive task switching and therefore it was not possible to evaluate for example TDMA scheduling. To access the main memory, the arbiter gives priority to the DMAC. A possible modification on the arbiter priority should be considered and subjected to further analysis based in the observations described in chapter 6.

Applications that require a high computational performance can only be provided by multiprocessor architectures. The current platform uses central resources: bus and memory. These central resources limit the scalability of the system. In hard-real time applications timing deadlines are very important, it is important that the architecture is predictable. The use of caches and central resources insert uncertainty and it is hard to predict if the deadlines are going to be met.

7.1 References of Chapter 7

[1] Marco Bekooij, Sonali Parmar, Maarten Wiggers, Orlando Moreira and Jef van Meerbergen, Mapping of Soft Real Time Applications with Deadline Miss Probability Constraints. Submitted to date 2006.

Appendix A

MP3 Decoder and AAC Decoder Case Study

A.1 Required memory resources

The required memory resources (in 16 bit words) for the MP3 decoder application [1] are shown in the table 1.

Algorithm	Input buffer	Output buffer	Static RAM	Scratch RAM	Data ROM	Program ROM	Stack
MP3 decoder	977	1154	4752	2262	10304	12455	31

Table 2 – MP3 Application required resources

The required memory resources (in 16 bit words) for the AAC decoder application [1] are shown in table 2.

Algorithm	Input buffer	Output buffer	Static RAM	Scratch RAM	Data ROM	Program ROM	Stack
AAC decoder	1024	4096	2780	3492	13608	14820	54

Table 3 –AAC Application required resources

A.2 Memory references in the traces

Between brackets is the percentage of the respective memory.

	MP3	AAC
ZM R	655734	1711445
XM	390698 (53.13%)	929311 (58.02%)
YM	344699 (46.87%)	672645 (41.98%)
XM R	336535 (45.76%)	612670 (38.25%)
XM W	54163 (7.36%)	316641 (19.76%)
YM R	245616 (33.40%)	423940 (26.46%)
YM W	99083 (13.47%)	248705 (15.53%)

Table 4 – Number of Memory and data references in the traces

	XM R	XM W	YM R	YM W
MP3	7019	2880	2464	8257
AAC	9817	7609	12549	9852

Table 5 – number of different data memory references in the trace

A.3 XY$ vs X$ Y$ model profiling results

The notation is the following one: memory_size(Kw)_associativity

- I$ Instruction memory
- D$ double ported XY shared data memory
- X$ uni-ported data memory
- Y$ uni-ported data memory

Trace simulation	Cache Configuration (cache&size&ways)	Total cache size (I$+D$)	DSP miss rate
MP3||AAC	I$4Kw2W D$4Kw4W	8Kw	4.39
MP3||AAC	I$4Kw2W X$2Kw4W Y$2Kw4W	8Kw	6.33
MP3||AAC	I$2Kw2W X$4Kw4W Y$2Kw4W	8Kw	5.25
MP3||AAC	I$2Kw2W X$2Kw4W Y$4Kw4W	8Kw	5.52
MP3||AAC	I$8Kw2W D$8Kw4W	16Kw	1.87
MP3||AAC	I$8Kw2W X$4Kw4W Y$4Kw4W	16Kw	3.82
MP3||AAC	I$4Kw2W X$8Kw4W Y$4Kw4W	16Kw	3.32
MP3||AAC	I$4Kw2W X$4Kw4W Y$8Kw4W	16Kw	3.02

Table 6 - XY$ vs X$ Y$ model profiling results

A.4 Cache Size, Line size and associativity level

In *Figure 51* and *Figure 52*: LS denotes the line size and (2, 4, 8, 16, 32, 64 and 128)w denotes the number of words in a cache line.

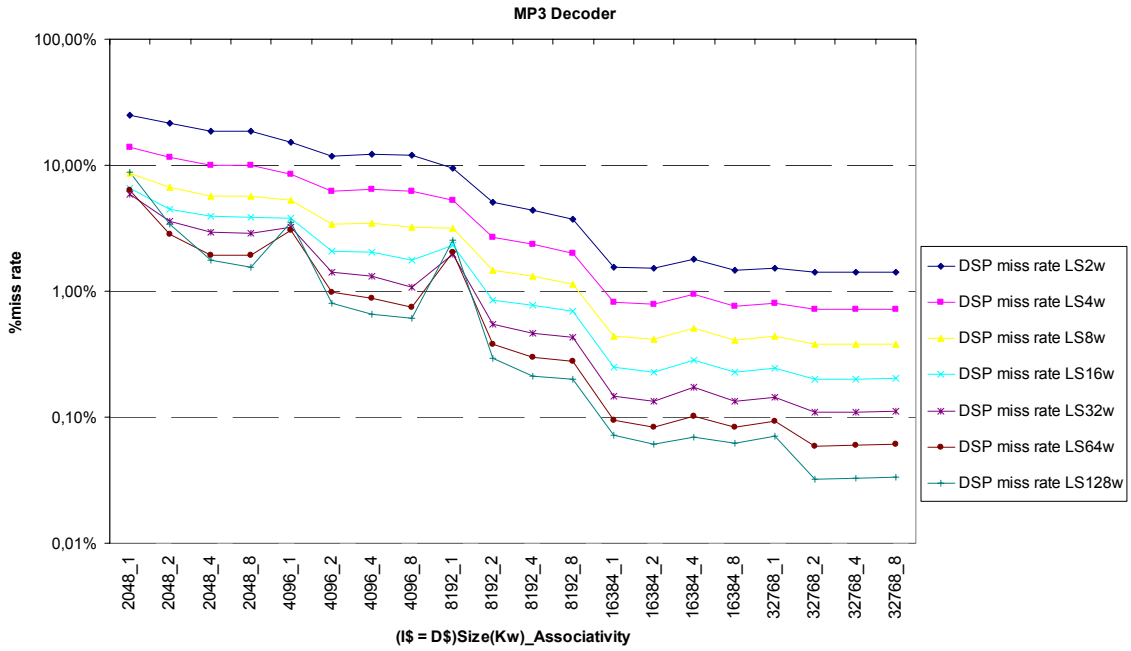


Figure 51– MP3 decoder miss rate as function of the cache configuration

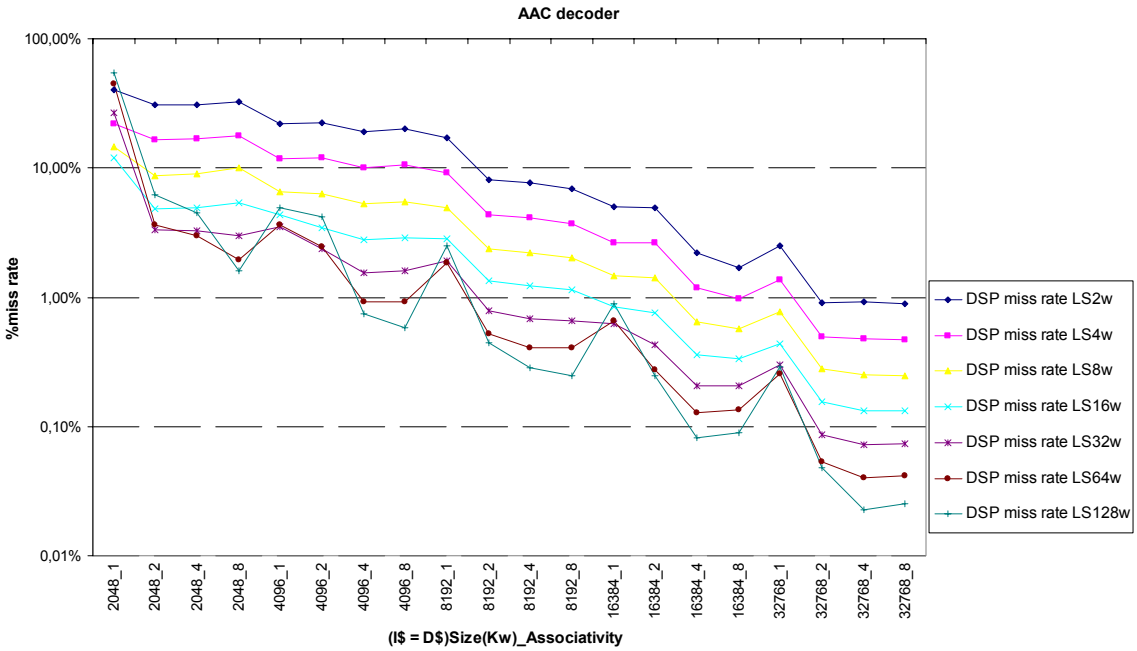


Figure 52 - AAC decoder miss rate as function of the cache configuration

In the following graphs the cache line size is assumed to be 8 words.

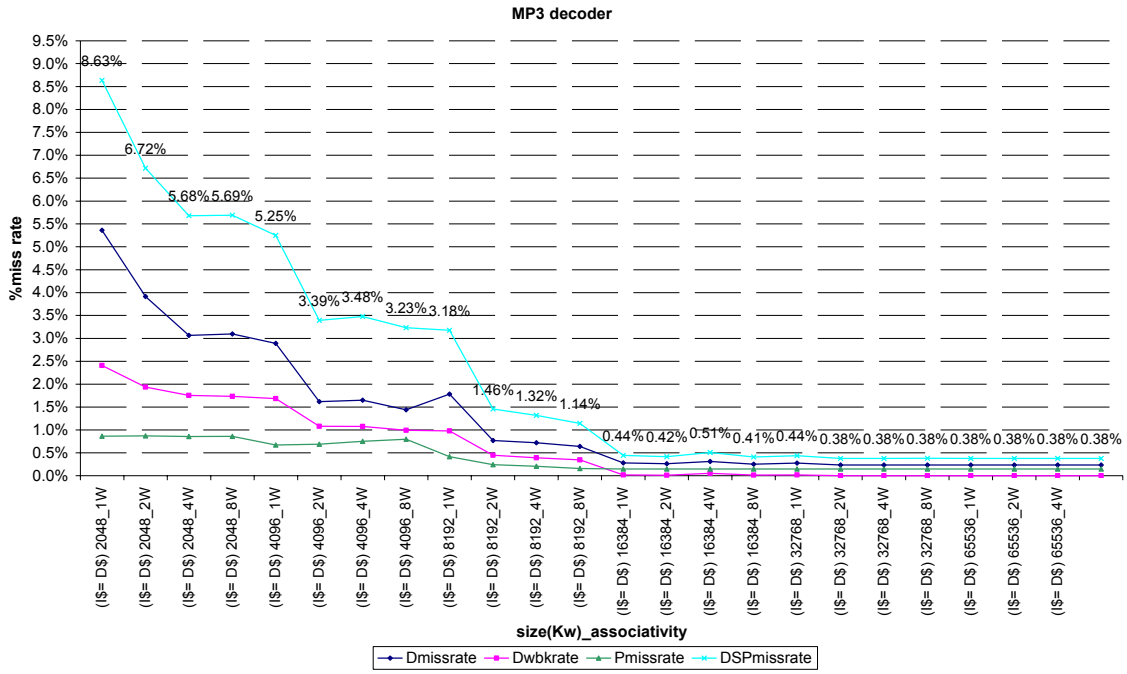


Figure 53 – MP3 decoder miss rate as function of the cache configuration

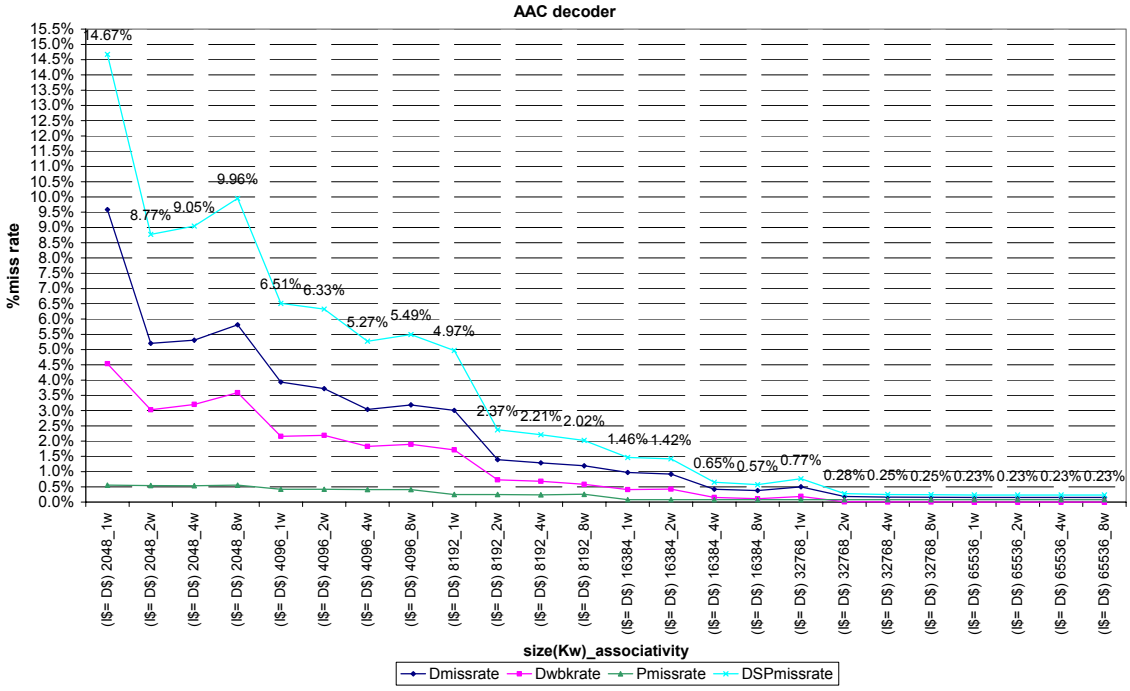


Figure 54 - AAC decoder miss rate as function of the cache configuration

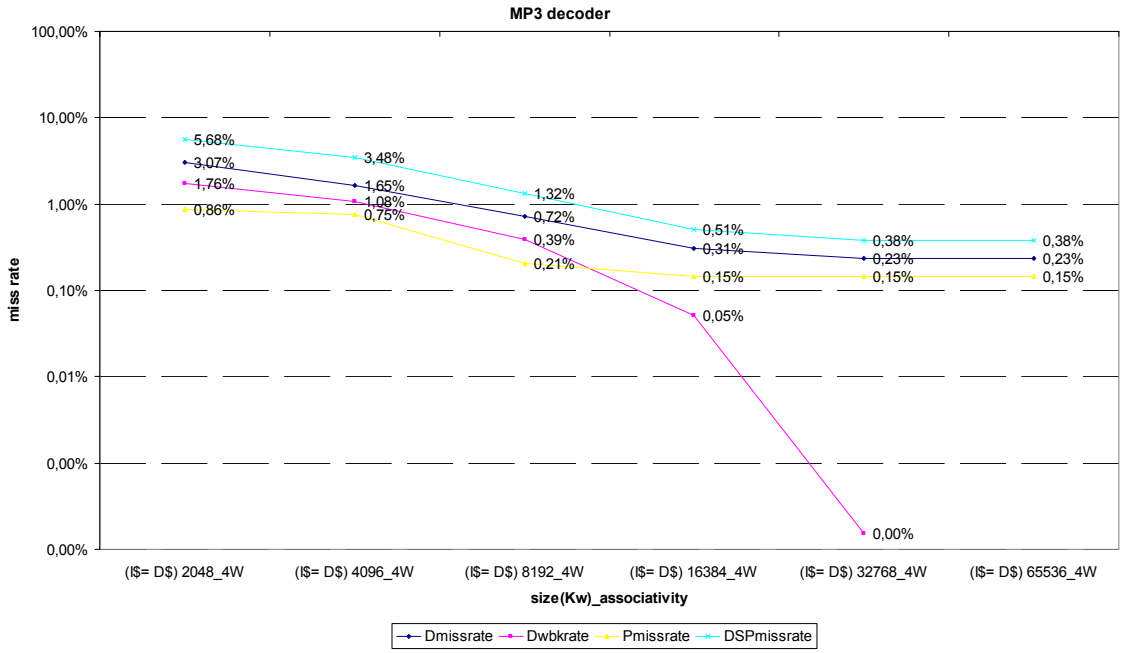


Figure 55 – log-log scale of the MP3 miss rate as function of cache configuration

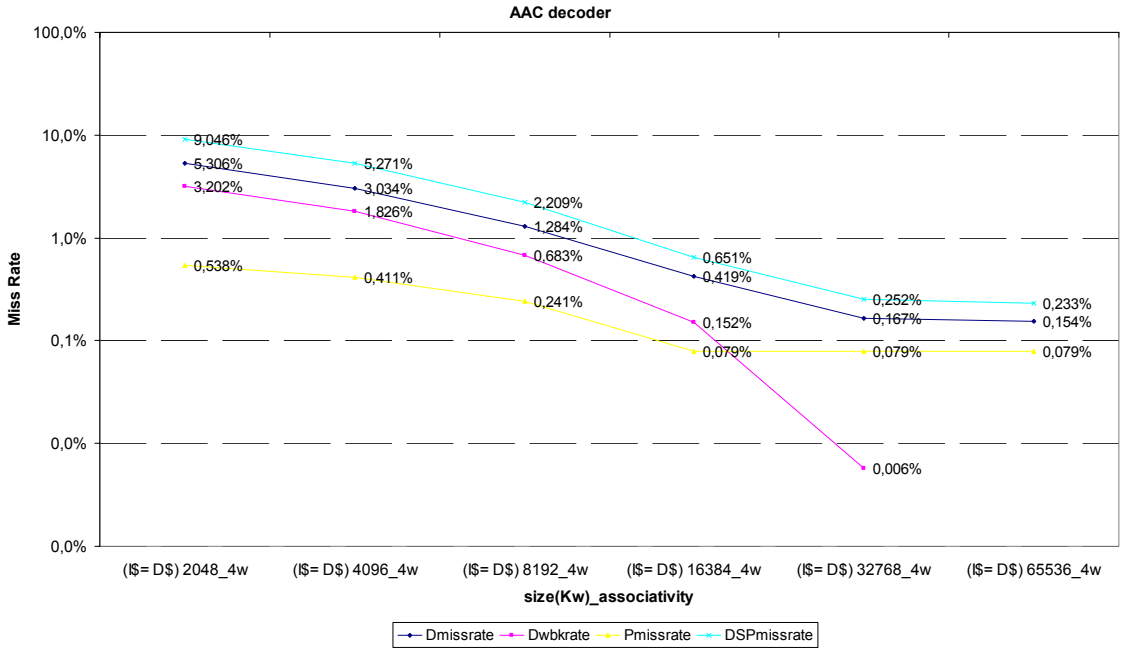


Figure 56 - log-log scale of the MP3 miss rate as function of cache configuration

A.5 Algorithm Behaviour

The x-axis denotes the algorithms procedure being executed (init – initialization routine, sync – synchronization, p1,2,3,4,... – main process routine)

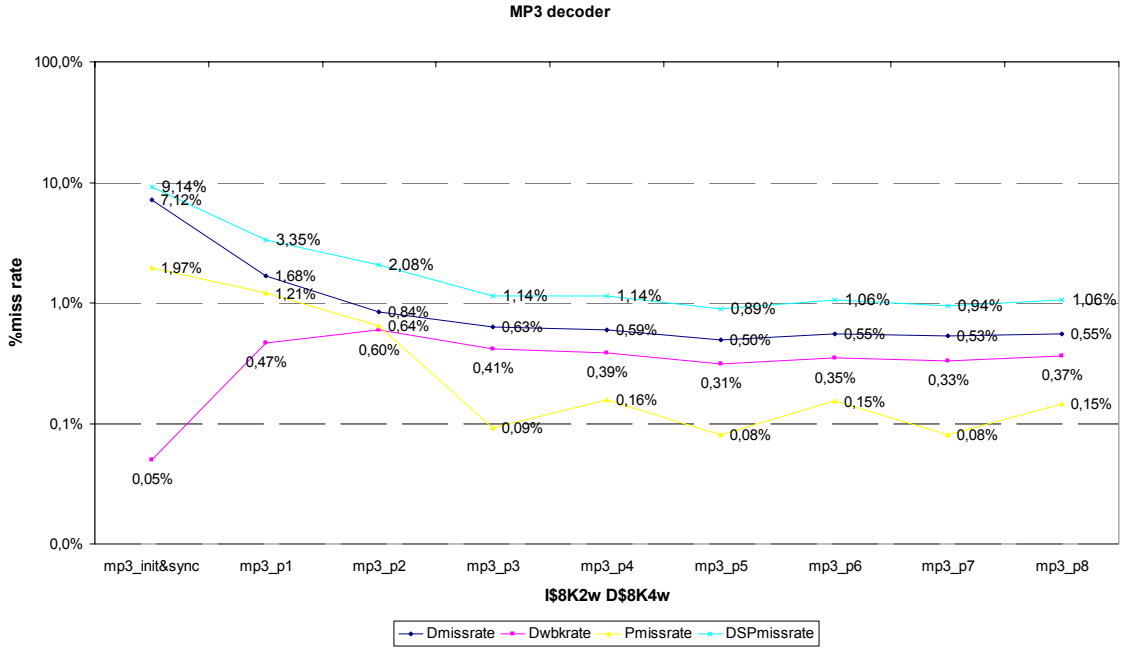


Figure 57 – MP3 miss rate as function of the procedure calls

Write back	Drm	Dwm	Dwb	Prm	cpu cycles	Dmiss rate	DwBK Rate	Pmiss rate	DSPmiss rate
Mp3_init	3	570	2	36	4777	11.99%	0.04%	0.80%	12.83%
Mp3_sync	132	8	3	159	5235	2.67%	0.06%	3.04%	5.77%
Mp3_p1	9	213	62	160	13234	1.68%	0.47%	1.21%	3.35%
Mp3_p2	393	375	549	587	91342	0.84%	0.60%	0.64%	2.08%
Mp3_p3	426	154	370	83	89917	0.63%	0.41%	0.09%	1.14%
Mp3_p4	349	188	350	143	90566	0.59%	0.39%	0.16%	1.14%
Mp3:p5	328	116	278	72	89561	0.50%	0.31%	0.08%	0.89%
Mp3_p6	342	156	319	140	90565	0.55%	0.35%	0.15%	1.06%
Mp3_p7	354	123	298	72	89918	0.53%	0.33%	0.08%	0.94%
Mp3_p8	345	156	332	132	90619	0.55%	0.37%	0.15%	1.06%
total	**2671**	**2059**	**2563**	**1586**	**655734**	**0.72%**	**0.39%**	**0.24%**	**1.35%**

Table 7 – MP3 miss rate as function of the procedure calls

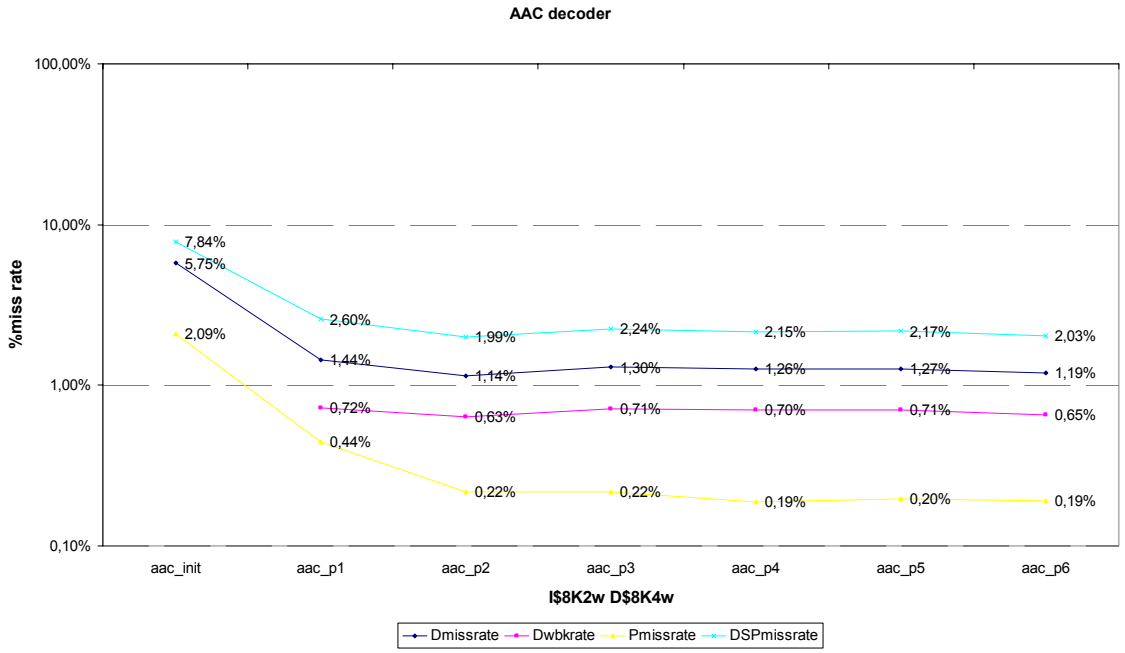


Figure 58 – AAC miss rate as function of the procedure calls

	Drm	Dwm	Dwb	Drm	cpu cycles	Dmiss rate	DwbK rate	Pmiss rate	DSPmiss rate
AAC_init	139	322	0	168	8020	5.75%	0.00%	2.09%	7.84%
AAC_p1	2538	1321	1934	1192	268490	1.44%	0.72%	0.44%	2.60%
AAC_p2	2597	992	1993	683	314479	1.14%	0.63%	0.22%	1.99%
AAC_p3	2578	1050	1988	606	278285	1.30%	0.71%	0.22%	2.24%
AAC_p4	2481	999	1945	521	275917	1.26%	0.70%	0.19%	2.15%
AAC_p5	2513	1028	1971	545	279419	1.27%	0.71%	0.20%	2.17%
AAC_p6	2337	1084	1865	549	286835	1.19%	0.65%	0.19%	2.03%
total	15183	6796	11696	4264	1711445	1.28%	0.68%	0.25%	2.22%

Table 8 – AAC miss rate as function of the procedure calls

A.6 Impact of data types

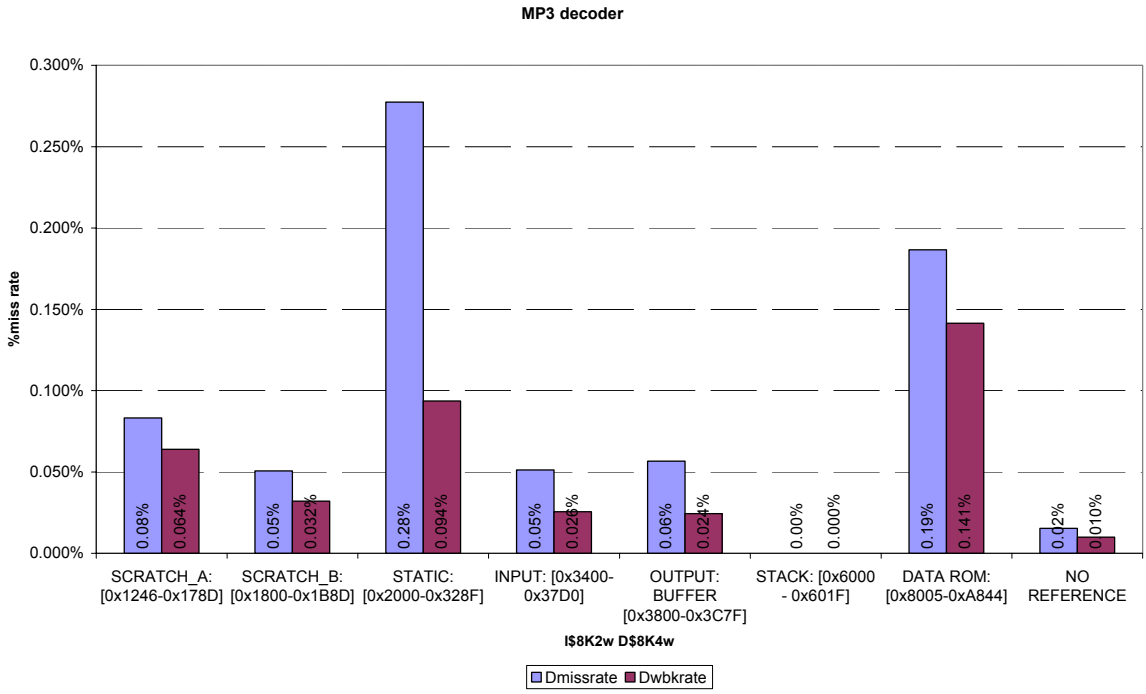


Figure 59 – MP3 miss rate as function of data type segments

Mp3	Drmisses	Dwmisses	Dwbks	Dmiss rate	Dmiss rate
SCRATCH_A [0x1246-0x178D]	5	541	419	0.083%	0.064%
SCRATCH _B [0x1800-0x1B8D]	6	326	210	0.051%	0.032%
STATIC: [0x2000-0x328F]	1024	795	614	0.277%	0.094%
INPUT BUFFER [0x3400-0x37D0]	309	27	168	0.051%	0.026%
OUTPUT BUFFER [0x3800-0x3C7F]	11	361	160	0.057%	0.024%
STACK [0x6000-0x601F]	0	0	0	0.000%	0.000%
DATA ROM [0x8005-0xA844]	1224	0	927	0.187%	0.141%
NO REFERENCE	92	9	65	0.015%	0.010%
total	**2671**	**2059**	**2563**	**0.721%**	**0.391%**

Table 9 – MP3 Data type table of results

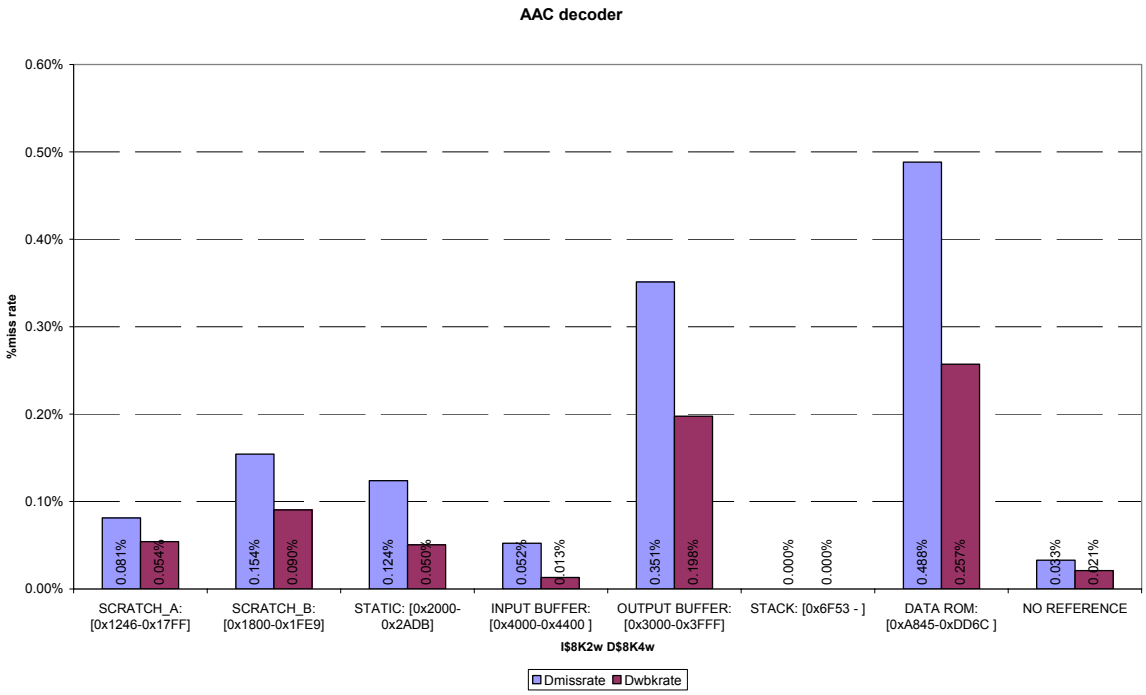


Figure 60 – AAC miss rate as function of data type segments

AAC	Drmisses	Dwmisses	Dwbks	Dmiss rate	Dmiss rate
SCRATCH_A [0x1246-0x17FF]	187	1201	924	0.08%	0.054%
SCRATCH _B [0x1800-0x1FE9]	428	2213	1546	0.15%	0.090%
STATIC [0x2000-0x2ADB]	1682	438	863	0.12%	0.050%
INPUT [0x4000-0x4400]	782	115	224	0.05%	0.013%
OUTPUT BUFFER [0x3000-0x3FFF]	3223	2789	3382	0.35%	0.198%
STACK [0x6F53]	0	0	0	0.00%	0.000%
DATA ROM [0xA845-0xDD6C]	8358	0	4400	0.49%	0.257%
NO REFERENCE	523	40	357	0.03%	0.021%
TOTAL	**15183**	**6796**	**11696**	**1.28%**	**0.683%**

Table 10 – AAC Data type table of results

A.7 Granularity

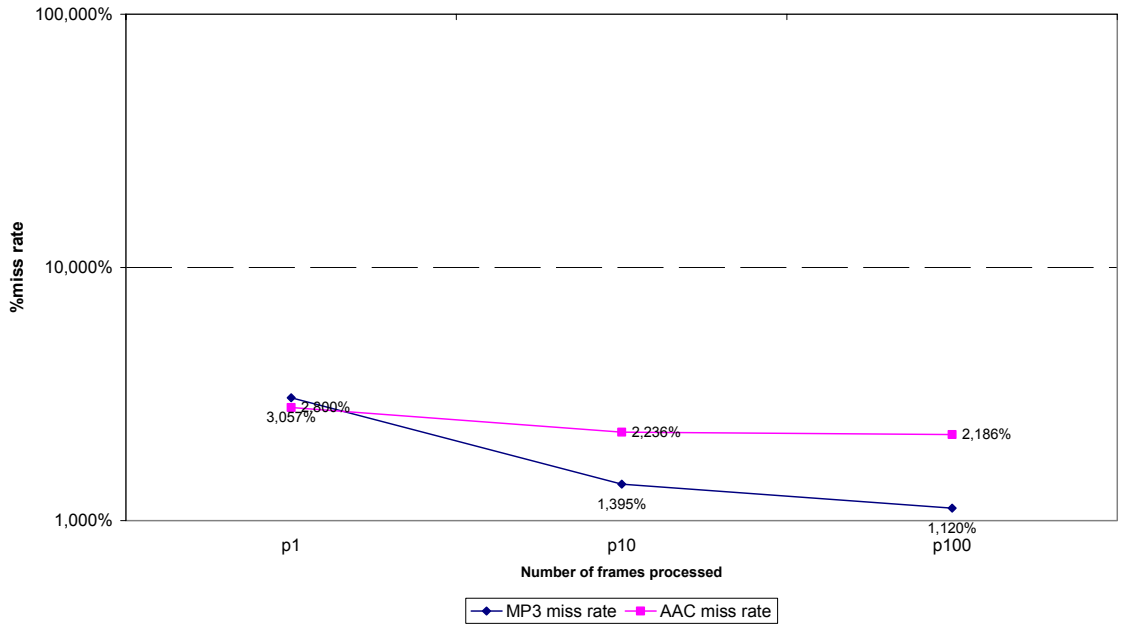


Figure 61 – Miss rate as function of application granularity

A.8 Cache behaviour as function of time

The x-axis denotes the number of execution cycles. The task-switch occurs after 1 million execution cycles. The miss rate is 'sampled' each 50K cycles of execution time.

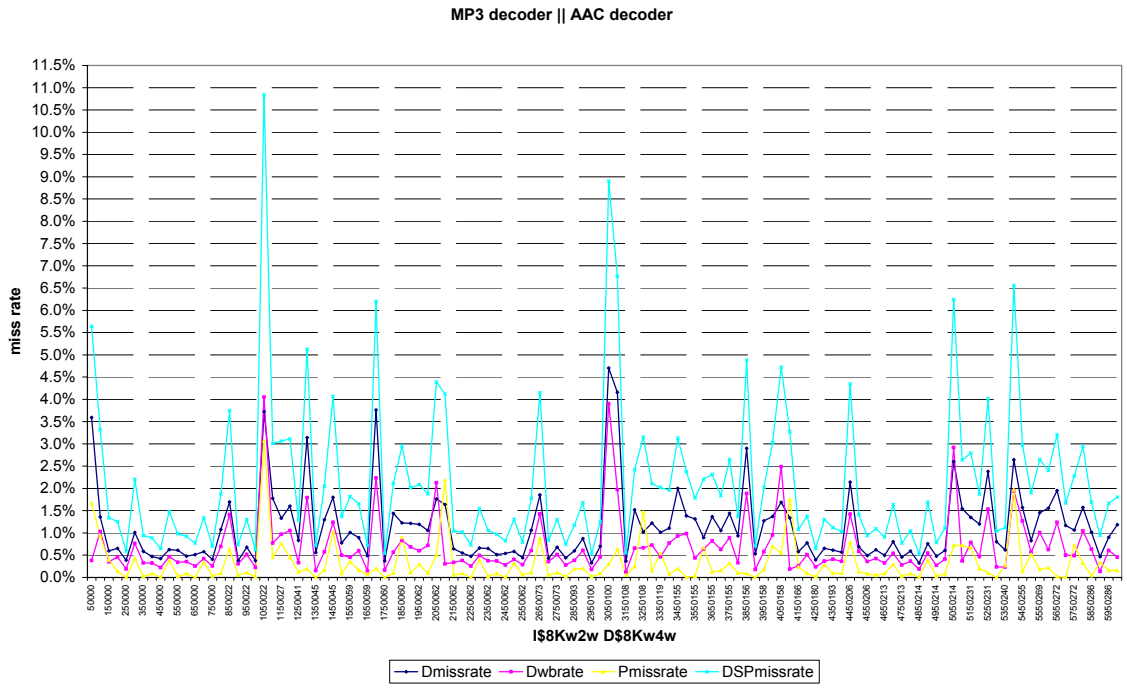


Figure 62 – Miss rate as function of execution time.