

Handling cache in real-time scheduling simulation.

Master thesis in Information and Communication Technology

TRAN HAI NAM

Department of Information & Communication Technology
University of Science and Technology of Hanoi
Intake 2011-2013

Supervisor: Professor Frank Singhoff
University of Western Bretagne
Lab-STICC, CNRS, UMR 6285

Tutor: Professor Daniel Hagimont
University of Toulouse
INPT/ENSEEIHT

Abstract

This is the report describing works in the research topic: "Handling cache in real time scheduling simulation" during the 6 month internship at Lab-STICC, CNRS, UMR 6285, University of Western Bretagne, in the context of SMART project and Cheddar tool set. Cheddar implements a simulator of real-time systems. This simulator allows architecture designers to model a real-time system and to investigate its behaviour [1]. The framework of SMART project includes a complex hardware platform with multi-core processor and multi-level of cache system, which is need to be verified. However, the current object models in Cheddar does not support simulation of this architecture.

In the work, there are three important achievements. The first achievement is to model and implement the cache systems in the Cheddar. The second achievement is to model the task's cache related behaviours and implement the model. The third achievement is to implement cache analysis and provide statistics in the process of scheduling analysis or feasibility test. Developments and improvements for the cache analysis can be made in order to adapt with the various type of caches and system configurations.

The report is organized in five chapters. "Chapter 1 - Introduction" gives an overview about the SMART project, Cheddar tool set, as well as the problem and motivation for the research. In "Chapter 2 - Cache memory and real time system", we revise basic knowledge about the cache memory and how cache effects real-time systems. "Chapter 3 - Related works" presents current work in this subject. "Chapter 4 - Handling cache in real-time scheduling simulation" is the important chapter, the abstract model, analysis method and implementation are introduced in this chapter. The 5th and 6th chapter are Evaluation and Conclusion.

Acknowledgements

Special thanks to professor Frank Singhoff and colleagues in Lab-STICC, UBO. Thank you very much for all the supports in not only research activities but also in living conditions in the first time come to France. Thanks my family for always supporting my decisions and provide back up whenever needed.

Tran Hai Nam, Brest, France, 01/09/13

Contents

List of Figures

List of Tables

1

Introduction

This chapter presents an overview about the context of the work and Cheddar tool set. Section 1.1 gives a short introduction about Cheddar tool set and its current state of development. In Section 1.2, problem and motivation for the research topic: "handling cache in real time scheduling simulation" is presented. Section 1.3 gives the objectives and of the work. In Section 1.4, the overall approach of the research is presented.

1.1 Context

The context of the work is the Cheddar project. Cheddar [1] is a free real time scheduling tool supports modelling of hardware components such as: *processor*, *core* and *network*; software components such as: *task*, *resources* and *address spaces*. In the framework of Cheddar, real-time scheduling algorithms and feasibility tests are included. Cheddar is also possible to perform scheduling simulation with a graphic user interface. Cheddar is developed and maintained by a team composed of the Lab-STICC laboratory/University of Western Bretagne and Ellidiss Technologies.

The work of the internship is proceeded at Lab-STICC laboratory, advisor: professor Frank Singhoff. The work is implemented in the framework of the Cheddar tool set.

1.2 Problem / Motivation

The gap between processor speed and memory speed is increasing considerably. Modern processors can run normally at clock speeds of several GHz and can execute more than one instruction per clock cycle. For example, a 3 GHz processor capable of executing 3 instructions per cycle has a peak execution speed of 9 instructions per ns.

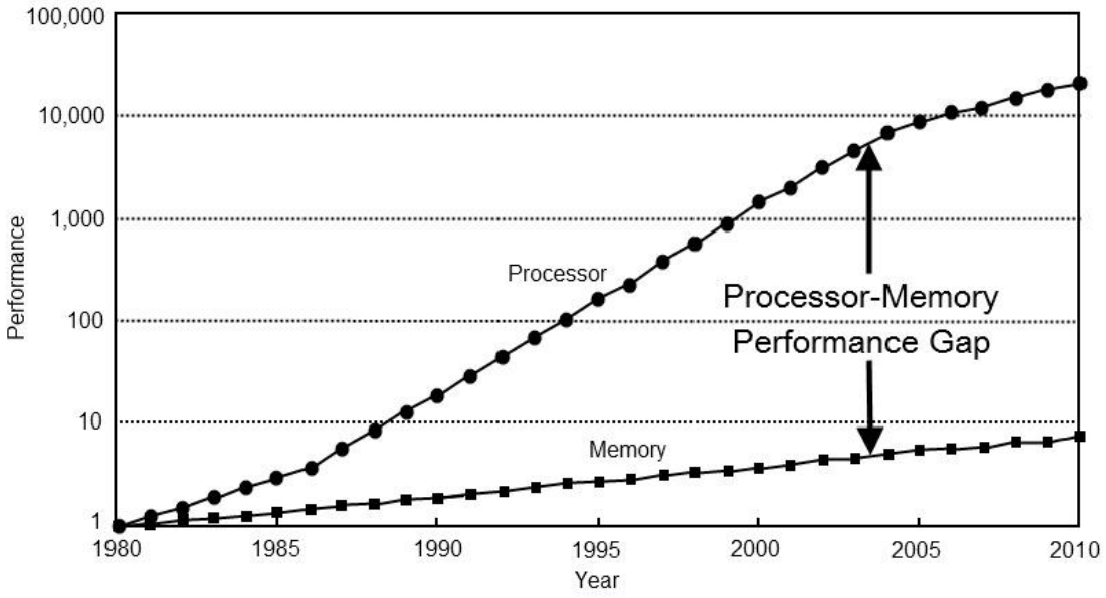

Figure 1.1: The gap between processor and memory speed. Adapted from Computer architecture: a quantitative approach, by John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau, 2011.

The speed of modern memories is quite slow. As we can see in the figure 1.2, an access to main memory or hard disk takes 50 - 100 ns. The processor has to wait for the data to arrive. If a processor can execute 9 instruction per ns, it can execute more than 450 executions in the time waiting to perform a single data access on main memory or hard disk. As we can see, memory access on high level memory is very slow operation comparing to an instruction access..

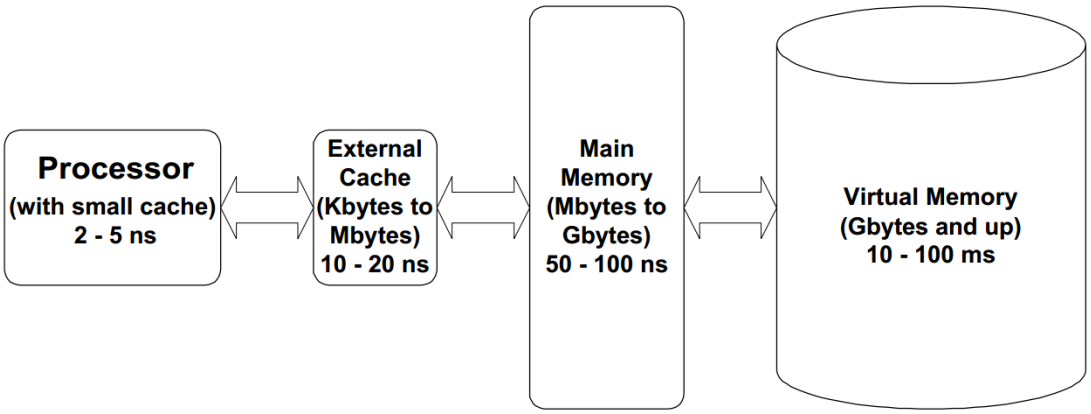

Figure 1.2: Memory speed. Adapted from Architecture of Parallel Computers, By E.F. Gehringer, G.Q. Kenney, 2005

The time it takes to load the data from memory is called the latency of the memory operation. It is usually measured in processor clock cycles or ns. Since memory accesses are very common in programs, memory access latencies cause bad impact on processor execution speed if they could not be avoided in some way.

Cache memories is introduced in order to solve in this problem. Cache are small, but extremely fast memories, lies between the processor and the main memory. Frequently used data are automatically

loaded into the cache. Despite the impressive performance in reducing the average memory access time, cache has not been really useful in real-time system because of its unpredictable behaviour:

- The associativity architecture - explained in Section 2.1 makes the cache becomes not a simply shared resources.
- Cache fetch algorithm and cache replacement protocol make it becomes harder to predict the cache miss/hit.
- The analysis of cache should be performed in the program source code level. It is difficult to perform modelling at this level in order to perform the analysis.

There is a trade-off when using the cache in real-time systems. It provides significantly performance increase but it also creates unpredictable timing behaviour. The execution time of a program can vary because of cache hit/miss. The use of cache will only be suitable if there is a reasonably bound on the performance of the program using cache can be predicted. It is good to have a strategy to predict the cache behaviour for real-time systems.

Today, the current simulator in Cheddar and its Architect Description language handle a very simple model of the hardware part. Hardware of a real-time system is simply modelled as a set of processors. In the SMART project, the targeted systems are quiet more complicated. The SMART project expects hardware platforms composed of multicore processors and several levels of cache units.

Indeed, in the framework of the SMART project, a case study, which is a real-time system composed of several processors and cache systems connected by a bus, need to be modelled. This system is dedicated to a multimedia application and requires a high level of performance. Then, the Cheddar simulator must be able to predict the performance of this system prior to its implementation. The current Cheddar provides very few simulation features for the hardware models focused by the SMART project (systems including cache systems). Then, the objective is to extend Cheddar to those hardware systems.

1.3 Objectives

For this internship, there are five objectives need to be completed:

1. *Do a survey of the cache consistency protocols and their compliancy with real-time scheduling technics applied on the hardware platforms investigated in the SMART project.*
2. *Investigate the performance criteria the simulator have to* compute: Different performance criteria have to be chosen for the simulator to compute. For each criteria, it is necessary to prove efficient algorithm to compute it. Performance criteria we look for may reflect the processor, the bus, the caches and the memory utilization factor.
3. *Update the current ADL of Cheddar in order to model real-time systems with cache systems*: Cheddar is implemented by a Model-driven process. The metal-model of the Cheddar ADL is expressed with a modeling language called “EXPRESS”. The Cheddar source code is then automatically produce from this EXPRESS model [2]. A new version of this EXPRESS ADL

metamodel should be proposed in order to allow the modelling of cache systems. Finally, XML printers and parsers of Cheddar will have to be updated according to this new release of the Cheddar ADL.

4. *Implement selected consistency cache protocols, scheduling technics and selected*: implement the simulator according to the scheduling algorithms and the cache consistency protocols selected in step 1), and according to the performance criteria selected in step 2). The simulator must be implemented in the Cheddar simulator engine. The simulator engine is the part of the Cheddar toolset that is responsible for ADL model simulation. This software will be written in Ada 2005 performance criteria in the Cheddar toolset.

5. *Test the contributions with the multimedia application case study*: The last step of this work will be the validation of the developed simulator with a case study.

1.4 Overall approach

We focus on the problem: "Handling cache in real-time system simulation". First, cache system is modelled in EXPRESS language, other hardware architecture such as processor, core is supported by Cheddar framework. The work is described in Section 4.2. Secondly, we model a program's control flow graph with cache behaviour in EXPRESS language. The work is described in Section 4.3. Then, we perform generating the source code using the tool platypus in order to integrate the model into current Cheddar framework. Next step is implementing an analysis method points out the effect of pre-emption on cache into Cheddar framework. The analysis method is Useful Cache Block, applied on Direct mapped instruction cache. The work is described in Section 4.4.Finally, the simulation engine is updated in order to perform simulation. The work is described in Section 4.5

2

Cache memory and real time system

Cache memory is an efficient way to narrow the performance gap between the CPU and memory speed. The knowledge about how cache memory actually works is not common even if it has been described in many text books [3], technology reports [4] and undergraduate courses. In this section, a brief summary about how cache memory works is provided.

2.1 Basics concept about cache

2.1.1 Locality

One fundamental of cache memories is locality. There are two types of locality:

- *Temporal locality* (locality in time): if a program use an address the chance to re-use it in the near future is bigger than other addresses.
- *Spatial locality* (locality in space): addresses that are close to each other in address space tend to be referred close in time too.

Base on the spatial locality, data in higher level memory (e.g RAM) should be loaded into at one the cache to take advantage of bandwidth and latency. Because of the locality, there are algorithms to pre-fetch data to the cache; however, in the current work, we did not address pre-fetch algorithms.

The smallest piece of data, which a cache can handle, is *cache line*. The physical location in the cache memory where a line is stored is called a *cache blocks*. In fact, for the simplicity we can consider/use two definitions equivalent.

2.1.2 Associativity

The hardware implementation of the cache memory can be expressed as a hash table. The key column is memory address of the block. There are three types of cache placement protocol:

- *Direct mapped*: a block can only be mapped to one distinct place in the cache. The mapping is usually:

$$(Block\ address)\ MOD\ (Number\ of\ blocks\ in\ cache)$$

- *Fully Associative*: a block can be placed anywhere in the cache.
- *Set associative*: a block can be placed in a set of places in the cache. The cache is called *n-way set-associative* cache. The cache is organized in *ways*; the most common is 2,4 and 8. In fact, we can consider the *direct mapped* cache as a 1-way set associative cache.

To indicate clearly the mapping scheme, let's take an example of 2-way set associative cache memory with 8 block frames. The main memory is organized as 64 blocks with a block size of 8 words.

The main memory contains 64 blocks, we need 6 bits to specify the address of a block in the memory. The cache contain 4 sets, we need 2 bits to specify the number of set. Size of words is 8, we need 3 bits to specify the position of a word. As a result, our memory address has 4 bits tag field, 2 bit set field and 3 bits word field.

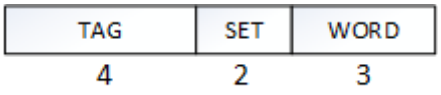

Figure 2.1: Cache addressing example

2.1.3 Reading

A block can be identified in the cache or not based on two informations: a *valid-* or *invalid-flag* and a *tag* for each block. When the computer system starts, the cache memory is flushed and all blocks are marked as invalid. The flag becomes valid when the data is written into the cache set. The data access to the cache is done as the following order:

- The set field is used to find the set.
- All valid tag fields in the set is compared to the tag field of the address. If the comparison is equal for one tag field, we get a hit. If there is no equal, we get a miss and the correct block must be loaded from the lower level memory.
- The word field is used to find the position of the word in the block.

Cache misses can occur for three reasons [3]:

- Compulsory: the line is not in the cache since the associated blocks are empty.
- Conflict: the line is not in the cache and all blocks associated to the set are being used.
- Capacity: the cache memory is full.

2.1.4 Writing

There are two policies for writing on the hit and two policies for writing on the miss. When a cache hit occurs, writing can be done in two different manners:

- *Write-through*: the writing on the cache is also made to the lower memory level

- *Write-back*: the writing is only done on the cache, writing one the lower memory is done when the block is replaced. In some articles, it is called *copy-back*

There are also two strategies for writing on the miss:

- *Write allocate*: the block is written in the lower memory and then loaded into the the cache.
- *No-write allocate*: the block is only modified in the lower memory.

2.1.5 Replacement Protocol

When the set for a block is full and a cache miss occurs, one of the blocks must be chosen to be replaced. It should be the block that will not be used in the near future. There are various algorithms for the replacement policies:

- *Random*: a block is randomly chosen.
- *Least Recently Used - LRU*: the least used block is replaced, the cache access in this case is logged.
- *Other*: FIFO, LFU and etcetera.

2.2 Cache problems in Real Time System

The problem with cache in real-time system is its limited predictability. It is difficult to obtain reliable information of cache performances even for a single task. In addition, impact on the performance of cache caused by pre-emption in a multi-task environment is also hard to be quantified.

The basic cache architectures, including associativity, fetch protocols and replacement protocols, makes it not a simple shared resources. Cache performance analysis often is based on source code level because necessary informations for analysis such as memory access trace are not easy to estimate.

In current real-time system analysis tools and simulators, cache is not included. It can have two meanings; the first one is that the cache is not included in the design of real time system, capacity of the task is the worst cache execution time (WCET) without cache. In this case, this is a pessimistic approach because as presented in section 1, the performance boost of cache is significant. Secondly, single task's cache performance is analysed, capacity of the task is the WCET with cache. However, it still left out the effect of pre-emption on the cache. In multi-task environment, pre-emption causes the cache related pre-emption delay (CRPD). When the task is pre-empted by another task, its cache content can be wiped out and time to refill is needed.

2.3 Conclusion

In conclusion of the chapter, we can notice that the complexity of the cache architecture limited its predictability. It is difficult to analyses all the features of cache to obtain a precise cache performance even for a single task. Researches on cache normally solves one or two problems. In

this project, the cache related pre-emption delay problem for direct-mapped cache is chosen. Even though the architecture of the cache is simple (no replacement protocol, fetch protocol), the work includes modelling and simulation approach. In the next chapter, the cache analysis for single task (cache intrinsic behaviour analysis) and multi tasks (cache extrinsic behaviour analysis) is described more rigorous. Examples of analysis tools is also presented.

3

Related works

In this chapter, a summary of cache memory analysis is described. Section 3.1 and section 3.2 indicates the two direction of analysis works. In section 3.3, the different between data cache and instruction cache is indicated. Finally, in the last section, notables real-time system scheduling simulators besides Cheddar are presented. The section 3.4 gives an overview about the tools and software and hardware architecture supported by them.

3.1 Intrinsic interference analysis

Intrinsic interference is the removals of task's instruction or data in the cache by itself. In the previous years, there are many program's intrinsic cache behaviour analysis techniques have been proposed. The main idea is to create an abstraction of the program (source code) and analyse it by using: integer linear programming [5], static analysis with graph colouring [6], abstract interpretation [7] or data flow analysis [8]. The objective of cache analysis in this context is to give a tighter bound on the WCET of a task. In our work, there are limitations if we focus on intrinsic interference:

- The complexion of the task model is high because the base of the analysis methods in this subject is the task's source code. The output of the analysis should be the cache behaviour of the task. So, even though we choose a higher level of abstraction, it is still required enough details to model the code. For example, in the method of Li et al [5], the input for an automatic linear programming solver is control flow graphs (CFG) generated from the source code by compiler.

- A complex model leads to impractical use of the work. It can be faster and less complex for a programmer to write the code and test it practically than try to estimate its behaviours to give input for our tool. The model at first need to be simple by the mean that it does not requires many inputs and those input is easy to estimate. Imagine that it is impractical if instead of give the scheduler the task's capacity, it is required to inputs number of variables, loop bounds for each loop, execution paths,...

- In the context of the CHEDDAR project, giving a tighter bound on the WCET means exclude the capacity attributes of the task. It requires a long development phase following in order to

give a WCET because analyse the cache only give the memory access time, not the WCET of the task.

To sum up, there are limitations in complexity and practicability when building an abstract model of program to analyse cache intrinsic interference.

3.2 Extrinsic interference analysis

Extrinsic interference can be defined as an delay to the pre-empted task, the side effect of it is the cache related pre-emption delay (CRPD). The delay is caused by:

- Impact of preemption on the cache content. If the data or part of data of two tasks are mapped into the same cache lines. Execution of the task can exclude cache content of the other tasks.
- Overall cost of additional reloads due to pre-emption.

The figure below is an example illustrating the effect of CRPD:

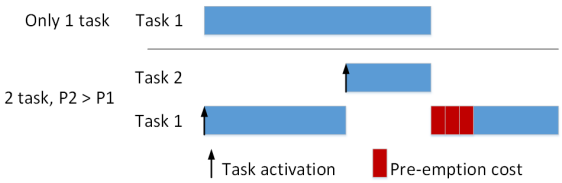

Figure 3.1: Extrinsic interference

There is a method named cache partitioning [9], which can be used to avoid extrinsic cache behaviour. The limitation of this method is it requires special hardware or software. In addition, it also means that in many cases we do not fully utilize the cache.

Consider a task with analysed cache intrinsic behaviour, extrinsic behaviour analysis 's objective is to calculate the CRPD. As a result, the WCET of the task in the previous analysis is increased. In the context of the Cheddar project, the cache extrinsic behaviour analysis increases the capacity of the task. A task can have a precise capacity. The important idea is it acts as an add-ons to the current Cheddar system. With the new capacity calculated, Cheddar can perform scheduling analysis normally. The requirement is that we need to have cache intrinsic behaviour analysis result, which will be modelled.

3.3 Data cache and instruction cache

Instruction access for one program is more predictable than data access.

- In one basic block in the control flow graph, the instruction access is sequential.
- Based on control flow graph of a task, we know the in and out paths for a set of instruction in one basic block. In other words, we know how a set of instruction is used.

As a result, in the early stage of the work, we focused on the instruction cache analysis.

3.4 Real-time system scheduling simulator

MAST[10] (Modeling and Analysis Suite for Real-Time Applications) is a tool-set supporting various analysis for real-time system such as: Worst-case response time schedulability analysis, blocking times and optimized priority assignment techniques. The hardware component abstraction of MAST model is generic, it includes: processing resource and shared resources. The support for cache and cache related pre-emption delay calculation techniques is not mentioned in the MAST publications.

STORM[11] is a Simulation TOol for Real-time Multiprocessor scheduling. The system is modelled by software and hardware architecture, the tool performs analysing, simulating and then produce result in form of diagram, reports or exportable content. In the documents on STORM websites (http://storm.rts-software.org/), it indicates that STORM supports modelling the processor as hardware, task, data and scheduler in software architecture.

YARTISS[12] is a real-time multiprocessor scheduling simulator which also supports energy consumption as a parameter. It allows simulation of task sets on multi processor architectures. The tool is written in Java with modular architecture. Users can add customized scheduling algorithm to the tool.

3.5 Conclusion

To sum up, even though there were many researches about cache memory in both intrinsic and extrinsic interference, real-time scheduling tools/simulators did not supported a detailed model of hardware architecture with memory components.

4

Handling cache in real time scheduling simulation

The problem statement is how to handle cache in real-time system simulation". Requirements for the solution are: a model of hardware platform and software architecture, an analysis method points out the effect of pre-emption on cache and a simulation tool.

In the next part of the section, we present an overview of the solution. The solution is implemented as a part of Cheddar framework. As a result, we also mention the existing hardware models in Cheddar.

4.1 Solution overview

A model of hardware platform:
The solution is implemented in the context of Cheddar project. In Cheddar, there are models of hardware components including: *processor*, *core*, *bus* and *memory*. The model of the cache is lacked, consequently, the first step in this project is modelling the cache and implementing the model in Cheddar project. Detailed model of the cache is presented in Section 4.2

A model of software
In Cheddar, software is modelled as a set of tasks with these following notable attributes: *capacity*, *deadline*, *priority*, *offsets*. A task can be *periodic*, *aperiodic*, *sporadic* or *poisson*. However, the current attributes of the task is not enough to model the cache behaviour of itself. As a result, an extended model is proposed in Section 4.3

An analysis method points out the effect of pre-emption on cache
We focus on the extrinsic interference analysis (presented in 3.2). We chose the method named Useful Cache Block (UCB), described in the work of Lee et al [13]. The method is presented in Section 4.4

A simulation engine
Cheddar is a real time scheduling simulation tool so there was an existing simulation engine in my research project. To adapt for cache support, we must integrate the result from the analysis method into the simulation engine. This work is presented in Section 4.5

4.2 Cache model

The model of the cache composes all the basic attributes presented in section 2.1. A cache is modelled by:

- *Name*: it is the unique name of the cache.
- *Cache Size*: it is the size of the cache - the number of bytes that a cache can contain.
- *Block Size*: it is an integer, which corresponds to the number of contiguous bytes that are transferred from main memory on a cache miss.
- *Associativity*: it is an integer, which corresponds to the number of cache locations where a particular memory block may reside.
- *Hit Time*: it corresponds to time to access to a memory block that is in the cache.
- *Miss Time*: it corresponds to the time to access a memory block that is not in the cache.
- *Cache Type*: it specifies a cache is instruction cache, data cache or an combined one.
- *Replacement Protocol*: it is a cache replacement policy, which decides which cache block should be replaced when a new memory block needs to be stored in the cache.
- *Coherence Protocol*: it is a coherency protocol, which maintains the consistency between all the caches in a system of distributed shared memory.
- *Write Policy*: it determines how the cache performs the write of the data in the cache back to the main memory.

The detailed explanation of each attributes is mentioned in 2.1

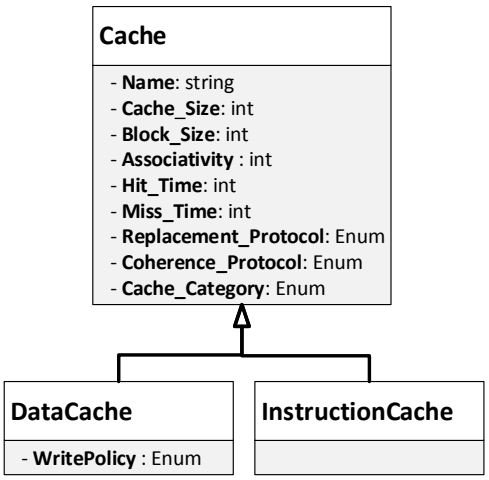

Figure 4.1: Cache model

4.3 Task model

First, we want to bring two important definitions:

- **Control flow graph** (CFG): a representation, using graph notation, of all paths that might be traversed through a program during its execution
- **Basic Block**: a sequence of consecutive instructions in which flow of control enter at the beginning and leaves at the end without halt or possibility of branching except at the end.

We used the basic block to model task's control flow graph. The important points are:

- We need to model the control flow graph. Consequently, each basic block must has previous block(s) and next block(s) - except the starting block(s) and ending block(s).
- Information about instruction/data access is composed in the model.

In the figure below, a basic block model in Cheddar is presented. A basic block is modelled by

- *Name*: it is a unique name of the basic block.
- *Previous Blocks*: it is a list basic blocks, which can be executed right after this basic block.
- *Next Blocks*: basic it is a list of basic blocks, which can be executed right before this basic block.
- *Instruction Offset*: it is the offset of the assembly instruction of this basic block in the main memory.
- *Instruction Capacity*: it is the capacity of the assembly instruction of this basic block.
- *Loop Bound*: it is the number of times a basic block can be executed.
- *Type*: it is the type of basic block, we have *START*, *MIDDLE* and *TERMINATE*.

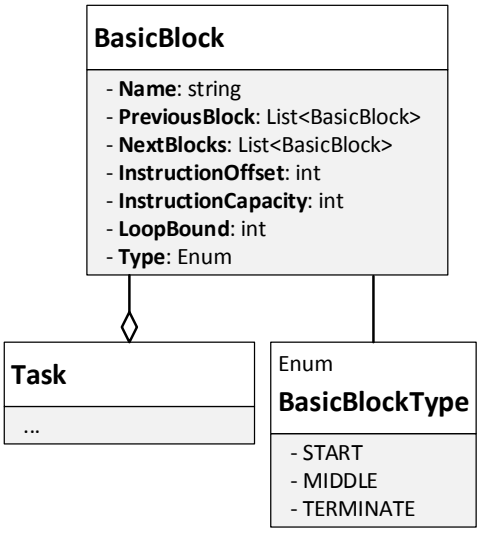

Figure 4.2: Task model

With the basic block model, it is possible to model an execution graph in the picture below. Conditional statements or loop statements are not modelled to keep the simplicity of the model. We only focus on which blocks can be the previous blocks and next blocks of one basic block.

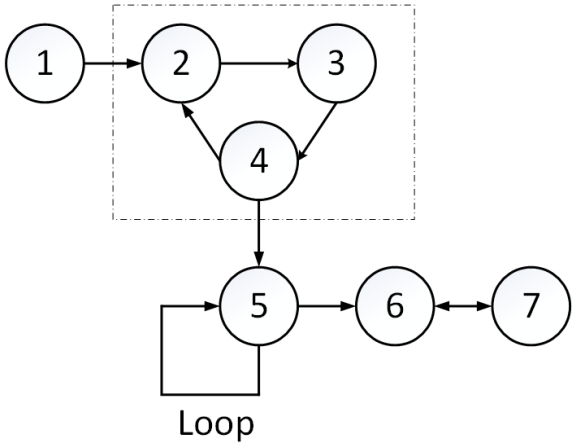

Figure 4.3: Task's flow graph

From the offset and capacity of the basic block, it is possible to calculate the mapping of the memory (data/instruction) block of a basic block in the cache memory.

4.4 Analysis method: Useful Cache Blocks

This is the analysis method described in [13]. In this section, we want to present the important idea, definition of the method.

4.4.1 Definitions

Useful cache block (UCB): A memory block m at program point p is called useful, if m may be cached at p and m may be reused at program point p' that may be reached from p with no eviction of m in the cache on this path. The cache block contains memory block m at p is an *useful cache block*.

Let us take an example with a direct mapped cache with 4 cache blocks: 0,1,2,3. At time t: the mapping is:

Cache block:	c0	c1	c2	c3
Memory block	m0	m5	m6	m3

Next memory access sequence: m4 -> **m5** -> **m6** -> m7. Cache mapping is:

Cache block:	c0	c1	c2	c3
Memory block	m4	**m5**	**m6**	m7

We can see that m4 and m5 are reused while they are still in the cache, as a result, c1 and c2 are useful cache blocks.

If we can find the number of useful cache block for each basic block, it is possible to estimate the cost of pre-emption occur in this basic block. This is the basic idea in the work in [13]. The authors introduced two definitions:

- **Reaching Memory Blocks** - RMB: set of memory blocks that may be reside in the cache block. (at the beginning of the basic block).
- **Live Memory Blocks** - LMB: set of memory blocks that may be first reference to cache block. In other words: memory blocks that are expected to be in the cache block.

The useful cache block is the cache block where:

$$RMB \bigcap LMB \neq \emptyset$$

In order to illustrate the RMB and LMB, we present an example with the same cache configuration mentioned above in Section 4.4.

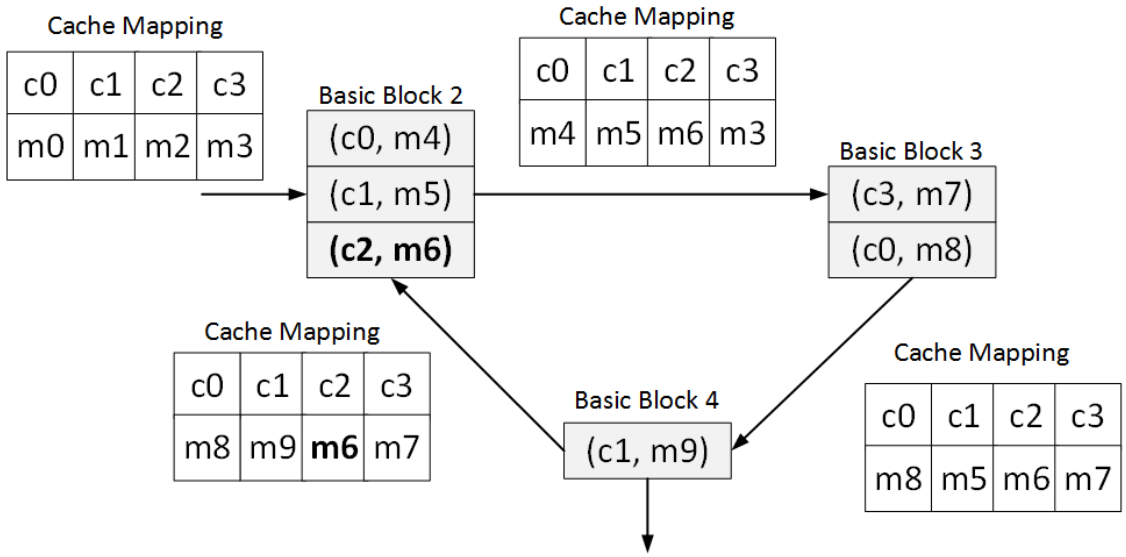

Figure 4.4: RMB and LMB example

The arrows presents the flow of the graph, we have a loop here. We can have a closer look at the basic block 2, cache block 2.

- RMB: m2, m6
- LMB: m2

The cache block c2 can be a useful cache block for the basic block 2.

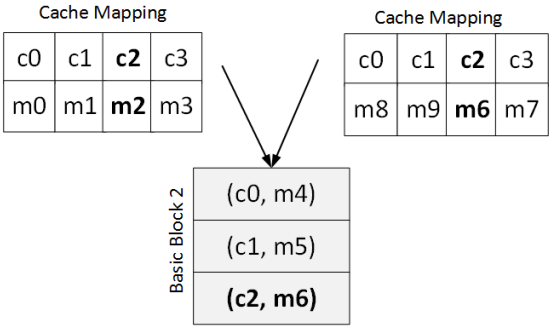

Figure 4.5: RMB and LMB for one basic block

4.4.2 Algorithm

As explained in the section above, a cache block is useful in one basic block if there is an existence of intersection between its Reaching Memory Block (RMB) and Live Memory Block (LMB). The algorithm to calculate the RMB and LMB is presented in [13]. The algorithm to calculate the RMB and LMB is based on the well-known iterative data flow analysis approach.

RMB Calculation

There are three new definitions that are needed to mention before presenting the algorithm:

- $gen^c[B]$: set contains the unique element - the memory block that is the last reference to the cache block c in the basic block B. This set is empty if basic block B does not have any reference to the cache block c, otherwise, $gen^c[B]$ contains one element: the memory. If the cache has n cache lines, for each basic block, we will have n sets $gen^c[B]$.

The example is used to illustrates $gen^c[B]$ for the basic block 1 and 2. We have:

$gen^0[B_1] = \{m4\}$
$gen^1[B_1] = \{m1\}$
$gen^2[B_1] = \{m2\}$
$gen^3[B_1] = \{m3\}$

$gen^0[B_2] = \emptyset$
$gen^1[B_2] = \{m5\}$
$gen^2[B_2] = \{m6\}$
$gen^3[B_2] = \emptyset$

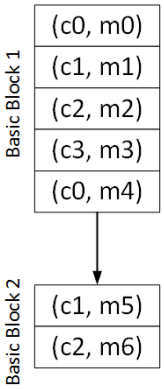

Figure 4.6: $gen^c[B]$ example

- $RMB^c_{IN}[B]$: RMB of cache block c just before the beginning of basic block B
- $RMB^c_{OUT}[B]$: RMB of cache block c just before the end of basic block B

There are two equations that are used to calculate the value of $RMB^c_{IN}[B]$ and $RMB^c_{OUT}[B]$:

$$RMB^c_{IN}[B] = \bigcup_{P\ a\ predecessor\ of\ B} RMB^c_{OUT}[P] \tag{4.1}$$

Equation 4.1: the memory blocks that reach the beginning of a basic block B can be derived from those that reach the ends of the predecessors of B.

$$RMB^c_{OUT}[B] = \begin{cases} gen^c[B] & \text{if } gen^c[B] \text{ is not null,} \\ RMB^c_{IN}[B] & \text{otherwise.} \end{cases} \tag{4.2}$$

Equation 4.2: $RMB^c_{OUT}[B]$ is equal to $gen^c[B]$ if $gen^c[B]$ is not null and $RMB^c_{IN}[B]$ otherwise. The two data flow equations can be solved using a well-known iterative approach. It starts with $RMB^c_{IN}[B] = \emptyset$ and $RMB^c_{OUT}[B] = gen^c[B]$ for all basic blocks, cache blocks and use the equation 4.1 and 4.2 repeatedly until the values of $RMB^c_{IN}[B]$ converge.

The algorithm below is presented in [13]:

for each basic block B **do**
begin
  for each cache block c **do**
  begin
    $RMB^c_{IN}[B] := \emptyset$;
    $RMB^c_{OUT}[B] := gen^c[B]$;
  end
end
change := **true**;
while *change* **do**
begin
  change := **false**;
    for each basic block B **do**
    begin
      for each cache block c **do**
      begin

$$RMB^c_{IN}[B] = \bigcup_{P \text{ a predecessor of } B} RMB^c_{OUT}[P]$$

        $oldout := RMB^c_{OUT}[B]$

        if $gen^c[B] \neq \emptyset$ **then**
          $RMB^c_{OUT}[B] := gen^c[B]$;
        else
          $RMB^c_{OUT}[B] := RMB^c_{IN}[B]$;

        if $RMB^c_{OUT}[B] \neq oldout$ **then**
          change := **true**;
      end
    end
end

LMB Calculation

LMB is calculated similar to RMB, but it is backward data flow problem so it is calculated from the end basic block. The definition of $gen^c[B]$ becomes:

- $gen^c[B]$: set contains the unique element the memory block that is the first reference to the cache block c in the basic block B.

$$LMB^c_{OUT}[B] = \bigcup_{P\ a\ successor\ of\ B} LMB^c_{IN}[S] \tag{4.3}$$

$$LMB^c_{IN}[B] = \begin{cases} gen^c[B] & \text{if } gen^c[B] \text{ is not null,} \\ LMB^c_{OUT}[B] & \text{otherwise.} \end{cases} \tag{4.4}$$

Useful cache block calculation

Once the RMBs and LMBs are computed, we can determine a cache block in one basic block is useful or not. The total number of useful cache blocks is given by the number of useful cache blocks *plus one*. It is because if a task is pre-empted within a memory block, the cache block mapped by the memory block is useful at the pre-emption point.

4.4.3 Pre-emption cost

For basic block, the cache related pre-emption (CRPD) delay is calculated by:

$$CRPD := CacheMissTime * (NumberOfUsefulBlock + 1)$$

From the calculated CRPD for all basic blocks, we build a pre-emption cost table: table of CRPD of basic blocks in the task. The table is sorted from the biggest CRPD to the smallest. Two hypothesis are taken. Firstly, each basic block can be pre-empted only one time. Secondly, the first pre-emption occur on the worst case.

# Pre-emption	1	2	3	4	...
Cost	40	35	30	28	...

Table 4.1: Pre-emption cost table

4.5 Implementation

In this section, we present how to integrate the analysis method into the Cheddar simulation engine. Firstly, we model the new hardware and software architecture in EXPRESS language. Secondly, the tool *platypus*[14] is used to generate Ada code, which is compatible with the framework of Cheddar. Finally, we implement the analysis method in the scheduling simulation.

- The result of the analysis in section 4.4 gives us the pre-emption cost table for each task.
- Cheddar allows simulating the scheduling of tasks.
- Cheddar allows recording different event raised during the simulation. The figure below is the list of event can be raised during the simulation. When the task is pre-empted, the event Context-Switch is raised.

End of task capacity
Write to buffer
Read from buffer
Context switch overhead
Running task
Task activation
Allocate resource
Release resource
Wait for resource
Send message
Receive message
Wait for memory
Address space activation

Table 4.2: Table of events in Cheddar

To sum up, the process can be described as follow: we model task with information about assembly instruction. Then, we perform per-task analysis based on the task model and useful cache block analysis method. Result in pre-emption cost table for each task. The simulation is done by using Cheddar scheduling simulator. When the event Context-switch is raised (task is pre-empted), lookup the corresponding cache related pre-emption delay in the cost table and add the delay to task's capacity.

4.6 Conclusion

In conclusion, the solution for the problem can be summarized in four main steps. Firstly, we add the model of the cache and reuse the existing hardware model in Cheddar framework. Secondly, an abstraction of a program's control flow graph is modelled as a set of basic blocks containing information about assembly instruction and memory mapping. After hardware and software architectures are modelled, we apply the analysis method useful cache block [13]. In the final step, the Cheddar scheduling simulator is used to perform simulation.

5

Evaluation

Evaluation for the work is done by using a case study and experiments. In the case study, we present a simple example to demonstrate the steps of the method and how can we have useful block for instruction cache. In the experiment, we choose will examine a simple program from the Malardaren benchmark suite [15]. The program control flow graph, assembly instructions code is generated by using the tool aIT[1].

The limitation of the evaluation is the lack of comparison with other real time scheduling analyser. In our research, we did not find a real time scheduling tool that take the cache and its cache related pre-emption delay in to account. The approach is that: firstly, we take a simple example to explain the algorithm and. Secondly, we take a program that is analysed in previous research on extrinsic interference analysis, perform our analysis and compare the result.

5.1 Case study

The configuration of the case study is:

- Cache: Direct-mapped instruction cache with 16 cache blocks.
- Task: 7 basic blocks, each basic block consumes 10 cache blocks for their instruction, offset of the first basic block in the memory is 0 and offset is increased by 10 for each basic blocks from 2 to 7. The flow graph of the task:

[1] www.absint.com

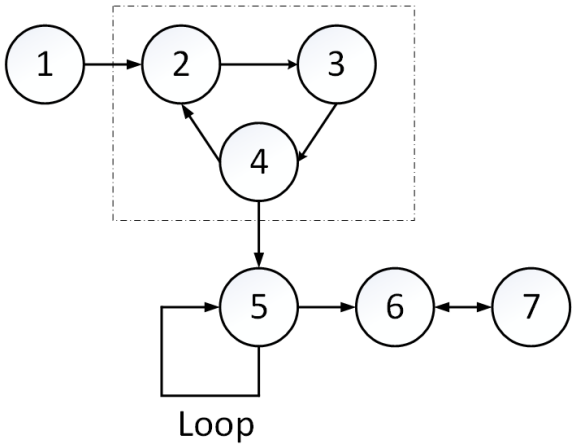

Figure 5.1: Task's flow graph

The memory and cache mapping is done by using the formula:

$$(Block\ address)\ MOD\ (Number\ of\ blocks\ in\ cache)$$

For example:

Basic block 1

Memory Address	0	1	2	3	4	5	6	7	8	9
Cache Address	0	1	2	3	4	5	6	7	8	9

Basic block 2:

Memory Address	10	11	12	13	14	15	16	17	18	19
Cache Address	10	11	12	13	14	15	0	1	2	3

Basic block 3:

Memory Address	20	21	22	23	24	25	26	27	28	29
Cache Address	4	5	6	7	8	9	10	11	12	13

Basic block 4:

Memory Address	30	31	32	33	34	35	36	37	38	39
Cache Address	14	15	0	1	2	3	4	5	6	7

The mapping for other basic blocks is done same above. It is important to mention that in this case, the size of the cache is larger than the capacity of instruction for each basic block. As a result, the $gen^c[B]$ for Reaching Memory Block (RMB) and Live Memory Block (LMB) are the same.

We can see that the basic block 2 can contain memory blocks that is useful for basic block 3 and 4 in the second round of execution. The table belows is the $RMB^c_{OUT}[2]$ or the RMB of basic block 3.

c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$RMB^c_{OUT}[2]$	16	17	18	19	4 36	5 37	6 38	7 39	8 **24**	9 **25**	10	11	12	13	14	15

The next table shows the $LMB^c_{IN}[3]$ or the LMB of the basic block 3.

c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$LMB^c_{IN}[3]$	32	33	34	35	20	21	22	23	**24**	**25**	26	27	28	29	30	31

By doing the intersection between RMB and LMB of the basic block 3, we can conclude that there are 2 useful cache blocks.

5.2 Experiments

The configuration of the case study:

- Processor: LEON SPARC v3 processor. Each instruction is encoded by 32 bits/4 bytes.
- Compiler: Bare-C Cross-Compiler System for LEON3/4 gcc-3.4.4 and gcc 4.4.2.
- Cache: 1 KB Direct-mapped instruction cache with 8 bytes block-size. The memory and cache mapping is done by using the formula: $(Block\ address)\ MOD\ (Number\ of\ blocks\ in\ cache)$
- Task: the programs from Malardaren benchmark suite are used.
  - bs.c: a binary search for an array of 15 integer elements. Program's total number of assembly instructions is 47.

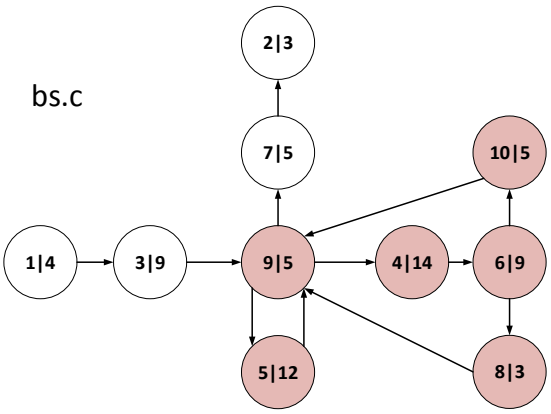

Figure 5.2: bs.c's flow graph

  - fac.c: program calculates the faculty function. Program's total number of assembly instructions is 69.
  - insertsort.c: insertion sort on a reversed array of size 10. Program's total number of assembly instructions is 52.
  - fibcall.c: simple iterative Fibonacci calculation. Program's total number of assembly instructions is 52.
  - fdct.c: Fast Discrete Cosine Transform. Program's total number of assembly instructions is 677.

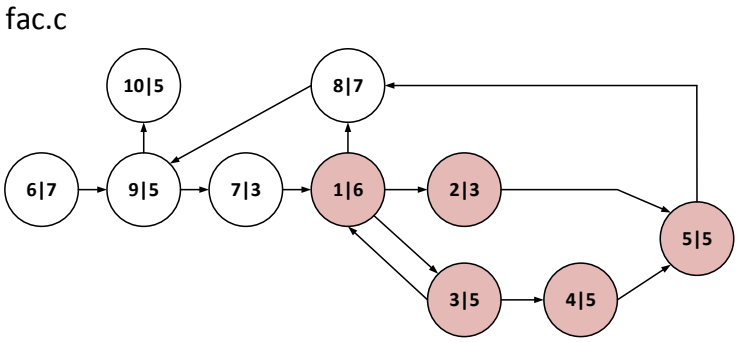

Figure 5.3: fac.c's flow graph

The figure 5.2 and figure 5.3 is the interpretation of the control flow graph of the tasks. Each circle represents a basic block. The first number in the circle is the id of the basic block. It is also the position of the assembly instruction block. Output of the analysis:

	Number of UCBs				
Basic Block	bs.c	fac.c	fdct.c	fibcall.c	insertsort.c
1	0	18	2	3	0
2	0	18	91	12	1
3	1	18	94	12	21
4	26	18	2	12	21
5	26	18	47	1	0
6	26	1	50	2	0
7	1	18	0	0	0
8	26	18	1	-	-
9	26	18	9	-	-
10	26	0	-	-	-

Table 5.1: Analysis result.

We can compare the accuracy of the analysis method with the result on [16] (on the work, the authors only give the upper bound of the UCB and the processor is ARM processor). Our implementation returns slightly different values (+/-1), we can assume that it is the because of the different in instruction set between two processors (LEON v3 and ARM). From the result, we can have some remarks. The number of useful cache block is high on basic blocks within a loop. As the result, we can consider a loop as one basic block. This helps simplify the model of the control flow graph, we can consider one loop as one basic block which all the cache blocks used are useful.

5.3 Conclusion

An simulation was performed to verify the correctness of the analysis method with the case study. It is possible to perform simulation with different software/hardware architectures, scheduling analysis algorithms and produce correct cache related pre-emption delay for a simple case study. However, we did not perform experiments on a real program with complex control flow graphs. We targeted to complete the work latter.

The problem is that for a complex program the control flow graph is also complex. There are existing tools such as Control Flow Graph generator for Eclipse, CoFlo and PyPy, which support generator control flow graph from program source code to textual, graphical presentation. However, it is a difficult to transform the result to appropriate software architecture of Cheddar.

6

Discussion and conclusion

In this chapter, the work is concluded and future plan is presented. The problem statement is how to handle cache in real-time scheduling simulation. In section 6.2, main outcomes of the project is presented. Limitation of the work and possible future extensions are described in section 6.3 and 6.4, respectively.

6.1 Problem statement

The work is dealing with the problem of handling cache memory in real time scheduling simulator. The unpredictability limits the use of cache in real-time system. Current scheduling analysis and simulator tools such as STORM[11], YARTISS[12], FORTAS[17] and also Cheddar[1] are not addressing the cache memory in their architecture models.

6.2 Project Outcomes

- We added the cache model and task flow graph model in Cheddar. The cache model is stable for a basic cache architecture. We also include the cache coherence protocol attribute, which is not used in the analysis method at the moment. The task model is extended based on the control flow graph model. There are attributes added to model the cache behaviours.
- We added the effect of cache related pre-emption delay to the scheduling simulation. In fact, the effect of cache related pre-emption delay can be calculated at many different level of abstractions and use different methods thanks to the implementation of the cost table. The idea is try to applying various cache related pre-emption delay analysis method to the simulator, not only one analysis method.
- The analysis applied to: Direct Mapped Instruction Cache. It is the simplest cache model to analysis. The popular cache model in modern processor is set-associativity cache.

6.3 Limitation

- We are limited in cache type, fetch and replacement algorithm at the moment. At the moment, the analysis method is limited, however, the implementation is extensible. With different analysis methods, we can simply change the value of cost table and add necessary attributes to the basic blocks models.
- Pre-analysis steps are required to obtain information for the basic block. The basic block is a node in the Control Flow Graph of a program. Normally, the control flow graph (CFG) is generated from the compiler, however, we are looking for a solution to synthesize the in XML format.

6.4 Future works

- The first future work is to apply different analysis methods to calculate the cache related pre-emption delay. Moreover, the analysis methods should be used in both simulation test and feasibility test. In the near future, we would be able to to extend the useful cache block method to set associativity cache and implement the method evicting cache block[18].
- Secondly, it is to implement/integrate program source code analysis to support the method if possible. There are existing tool to model the control flow graph of a program. However, the ability to exchanging output in order to integrate the tool in a project is not addressed.
- We plan to extend the research to higher levels of memory. The simulation will be done not only with cache architecture but also with RAM and hard disk. In addition, we want to see how to handle shared cache analysis in real-time system with multi-core processor. The work in [19] has provided ideas about timing analysis of different processes running on multiple processor with a shared instruction caches.

Bibliography

[1] F. Singhoff, A. Plantec, P. Dissaux, J. Legrand, Investigating the usability of real-time scheduling theory with the cheddar project, Real-Time Systems 43 (3) (2009) 259–295.

[2] A. Plantec, F. Singhoff, Refactoring of an ada 95 library with a meta case tool, ACM SIGAda Ada Letters 26 (3) (2006) 61–70.

[3] J. L. Hennessy, D. A. Patterson, Computer architecture: a quantitative approach, Morgan Kaufmann, 2011.

[4] F. Sebek, The state of the art in cache memories and real-time systems, Tech. rep. (2001).

[5] Y.-T. S. Li, S. Malik, A. Wolfe, Performance estimation of embedded software with instruction cache modeling, in: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design, IEEE Computer Society, 1995, pp. 380–387.

[6] J. Rawat, Static analysis of cache performance for real-time programming, Master's thesis, Iowa State University.

[7] C. Ferdinand, R. Wilhelm, Fast and efficient cache behavior prediction.

[8] F. Mueller, D. B. Whalley, M. Harmon, Predicting instruction cache behavior, in: Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems, Citeseer, 1994.

[9] F. Sebek, The state of the art in cache memories and real-time systems.

[10] M. González Harbour, J. Gutiérrez García, J. Palencia Gutiérrez, J. Drake Moyano, Mast: Modeling and analysis suite for real time applications, in: Real-Time Systems, 13th Euromicro Conference on, 2001., IEEE, 2001, pp. 125–134.

[11] R. Urunuela, A. Deplanche, Y. Trinquet, Storm a simulation tool for real-time multiprocessor scheduling evaluation, in: Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on, IEEE, 2010, pp. 1–8.

[12] Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet, M. Qamhieh, et al., Yartiss: A tool to visualize, test, compare and evaluate real-time scheduling algorithms, proceedings of WATERS 2012.

[13] C.-G. Lee, H. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, C. S. Kim, Analysis of cache-related preemption delay in fixed-priority preemptive scheduling, Computers, IEEE Transactions on 47 (6) (1998) 700–713.

[14] A. Plantec, V. Ribaud, et al., Platypus: A step-based integration framework, 14th Interdisciplinary Information Management Talks (IDIMT-2006) (2006) 261–274.

[15] J. Gustafsson, A. Betts, A. Ermedahl, B. Lisper, The Mälardalen WCET benchmarks – past, present and future, OCG, Brussels, Belgium, 2010, pp. 137–147.

[16] S. Altmeyer, C. Maiza Burguière, Cache-related preemption delay via useful cache blocks: Survey and redefinition, Journal of Systems Architecture 57 (7) (2011) 707–719.

[17] P. Courbin, L. George, Fortas: Framework for real-time analysis and simulation.

[18] H. Tomiyama, N. D. Dutt, Program path analysis to bound cache-related preemption delay in preemptive real-time systems, in: Proceedings of the eighth international workshop on Hardware/software codesign, ACM, 2000, pp. 67–71.

[19] Y. Li, V. Suhendra, Y. Liang, T. Mitra, A. Roychoudhury, Timing analysis of concurrent programs running on shared cache multi-cores, in: Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE, IEEE, 2009, pp. 57–67.

Appendices

A

Appendix A: Programming Tools and Ada Implementation

A.1 Programming Tools

A.1.1 Platypus

Platypus [14] is a STEP-based meta-environment. The tool is used for meta-models and code generation. Platypus run on the Pharo - an open-source Smalltalk-inspired environment.

A.1.2 The GNAT Programming Studio

GNAT Programming Studio (GPS) is an integrated development environment for Ada developed by AdaCore. The tool is available for download at:

http://libre.adacore.com/tools/gps/

A.2 Ada Implementation

The implementation of the work in the project, at the moment the report is written, can be found at the address below. This is inside the cache branch of the Cheddar repository.

http://beru.univ-brest.fr/svn/CHEDDAR/branches/caches/src/framework/cache/

B

Appendix B: Publication and Presentations

International conferences

- *Scheduling Analysis from Architectural Models of Embedded Multi-Processor Systems.* Stéphane Rubini, Christian Fotsing, Frank Singhoff, Hai Nam Tran and Pierre Dissaux. EWiLi'13, The 3rd Embedded Operating Systems Workshop 26 - 27 August 2013, ENSEEIHT, Toulouse, France

Talk

- *Handling cache in real time system simulation.* Hai Nam Tran, SAPIENT meeting. EGIDE/Campus-France PESSOA project n 27380SA, July 2013.

Submitted papers

- *The SMART project: Multi-agent scheduling simulation of real-time architectures.* P. Dissaux, O. Marc, S. Rubini, C. Fotsing, V. Gaudel, F. Singhoff, A. Plantec, Vuong Nguyen Hong, Hai Nam Tran. ERTS conference. Toulouse. September 2014.