Instruction cache in hard real-time systems: modeling and integration in scheduling analysis tools with AADL

Hai Nam Tran, Frank Singhoff, Stéphane Rubini, Jalil Boukhobza
Univ. Bretagne Occidentale, UMR 6285, Lab-STICC, F-29200 Brest, France
Email:{hai-nam.tran,singhoff,rubini,boukhobza}@univ-brest.fr

***Abstract*—Cache prediction for real-time systems in a pre-emptive scheduling context is still an open issue despite its practical importance. In this paper, we propose a modeling approach for taking into account the cache memory in real-time scheduling analysis. The goal is to have a simple but practical implementation to handle the cache memory with a real-time scheduling analyzer. The proposed contribution consists of three main parts: (1) modeling the targeted system with the Architecture Analysis and Design Language (AADL), (2) applying the cache analysis methods in a real time scheduling analysis tool and (3) performing scheduling simulation to access schedulability. For such a purpose, we present an extension of both the scheduling analysis tool *Cheddar* and of the AADL modeling language in order to integrate the cache modeling and analysis methodology we proposed.**

Experiments are presented to illustrate our propositions. They provide results on analysis that show examples of the timing impact of task preemption as well as the increase in overall responses time of the task set. This impact is important and the developed tool provides means to precisely assess it.

I. Introduction

As the gap between processor speed and memory is continuously growing, making benefit of processor caches becomes crucial to absorb this performance gap. Despite the impressive performance in reducing the average memory access time, those caches hit a wall when it comes to optimize real-time systems. Indeed, cache behavior highly depends on the execution flows and data access pattern. The democratization of contemporary processors with large size and multi levels of cache in real-time systems motivates the proposition of verification methods that are able to handle this hardware component.

One of the important verifications performed for real-time systems is schedulability analysis. Schedulability analysis provides some means to prove that timing constraints of the tasks composing the software side of the system are satisfied. To study the schedulability, many assumptions are usually made in order to simplify the analysis. One of them is the preemption cost, which is most of the time assumed to be negligible. Indeed, classical task models used in real-time scheduling analysis only take into account the *worst-case execution time* (WCET) of the task - to define the *capacity* of the task. In practice, preemptions cause additional cost which can be substantial to task's execution time [1].

Integrating a processor cache memory component in a real-time system adds some costs to the task preemption. When a task is preempted, some memory blocks belonging to the task may be removed from the cache. Once the task resumes, it needs to reload previously removed memory blocks.

The substantial performance gain induced by cache comes at the expense of an increase in the system's unpredictability. Cache can help to reduce the *capacity* of a task; however, the inter-task preemption introduces the *cache related pre-emption delay* (CRPD) - the additional time to refill the cache with the cache blocks evicted by the preemption, thus increasing the variability of response time of task.

Even though there are many research projects about cache memory in real-time systems, such as the works in [2], [3], [4], [5] and [6], there is still a lack of practical implementation of those analysis methods in real-time scheduling analysis tools. In addition, a model based approach, which allows an abstraction of cache components, is necessary to combine the analysis tool with a system design and verification tool chain.

The contribution of our work is a real-time scheduling analysis tool, which supports verifying systems with cache memory. In this article, we investigate how an existing scheduling analysis tool such as Cheddar could be updated in order to take processor caches into account. For such a purpose and because Cheddar allows modeling the studied system thanks to the architecture description language AADL [7], we propose an extension of the AADL language to model cache memory. The CRPD is delivered based on the analysis methods in [2] and [3]. At the first steps, we only consider instruction cache. The purpose is to compare the CRPD to the WCET of task and to evaluate its impacts in real-time scheduling analysis.

The rest of the paper is organized as follows: Section II introduces the context of our work. Section III presents theoretical foundation and Section IV explains the implementation. Experiments are discussed in Section V. Section VI reviews related works. Section VII concludes the paper and presents possible future works.

II. Background

A. Cache analysis methods

The real-time scheduling for uniprocessor was well studied since the seminal paper of Liu and Layland [8]. However, the problem related to preemption is less addressed.

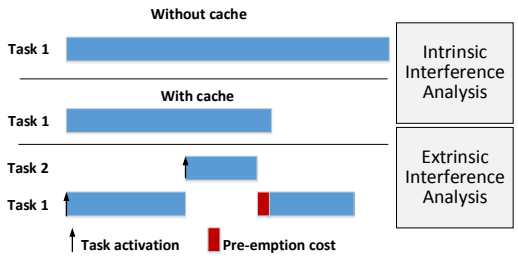

Fig. 1: Applying intrinsic and extrinsic interference analysis.

Cache analysis methods can be classified into two domains: *intrinsic* (intra-task) *interference* and *extrinsic* (inter-task) *interference* [9]

Intrinsic interference is the removal of a task's instructions or data in the cache by itself. Because the size of the cache is limited, memory blocks of a program may compete and collide with each other. The impact of *intrinsic interference* can be evaluated solely based on the task.

Extrinsic interference is the removal of a task's instruction or data in the cache by the effect of preemption by a higher priority task in preemptive scheduling context. The preempting task replaces data or instructions of the preempted task in the cache. The side effect of it is the CRPD.

Figure 1 is an illustration of *intrinsic* and *extrinsic* interference analysis. While *intrinsic interference* analysis methods focuses on lowering the bound of WCET, the *extrinsic interference* analysis methods focus on how to measure the impact of CRPD when several tasks are preemptively scheduled altogether.

A classical method to avoid extrinsic interference is cache partitioning [10] - dividing the cache into disjoint subsets where each program or processor core uses one. This method requires special hardware, software configuration and static allocation of cache blocks.

Another approach consists in bounding the CRPD. Two analysis methods exists for such a purpose: the *useful cache block* (or UCB) [2], and the *evicting cache block* (or ECB) [3]. Further works in [4], [5] and [6] improve those methods and give a more precise CRPD upper-bound.

The focus of our work is on *extrinsic interference* analysis. A part of it is the implementation of analysis methods in [2] and [3].

B. The Cheddar scheduling analyser

The work is performed in the context of Cheddar project [11]. Cheddar is an open source real-time scheduling analysis tool. It implements classical feasibility tests and scheduling algorithms for real-time systems. Users can model the system architecture by using a GUI tool provided by Cheddar. Then, feasibility tests and scheduling simulations could be run. Cheddar provides graphical representation of the result and a support to export the results in XML format.

Fig. 2: Cheddar and analysis tool chain: AADL models are designed and are verified by AADL Inspector. They are translated to Cheddar-ADL to perform scheduling analysis with Cheddar.

System architectures are defined with Cheddar Architecture Description Language (Cheddar-ADL). The Cheddar-ADL meta-model is specified with the modeling language EXPRESS [12]. The Cheddar meta-model defines hardware components such as: *processor*, *core* and *shared_resource*; and software components such as: *task* and *task_group*.

Cheddar-ADL can be updated to modify existing components or to add a new one. Cheddar class files are automatically generated from the Cheddar-ADL meta-model by the tool Platypus through a model-driving process. Platypus is a tool, which supports modeling in EXPRESS modeling language, model verification and code generation [12].

Although Cheddar has its own editor, it is not expected that users specify a complete architecture model with Cheddar. Indeed, Cheddar-ADL only focus on scheduling analysis point of a system. It is expected that users design their system with standard ADLs such as MARTE [13] or AADL. AADL or MARTE models are then translated to Cheddar-ADL for schedulability verifications. For example, Cheddar is used as a module in the industrial tool AADLInspector [14]. In this context, AADLInspector is responsible for both syntax and semantics checks of the AADL models provided by users before translating them into Cheddar-ADL prior calling verifications features of Cheddar, as shown in Figure 2.

C. AADL

The Architecture Analysis & Design Language (AADL) is a SAE standard, first published in 2004 [7]. The purpose of the AADL is to provide a standard for the modeling of the architecture of embedded real-time systems, in various domains such as avionic or automotive systems. It allows designers to perform various analysis, code generation, and other development activities.

AADL is a modeling language for description and analysis in terms of components and their interactions. It allows the modeling of software and execution platform components.

An AADL component is defined by a declaration section and implementations. Each component relies on a category. Categories of components are related to software entities, like *process*, *thread* or *data*, and hardware entities like *processor*, *memory*, *bus* or *device*. Each component may have several attributes called *properties*. The AADL standard includes a large set of pre-declared properties to model system characteristics. Moreover, new properties can be defined to precisely describe the expected system.

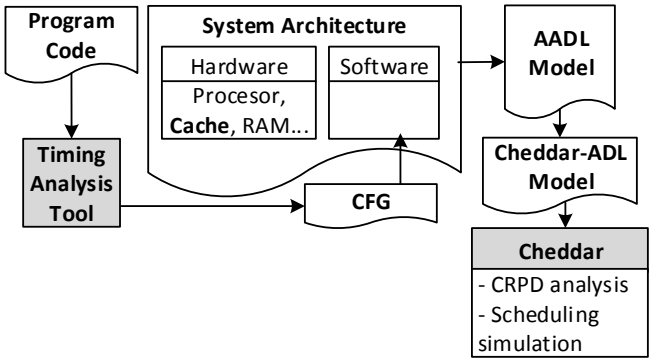

Fig. 3: Analysis Flow.

Below, we have an example of AADL components. Those are hardware components. In this example, a *memory* component and a *processor* component are defined. The memory component is a subcomponent of the processor. The size of the memory is 1 KB. The processor is a Leon V3, the hardware description is stored in a separate VHDL file.

```
memory Mem_ROM
  properties
    Memory_Size => 1 KBytes;
    Read_Time => (Fixed    => 0 ns..0 ns;
                  PerByte => 40ns..40 ns;);
end Mem_ROM;

processor Leon
end Leon;
processor implementation Leon.V3
   subcomponents
     Membank: memory Mem_ROM;
  properties
     Source_Text => "leonV3.vhdl";
     Source_Language => VHDL;
end Leon.V3;
```

Caches can be modeled with AADL *memory* components. However, the AADL *memory* component does not have attributes to model precisely a cache memory. In this article, we propose new attributes for such a purpose. In addition, to perform cache analysis, we need to capture the memory behavior of the program. We extend the *subprogram* component to model the control flow graph.

D. Overview of our approach

Our approach focuses on scheduling analysis of monoprocessor systems with shared cache memory. First, we investigate how to model the cache and program's memory accesses. Those components are modeled with AADL. Second, we apply the modeling approach and implement the CRPD analysis methods in the Cheddar tool. Finally, we perform scheduling simulation with the Cheddar scheduling simulator

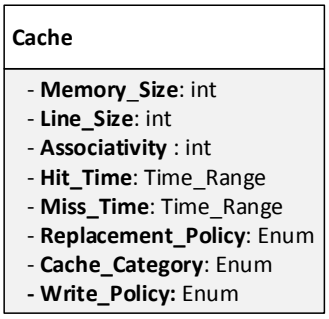

Fig. 4: Cache model.

to investigate the impact of CRPD. We aim to perform our analysis after code-generation context of a system design and verification tool chain. Program's memory accesses are deduced from the control flow graph generated by a timing analysis tool. Figure 3 shows an overview of the flow.

III. Model caches and analysis methods

Determining the cache access profile - program's memory accesses - is the main challenge when integrating cache analysis in real-time scheduling analysis tool. The WCET of a task is often modeled using only one attribute, the *capacity*. This is not enough to perform scheduling analysis with cache. Even though there are many researches on bounding WCET of tasks with cache, they are integrated in very few real-time scheduling analysis tool. In this section, we present entities and attributes we need to model the cache memory and program's memory accesses. Then, two classical basic analysis methods for CRPD are presented.

A. Modeling the cache

Starting from a simple cache architecture, we aim to analyze the CRPD on direct-mapped instruction cache. Compared to data caches, the access pattern for instruction cache is sequential and can be analyzed based on the control flow graph of a program.

In this section, basic attributes of cache are presented (see figure 4).

The *Memory_Size* attribute is the number of bytes which a cache can contain.

The smallest piece of data, which a cache can exchange with the higher levels of memory hierarchy, is a cache line. When a cache miss occurs, a cache does not only load the required bytes, but it loads also a block of data to utilize the spatial locality. The *Line_Size* attribute specify the size of a cache line.

When accessing a data block, if this block is already in the cache, we have a cache hit. If not, we have a cache miss. A data block from main memory is loaded into the cache when a cache miss occurs. *Miss_Time* and *Hit_Time* are two attributes used to model the access time in those two cases.

The memory-to-cache mapping policy is specified by the *Associativity* attribute. Associativity gives the number of cache

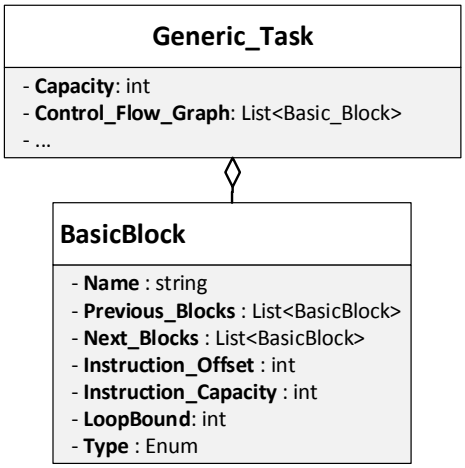

Fig. 5: Control flow graph model.

locations where a memory block can reside. If associativity equals one, we say that we have a *direct-mapped* cache. If associativity equals the number of cache blocks, we have a *fully-associative* cache. In other cases, we have a *n-way set associativity* cache, n being the number of possible locations for a single block.

When associativity is greater than one, a *replacement policy* is needed. Popular replacement policies for cache are first in, first out (*FIFO*), least recently used (*LRU*) and *pseudo LRU*.

A cache memory can be classified into three categories: *data cache*, *instruction cache* or *combined* one, specified by the *Cache_Category* attribute.

Finally,*Write_Policy* decides how the data in the cache are written back to higher level memory.

B. Modeling the control flow graph

Information about program control flow graph (CFG) is a requirement to obtain its memory accesses.

Definition 1 (Control flow graph): a representation, using graph notation, of all paths that might be traversed through a program during its execution. Each node of this graph is called a basic block - a sequence of consecutive instructions in which flow of control enter at the beginning and leaves at the end without halt or branching.

In our work, we use an abstract level of the CFG, modeled as a set of basic blocks, as shown in figure 5. We can extract such program's CFG from executable file by using tools such as aiT [15]. We only model information to locate the position of each basic block in the memory

A set of basic blocks representing a CFG is added as an attribute of each task besides the classical *capacity* attribute.

The two attributes *Previous_Blocks* and *Next_Blocks* specify the positions of the basic block in the CFG of the task.

The attribute *Instruction_Offset* indicates the position of the basic block in the main memory. *Instruction_Capacity* is the number of assembly instructions of the basic block.

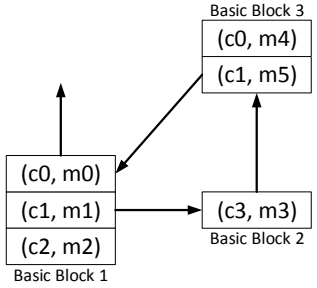

Fig. 6: UCB Example.

Finally, the *Loop_Bound* attribute specifies how many times a basic block can be executed.

C. Analysis methods

In this section, we present the two basic approaches for cache extrinsic interference analysis that we take into account in this work.

1) Useful Cache Block: UCB analysis method has been introduced in the work of [2]. The idea consists in performing static analysis on a task's CFG. An abstract state of the cache utilization on each basic block is kept. As we handle only instruction cache, it is possible to evaluate which memory blocks could be in the cache after the execution of each basic block and which memory blocks could be needed in the future. UCB can be defined as follows:

Definition 2 (Useful Cache Block): A memory block m is called a useful cache block at program point P, if m may be cached at P and m may be reused at program point P' after P that may be reached from P without eviction of m on this path.

Figure 6 illustrate UCB analysis. We have three basic blocks representing a small loop. The cache is a direct-mapped cache with four cache lines. After the first loop iteration (end of execution of the basic block 3), the mapping in the cache is described in Table I. As we can see, the memory block *m2* and *m3* are still in the cache and will be reused in the second loop. The cache lines c2 and c3 are called UCBs. The idea of the analysis method is to analyse all the basic blocks in the CFG and find how many cache blocks *can be* UCB in each basic blocks.

TABLE I: Cache mapping

Cache Block	c0	c1	c2	c3
Memory Block	m4	m5	m2	m3

By performing analysis on program's CFG, we can compute the number of UCBs for each basic block. The algorithm of the analysis method includes two data flow equations which can be solved by using an iterative approach. More details on this algorithm could be found in [2]. The complexity of the algorithm is $O(|V||E||C|)$, where V is the set of nodes (basic

blocks) of the flow graph, E is the set of (directed) edges of the flow graph and C is the set of cache blocks used in the program.

By multiplying the number of UCBs of one basic block with the cache miss time, we obtain the CRPD when a program is preempted at this basic block. Then, the CRPDs of all basic blocks are sorted in a descending order. If a basic block is in a loop - it can be executed several times, the CRPD of this basic block is duplicated. Those values constitute a cost table as follows:

TABLE II: Cost table

# of Preemption	1	2	...	n
CRPD	γ_1	γ_2	...	γ_n

The table describes the CRPD of the n-th times the task is preempted. The first row is the n-th times the task is preempted and the second row is the CRPD when the preemption occurs.

2) Evicting Cache Block: A second method to evaluate the CRPD is based on ECB. This method has been proposed by [3]. An ECB is defined as follows:

Definition 3 (Evicting cache block): A memory block of the preempting task is called an evicting cache block, if it may be accessed during the execution of the preempting task.

The bound of the CRPD is given by the number of cache blocks (of the preempted task) evicted by the preempting task. In this method, we focus on the pre-empting task. For a set of task, we need to know the location in the main memory of each task. By doing so, we identify which part in the cache of a task is evicted by another task.

The main difference between the two analysis methods is the information about the task. The UCB analysis requires information about the task itself. On the other hand, the ECB analysis does not require detailed information about the task but we need to know the task location in the main memory.

Our models capture the required information for those methods in order to perform the scheduling analysis and simulation.

IV. Implementation

In this section, we show how AADL has been extended to integrate theoretical models introduced in the previous section. Then, we present the implementation of those models in Cheddar-ADL. Finally, we explain how we have adapted the scheduling simulator of Cheddar.

A. Extending AADL to model cache memory and CFG

AADL was extended to model the cache memory and the CFG.

1) Modeling the cache with AADL: To model the cache, we extended the AADL *memory component*. According to AADL standard, memory component is an execution platform component that stores code and data binaries. Memory component has various properties including memory size, word size and read/write speed. However, it lacks properties to model cache memory's attributes such as *associativity*, *replacement policy* and *write_policy*. AADL allows adding user-defined properties to any component. Those user-defined properties can be grouped in property-sets. We then added a property set to model attributes of the cache memory as follows:

```
property set Cache is
  Line_Size: type aadlinteger;
  Hit_Time: type Time_Range;
  Miss_Time: type Time_Range;
  Associativity: type aadlinteger
    applies to (memory);
  Replacement_Policy: type enumeration
    (FIFO, LRU, PLRU, Random)
    applies to (memory);
  Cache_Category: type enumeration
     (Data_Cache, Instruction_Cache,
     Data_Instruction_Cache)
    applies to (memory);
  Write Policy: type enumeration
    (Write_Back, Write_Through
    Write_Through_Allocate)
    applies to (memory);
end Cache;
```

For others attributes of the cache, existing standard properties of the memory component could be reused. For example, the *Memory_Size* attributes could be used to model the *Cache_Size*.

The *Hit_Time* and *Miss_Time* of the cache could be derived from it and higher level memory components' *Read_Time*, *Write_Time* attributes if we have enough information about the processor's memory read, write operation.

The example below illustrates a direct-mapped (*Associativity* = 1) instruction cache. The *Line_Size* is 16 bytes and the *Cache_Size* is 1024 bytes. Cache *Replacement_Policy* is *Least Recently Used* (*LRU*).

```
memory Cache
  properties
    Memory_Size => 1024 bytes;
    Cache:Line_Size  => 16 bytes;
    Cache:Associativity => 1;
    Cache:Replacement_Policy => LRU;
    Cache:Cache_Category => Instruction;
    Cache:Hit_Time => 2 ns .. 4 ns;
    Cache:Miss_Time => 5 ns .. 10 ns;
end Cache;
```

2) Modeling the CFG with AADL: Two solutions exist to model the control flow graph with AADL. The AADL behaviour annex [16] can be used to model the CFG. A simpler solution is to store the control flow graph in a separate file and to annotate the AADL model with the name of this file. Then, we have defined a new property to store the CFG file name for each program. Each program is then modeled by a subprogram AADL component and its CFG can be stored with the following property:

```
cfg_source_file: type string
        applies to (subprogram);
```

B. Implementation in the Cheddar tool

In this section, we present the implementation of our work in the Cheddar tool.

1) Modeling the Cache with Cheddar-ADL: The Cheddar-ADL meta model is used to generate the source code of Cheddar to handle Cheddar-ADL entities. To generate the source code, we use the Platypus tools [12].

The Cheddar-ADL meta-model is specified with EXPRESS. In this meta-model, a hardware or a software component of Cheddar-ADL is modeled by an EXPRESS entity. EXPRESS entities have attributes, which allow us to model any property of the Cheddar-ADL components.

We have presented most of the classical attributes of a cache memory in section III-A.

From the *Generic_Cache* entity, Cheddar class files are generated. The example below is a part of the generated Ada 1995 code for the cache component. The code to handle the behaviors of the component needs to be written by the programmer.

```
type Generic_Cache is
  new Named_Object with
record
  memory_size : Natural;
  line_size : Natural;
  associativity : Natural;
  ...
end record;
```

2) Modeling the CFG with Cheddar-ADL: To model the control flow graph in the Cheddar-ADL meta-model, we updated the task model of Cheddar-ADL in order to add an attribute to reference the CFG. A part of the generated Ada code is written below.

```
type Generic_Task
  is new Named_Object with
record
  capacity: Natural;
  control_flow_graph:
      Basic_Blocks_Table;
  ...
end record;
```

The CFG is modeled as a set of basic blocks. Attributes of each basic block are presented in section III-B. The *capacity* attribute includes intrinsic effects while the CRPD is derived from the CFG.

3) Update the scheduling simulator: We update the scheduling simulator of Cheddar to be able to compute scheduling simulation with CRPD.

The scheduling simulation in Cheddar works as follows. First, a system architecture model including hardware/software components, schedulers and tasks is loaded. Then, the scheduling is computed by three successive steps: computing priority, inserting ready task into queues and electing task. The elected task will receive the processor for the next unit of time.

The scheduling simulator of Cheddar records different events raised during the simulation, such as task releases, shared resources lockings or unlockings, ... The result of the scheduling analysis is the set of events produced at simulation time. To achieve scheduling simulation with caches, we have extended the set of events Cheddar can produce. For each preemption, Cheddar produces an event that stores the CRPD.

V. Experiment and Discussion

In this section, we present experiments to illustrate the use of our tool. The first experiment is a CRPD analysis per task. We obtain the upper-bound CRPD for each task by using the UCB analysis method and compare it with the WCET of the task. The second experiment is a scheduling simulation, which reuses the result of the first one. It indicates the impact of CRPD in scheduling simulation with different scheduling algorithms.

A. CRPD analysis per task with UCB

In this experiment, we want to compare the value of CRPD to the WCET of the task. The programs used in the experiment are taken from Malardalen benchmark suite [17]. They are popular WCET benchmark programs, used to evaluate and compare different types of WCET analysis tools and methods.

We perform experiments with the LEON V3 processor, clock speed 400 MHz, with 1 KB instruction cache and 16 bytes line size. The cache miss time is 10 clock cycles. Data cache is disabled. Each instruction of LEON processor is encoded on 32 bits.

TABLE III

Program	# Basic Block	WCET 1 w/o cache (μs)	WCET 2 w/ cache (μs)	CRPD max (μs)	UCB
bs.c	12	6.1	4.5	0.35	14
fac.c	10	5.9	4.9	0.25	10
fdct.c	9	80.9	80.2	1.23	49
fibcall.c	11	8.1	4	0.18	7
insertsort.c	7	41.07	22.2	0.28	11
ns.c	17	545.3	273.3	0.50	20
prime.c	22	6.6	6.8	0.6	24

The result of the analysis is shown in the table III. The second column contains the number of basic blocks in the CFG of the program. The third and fourth columns are WCET of the programs without and with cache, respectively. The data is obtained by using the WCET analysis feature of the aiT tool. After the CFG is generated, we apply the analysis method to get the highest number of UCB for each program and deduce the upper-bound CRPD, which are displayed in column five and six.

We notice that tasks *bs.c* and *fibcall.c* in the Table III have WCETs with a difference of 0.5 μs but the CRPD of *bs.c* is 0.35 μs compare to 0.18 μs of *fibcall.c*. In some cases, it should be consider to reduce the overall response time of the system. It would be interesting to take the CRPD into account when assigning priority to tasks. For example, if the two tasks have similar WCET, assigning the higher priority to task which has bigger CRPD can avoid the task to be preempted. We can also take the CRPD into account when applying preemption threshold solution following the work of [18].

From this result, first, we can see the substantial reduce in WCET of tasks when the cache is enabled (except for *prime.c*). Second, we see that the impact of CRPD on task WCET should not be excluded. Our aim is to help the system designers utilizing the performance boost of cache while still keeping track of the impact of extrinsic interferences in a less pessimistic hypothesis such as complete cache reloading when preemptions occurs.

B. Experiment 2: CRPD and scheduling simulation

TABLE IV

Task	**Capacity** (μs)	**Period** (μs)
fibcall.c	4	30
bs.c	4.5	40
prime.c	6.8	50
insertsort.c	22.2	60

We bring this example to demonstrate the use of our tool in scheduling simulation. The task set above is configured to have a processor utilization - the fraction of processor time spent in the execution of the task set [8] - of 75%. We run the scheduling simulation in one hyperperiod - least common multiple of the tasks periods. The tasks' capacity and CRPD are inherited from the previous experiment. The result is displayed in the table below.

TABLE V

Algorithm	**CRPD** (μs) **(UCB only)**	**CRPD** (μs) **(Combined)**	**# Preemption**
RM	6.8	4.6	22
EDF	5.3	3.9	18

Table V shows the cumulated CRPD and the number of preemptions. The first column is the total CRPD when we only use the upper-bound CRPD delivered from UCB analysis. In the second column, the CRPD is calculated with the combination of UCB and ECB. We assume a memory-to-cache mapping scheme, which the instructions of tasks are located next to each other in the main memory.

In this experiment, we revise one of the advantages of Earliest Deadline First [8] (EDF) to Rate Monotonic [8] (RM) scheduling. It gives less preemptions [19]. In the simulation, our tool can indicate the number of overall increases in the worst case response time (WCRT). As a result, we can compare the impact of CRPD with different scheduling algorithms.

When performing scheduling simulation with RM, the overall increase in WCRT is 6.8 μs, and the number of preemption is 22 times. When we apply the EDF scheduling algorithm, the overall increase in WCRT is 5.3 μs and the number of preemptions is 18.

The evaluation for UCB could be combined with ECB to reduce the bound of CRPD. We assume that when a task is preempted by another task having intersection in the cache, it only needs to reload the cache blocks which are useful. It raises another idea to optimize memory-to-cache mapping to reduce the impact of CRPD. For direct-mapped cache, if tasks are located edge to edge in the main memory, their cache mappings tend to not overlap each others. Consequently, tasks, which only use a small proportion of cache, should be located near each other in the main memory.

The actual execution time of a task depends on which scheduling policy we use. It decides the number of preemptions, thus, changing the cumulative CRPDs. However, the CRPD creates a reverse impact on the scheduling policy because it contributes in many factors such as priority assignments and feasibility tests.

VI. Related works

In this section, we present several real-time scheduling analysis and WCET analysis tools. In the domain of real-time scheduling analysis, the support for cache and CRPD when performing scheduling analysis are not clearly specified in those tools.

MAST [20] is a modeling and analysis suite for real-time applications. The hardware component abstraction of MAST model is generic and it includes processing resources and shared resources. The shared resource component is not supposed to model a cache memory unit. However, MAST considers the overhead parameters of the components.

STORM [21] and YARTISS [22] are scheduling simulation tools for real-time multiprocessor architectures. However, the support for cache memory component is not defined. SymTA/S [23] and RTaW [24] (RealTime-at-Work) are model-based scheduling analysis tools, which target automotive electronics industry. The hardware components supported in those tools are specific in their domains (ECU, CAN and AFDX Networks).

SimSo [25] is a scheduling simulation tool. It supports cache sharing on multi-processor systems. It takes into account impact of the caches through statistical models and also the direct overheads such as context switches and scheduling decisions. The memory behavior of a program is modeled based on Stack Distance Profile - the distribution of the stack distances for all the memory accesses of a task, where a stack distance is by definition the number of unique cache lines accessed between two consecutive accesses to a same line [26].

Several WCET analysis tools allows cache analysis. SymTA/P [27], HEPTANE [28], Chronos [29] and aiT [15] are examples of them. The analysis of those tools are based

on program path analysis. The analysis requires information about program's CFG. Those WCET tools focus on the cache *intrinsic interference* analysis. The analysis result could be used as an input for a scheduling analysis tool, which focuses on *extrinsic interference* analysis domain.

Our approach tends to re-use information obtained from a timing analysis tool (CFG, WCET) to perform scheduling analysis with cache memory taken into account.

VII. Conclusion

In the article, we presented an approach to handle cache memory in real-time scheduling analysis. Even though cache memory is an essential component to overcome the speed gap between the processor and memory, it causes additional preemption delay and unpredictability, which limits the use of cache in real-time systems.

The work is proceeded in the context of Cheddar real-time scheduling analyzer and AADL. Our solution consists of three parts: modeling cache memory and program memory accesses, implementing the analysis methods and performing scheduling simulation. We extended Cheddar to be able to deals with cache and to help designers to use them into critical systems.

There are open problems we are aiming to address in the future. The first one concerns the refinement of other CRPD analysis methods. Several improvements have been proposed in [6] to reduce the upper-bound of the CRPD. Second, we plan to investigate the approaches to take the CRPD into account when performing feasibility tests following the work of [30]. Finally, we also plan to study priority assignment algorithms in order to minimize the impact of CRPD.

References

[1] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proceedings of the 2007 workshop on Experimental computer science*. ACM, 2007, p. 2.

[2] C.-G. Lee, H. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *Computers, IEEE Transactions on*, vol. 47, no. 6, pp. 700–713, 1998.

[3] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings, "Adding instruction cache effect to schedulability analysis of preemptive real-time systems," in *Real-Time Technology and Applications Symposium, 1996. Proceedings., 1996 IEEE*. IEEE, 1996, pp. 204–212.

[4] H. Tomiyama and 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.

[5] J. Staschulat and R. Ernst, "Scalable precision cache analysis for preemptive scheduling," in *ACM SIGPLAN Notices*, vol. 40, no. 7. ACM, 2005, pp. 157–165.

[6] S. Altmeyer and C. Maiza Burguière, "Cache-related preemption delay via useful cache blocks: Survey and redefinition," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 707–719, 2011.

[7] P. H. Feiler, B. Lewis, and S. Vestal, "The sae architecture analysis and design language (aadl) standard: A basis for model-based architecture-driven embedded systems engineering," in *Proceedings of the RTAS Workshop on Model-Driven Embedded Systems*, 2003, pp. 1–10.

[8] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[9] A. Agarwal, J. Hennessy, and M. Horowitz, "An analytical cache model," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 2, pp. 184–215, 1989.

[10] D. Thiébaut, H. S. Stone, and J. L. Wolf, "Improving disk cache hit-ratios through cache partitioning," *Computers, IEEE Transactions on*, vol. 41, no. 6, pp. 665–676, 1992.

[11] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: a flexible real time scheduling framework," in *ACM SIGAda Ada Letters*, vol. 24, no. 4. ACM, 2004, pp. 1–8.

[12] A. Plantec and F. Singhoff, "Refactoring of an ada 95 library with a meta case tool," in *ACM SIGAda Ada Letters*, vol. 26, no. 3. ACM, 2006, pp. 61–70.

[13] M. Faugere, T. Bourbeau, R. De Simone, and S. Gerard, "Marte: Also an uml profile for modeling aadl applications," in *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*. IEEE, 2007, pp. 359–364.

[14] P. Dissaux, O. Marc, S. Rubini, C. Fotsing, V. Gaudel, F. Singhoff, A. Plantec, V. Nguyen-Hong, and H. N. Tran, "The smart project: Multi-agent scheduling simulation of real-time architectures," *Proceedings of the ERTS 2014 conference*.

[15] "Absint inc." [Online]. Available: http://www.absint.com/ait

[16] P. Dissaux, J. Bodeveix, M. Filali, P. Gaufillet, and F. Vernadat, "Aadl behavioral annex," in *Proceedings of DASIA conference, Berlin*, 2006.

[17] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen wcet benchmarks: Past, present and future." in *WCET*, 2010, pp. 136–146.

[18] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on*. IEEE, 1999, pp. 328–335.

[19] G. C. Buttazzo, "Rate monotonic vs. edf: judgment day," *Real-Time Systems*, vol. 29, no. 1, pp. 5–26, 2005.

[20] M. González Harbour, J. Gutiérrez García, J. Palencia Gutiérrez, and 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.

[21] R. Urunuela, A. Deplanche, and 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.

[22] 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," in *Proceedings of the 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2012, pp. 21–26.

[23] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis–the symta/s approach," *IEE Proceedings-Computers and Digital Techniques*, vol. 152, no. 2, pp. 148–166, 2005.

[24] "Realtime-at-work." [Online]. Available: http://www.realtimeatwork.com/

[25] M. Chéramy, A.-M. Déplanche, P.-E. Hladik *et al.*, "Simulation of real-time multiprocessor scheduling with overheads," in *International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*, 2013.

[26] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems journal*, vol. 9, no. 2, pp. 78–117, 1970.

[27] J. Staschulat, S. Schliecker, and R. Ernst, "Scheduling analysis of real-time systems with precise modeling of cache related preemption delay," in *Euromicro Conference on Real-Time Systems (ECRTS)*, Palma de Mallorca, Spain, jul 2005.

[28] A. Colin and I. Puaut, "Worst-case timing analysis of the rtems real-time operating system," *Rapport N PI1277, IRISA, France*, 1999.

[29] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Science of Computer Programming*, vol. 69, no. 1, pp. 56–67, 2007.

[30] W. Lunniss, S. Altmeyer, C. Maiza, and R. I. Davis, "Integrating cache related pre-emption delay analysis into edf scheduling," *University of York, York, UK, Technical Report YCS-2012-478. Available from http://www-users. cs. york. ac. uk/˜ wlunniss*, 2012.