Technical Report

An Improved Preemption Delay Upper Bound for Floating Non-preemptive Region

José Marinho

Vincent Nélis

Stefan M. Petters

Isabelle Puaut

HURRAY-TR-120601

Version:

Date: 06-04-2012

An Improved Preemption Delay Upper Bound for Floating Non-preemptive Region

José Marinho, Vincent Nélis, Stefan M. Petters , Isabelle Puaut

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail:

http://www.hurray.isep.ipp.pt

Abstract

In embedded systems, the timing behaviour of the control mechanisms are sometimes of critical importance for the operational safety. These high criticality systems require strict compliance with the offline predicted task execution time. The execution of a task when subject to preemption may vary significantly in comparison to its non-preemptive execution. Hence, when preemptive scheduling is required to operate the workload, preemption delay estimation is of paramount importance. In this paper a preemption delay estimation method for floating non-preemptive scheduling policies is presented. This work builds on [1], extending the model and optimising it considerably. The preemption delay function is subject to a major tightness improvement, considering the WCET analysis context. Moreover more information is provided as well in the form of an extrinsic cache misses function, which enables the method to provide a solution in situations where the non-preemptive regions sizes are small. Finally experimental results from the implementation of the proposed solutions in Heptane are provided for real benchmarks which validate the significance of this work.

An Improved Preemption Delay Upper Bound for Floating Non-preemptive Region

José Manuel Marinho*, Vincent Nélis*, Stefan M. Petters*, Isabelle Puaut†
*CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Portugal
†University of Rennes 1, UEB, IRISA, Rennes, France
Email: {jmsm,nelis,smp}@isep.ipp.pt , Isabelle.Puaut@irisa.fr

***Abstract*—In embedded systems, the timing behaviour of the control mechanisms are sometimes of critical importance for the operational safety. These high criticality systems require strict compliance with the offline predicted task execution time. The execution of a task when subject to preemption may vary significantly in comparison to its non-preemptive execution. Hence, when preemptive scheduling is required to operate the workload, preemption delay estimation is of paramount importance. In this paper a preemption delay estimation method for floating non-preemptive scheduling policies is presented. This work builds on [1], extending the model and optimising it considerably. The preemption delay function is subject to a major tightness improvement, considering the WCET analysis context. Moreover more information is provided as well in the form of an extrinsic cache misses function, which enables the method to provide a solution in situations where the non-preemptive regions sizes are small. Finally experimental results from the implementation of the proposed solutions in Heptane are provided for real benchmarks which validate the significance of this work.**

I. Introduction

Embedded systems are ubiquitous and control most aspects of our everyday environment. Such systems which have additional timing constraints in the form of quality-of-service or hard real-time response requirements, are termed *real-time systems*. For real-time systems it is not only important to demonstrate that the system performs functionally as expected, but it needs to be shown that the temporal requirements are met as well.

Processors are composed of several subsystems (such as caches, pipelines, transfer lookaside buffers, etc.) which display, at any time-instant, an associated "*state*". All these processor subsystems quasi-continuously face state changes at run-time. In particular, it is the case for task preemptions: when a task resumes its execution (after being preempted), the cache(s) will display a state which is different from its state at the time the task got interrupted. Then, it is needed to reconstruct at least partially its working set after the task resumes execution. The reconstruction procedure is subject to time penalties. In real-time systems, where timeliness is an essential property of the system, these penalties need to be carefully evaluated to ensure that all deadlines are met.

In this work we will mainly focus on the cache-related preemption delay (CRPD), because this delay has the most important impact on the variation of the execution time of a preempted task [2]. Knowledge of preemption delays is crucial for the assessment of the timing behavior of task-sets when scheduled by a given real-time scheduling policy.

Real-time scheduling policies may be broken into three broad categories, with respect to how preemptions are handled: *(a) Non-preemptive scheduling*, where task preemptions are *not* allowed, *(b) Fully-preemptive scheduling*, where the highest priority active task always gets hold of the processor as soon as it arrives in the system (by interrupting the current executing task if needed), and *(c) Limited preemptive scheduling*, a hybrid solution between non- and fully-preemptive scheduling. This latter category can be itself divided into two subcategories: *fixed non-preemptive region scheduling*, where preemption points are hard-coded in the task's code and preemptions are allowed only when the execution of a task reaches one of these preemption points, and *floating non-preemptive region scheduling*. In the latter one, whenever a higher priority task is released, the currently running task starts to execute in a non-preemptive region. The length of this non-preemptive region is generally constant and defined offline for each task but might be subject to online extension as is presented in [3]. When the duration of the non-preemptive region elapses a preemption occurs. The task which is dispatched onto the processor is then the highest priority active task.

On the one hand the *floating* non-preemptive regions model is more flexible than the *fixed* one and does not require modifications in the applications. On the other hand *fixed* restricts the time instants at which the preemptions may take place, which makes it more predictable than the *floating non-preemptive* scheduling policies. These policies thus provide the system designer with more information about how the system will behave and decrease the pessimism involved in the analysis. It is important to state that the schedulability of these restricted preemption policies dominate over the fully preemptive ones [4]. The theory devised onwards assumes the scheduling using *floating non-preemptive regions* and proposes a new approach to safely but more tightly bound the preemption delay suffered by a task when compared to the state-of-the-art.

Even though the experimental results only look at the preemption delay caused by instruction caches, data caches and other subsystems may be integrated into the analysis, simply by construction the respective functions which describe the requests of each task dependent on time.

II. Related Work and Contribution

Our work presents an extension of the previously published result [1]. CRPD estimation has been a subject of wide study in the past. Several methods have been proposed that provide an off-line estimation based on static analysis, for this inter-task interference value.

Lee et al. presented one of the earliest works on CRPD estimation for instruction-caches [5] where the notion of the set of memory blocks that might have been used in the past and may be used in the future, termed useful cache blocks, was first introduced.

Since the assumption used in [5], that the value of CRPD throughout a control flow graph's basic block would remain constant, no longer holds for data caches a different approach had to be devised. Computation of the CRPD for data caches has been later proposed by Ramaprasad and Mueller [6].

Preemption delay estimation is of little value without its integration into the schedulability test of the systems. Since preemption delay is affected by all elements of the task-set several approaches exist to handle this situation. The scheduling analysis by Lee [5] is based on response-time analysis (RTA) by using the k highest values of preemption delay and incorporating that quantity into the response time of the task. Lee uses integer linear programming (ILP) to compute the preemption delay suffered by each task.

Busquets et al. also used RTA [7], but considered the maximum effect the preempted task may suffer by multiplying the number of preemptions with the maximum CRPD. While this is more pessimistic than Lee's approach, it removes the complex analysis of intersecting cache sets which for realistically sized programs suffers from heavy state explosion.

Also a less complex algorithm in comparison to Lee's resorting to RTA was presented by Petters and Färber [2]. Opposed to Busquets' approach Petters uses the knowledge of the maximum damage each preempting task may cause instead of only considering the worst-case preemption delay. The ILP problem is addressed by using an iterative algorithm. Staschulat and Ernst provided a method to estimate the CRPD for instruction cache in [8]. This area has been extended in several works of Altmeyer et al. [9].

Altmeyer et al. presents a summary of all of the literature so far relative to preemption delay on fully preemptive fixed task priority [10]. The authors also presented an enhancement to the available work by merging the approaches of Petters and Busquets in a safe way and considering jitter and the preemption delay suffered by the shared resource execution. A demand-bound function based procedure has been proposed by Ju et al. [11]. However, the general approach of computing the CRPD is similar to Lee's work.

All of the presented preemption delay-aware schedulability tests are specific to fully preemptive scheduling and are much more pessimistic than the one presented in this work since they do not consider the evolution of the preemption delay with the program progression of the preempted task. Our approach differs from past work in the sense that it ties the preemption delay with program-execution progression, thus enabling less pessimism in the preemption delay estimation.

Restricting preemption points presents a viable way to address the problem of preemption delay. The mechanism of preemption deferral was first proposed by Burns et al. [12]. It has a number of advantages as has been pointed out in several works e.g. [13], [14]. In particular, Gang Yao et al. provide a comparison of all available methods described so far in literature [14] regarding restricted preemptive scheduling using fixed task priority.

Bertogna and Baruah have devised a method to compute the size of the non-preemptive regions, for earliest deadline first (EDF) scheduling policy, using a demand-bound function based technique [4]. In this work the slack in the schedule depending on the length of the interval, assuming synchronous release of all the tasks, is computed. The method fits both the fixed non-preemptive region model and the floating one.

Several methods addressing the same issue in fixed task priority exist [15], [3]. A fixed priority scheduling method has been devised by Gang Yao et al. [15], where a maximum bound on the length of fixed non-preemptive regions is provided. In this situation the computed length of the fixed non-preemptive regions are generally larger than in previous work, as the last chunk of a task's execution is not subject to further preemptions. This enables a further reduction on the number of preemptions.

Marinho and Petters presented a method to increase, at run-time, the length of the preemption triggered floating non-preemptive regions for fixed task priority [3]. This method is taking advantage of off-line knowledge and on-line task release information to increase the length of the non-preemptive regions. This leads generally to a steep decrease on the preemptions suffered. Similar to previous work the preemption delay was not taken into account [3]. Reducing the number of preemptions helps decreasing the pessimism added to the schedulability test.

The preemption delay estimation problem using fixed non-preemptive region scheduling was presented by Bertogna et al. [16]. In order to reduce CRPD, the usage of fixed non-preemptive areas of code is proposed. The preemption points are thus reduced to a small number of well defined points. In this way the maximum CRPD is decreased and overall system's response time is enhanced. This work has the limitation that it requires manipulation of the code of tasks and thus is not an attractice option for system developers. In particular, it is not straightforward to take into account tasks with complex control flow graphs [16]. Additionally it can not be easily applied in situations where the task-sets are subject to run-time change, since the maximum allowed distance between preemption points is defined by the higher priority workload.

Our previous work [1] only uses the information on the preemption delay upper bound at any time t. In this work more information is extracted from the task code, so that the preemption delay estimation is decreased. The novel algorithm provided in this work also enables the analysis to be carried out in situations where the size of the non-preemptive regions is smaller than the maximum value of the preemption delay for the task. Moreover both methods were implemented in Heptane [17], and preemption delay estimations are provided from real benchmarks [18]. In the previous work [1] benchmark based experimental results were not available. Similarly to our previous work, we concentrate on instruction caches.

III. System Model

The system consists of a task set $\tau = \{\tau_1, \ldots, \tau_n\}$ scheduled to run on a single core processor. For each task τ_i, we assume that we have an estimation of its worst-case execution time (WCET) denoted by C_i.

Each task τ_i may generate a potentially infinite sequence of jobs, which are the entities that contend for the processor usage. The release of each consecutive job from τ_i is separated in time by at least T_i time units (sporadic task model). Every job of τ_i has to complete its workload D_i time units after it was released. It is assumed that T_i and D_i are independent (unrelated task-model).

There is an inherent priority relation between the jobs which governs the contention for the processor. This contention will be treated in a limited preemption model, which means that preemptions are allowed but are subject to some restrictions. This work supports both fixed task priority [15] and EDF [4] with floating non-preemptive region scheduling policies.

A floating non-preemptive region starts when a job is executing on the processor and a higher priority job is released. We denote by Q_i the length of the non-preemptive regions of task τ_i. This means that once a floating non-preemptive region has started, it will last for Q_i time units unless the currently running job completes before. Therefore, the preemption points which lead to the worst-case cumulative preemption delay are subject to the

constraint of being separated by *at least* Q_i time units. The first preemption can only happen after the task τ_i has completed Q_i units of execution. The Q_i value is a characteristic of each task. If the currently running job has not yet completed execution after the Q_i time units elapse then the highest priority job in the ready queue preempts it. The determination of Q_i can be performed by following the approaches determined by Bertogna and Baruah [4], Gang Yao et al. [15] or Marinho and Petters [3] and is assumed given within this work.

When a preempted task (say τ_i) resumes its execution, its remaining execution time will eventually increase, in comparison to the situation in which it was not preempted. This effect is due to the loss of working set in the hardware state. Within this work we focus on the largest contributor which is the CRPD. We call this increase in the remaining execution time the preemption *delay* that the task τ_i has to account for. This delay is as high as the amount of information, useful for the remaining execution of τ_i, evicted during the preemption.

The preemption delay varies during the execution of the job. This is illustrated with a simple example. Suppose that a task starts its execution by loading from the memory an important amount of data. Then the task processes all these data in a short period of time and finally, it performs a long-time computation using only a small subset of the data. In this case, the maximum preemption delay during the beginning of the task will be high, since in the worst-case scenario all the loaded data might be evicted during the preemption, hence forcing the task to reload them at the return from preemption. Once the data have been processed, the maximum preemption delay falls drastically, since a preemption during the long-time computation can only force the task to reload the few data elements that it needs when resuming its execution.

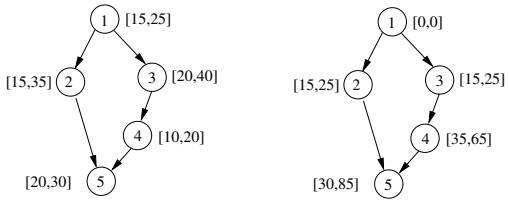
Each task may be characterised thus by a task-specific preemption delay function. As jobs execute their preemption delay varies with their progression through their execution. We model this varying cost of every task τ_i using a preemption delay function. As such, it displays, for any time-instant t where the function is defined, an *upper bound* on the preemption cost that the task would incur if it was preempted at time t. This function does not take into account the preemption delay that has to be paid in the post-preemption execution, as such it is only valid for the first preemption since it .

IV. Coupling Preemption Delay Cost With Execution Intervals

This section first focuses on determining the preemption delay function $f_i(t)$ of every task τ_i, that defines the maximum preemption delay when τ_i be preempted t time units after starting. The determination of $f_i(t)$ directly comes from [1]. The computation of $f_i(t)$ requires the CRPD when preempted at every basic block (BB_b) to be known (see § IV-A). Then the set of basic blocks that may be executed at time t has to be computed (see § IV-B). Function $f_i(t)$ can be computed as detailed in § IV-C. The concept of extrinsic cache miss, that will be used to tighten CRPD estimation, is introduced in § IV-D.

A. CRPD estimation

The CRPD when preempting a task at a given basic block is estimated using the method proposed by Negi et al in [19]. The method relies on the fixed-point computation, for every basic block of:



(a). CFG with Execution Time Intervals per BB

(b). CFG with Computed Earliest and Latest Start Offsets per BB

Fig. 1. Example of CFG for loop-free code. The CFG is composed of several basic blocks connected by directed edges that represent jumps in the code. Each basic block is a set of sequential instructions delimited by a jump. In the left part, intervals $[e_b^{min}, e_b^{max}]$ represent the minimum and maximum execution times of basic block b. In the right part, intervals $[s_b^{min}, s_b^{max}]$ represent the earliest and latest start time of every BB_b.

- Reaching Cache States (RCS). The Reaching Cache States at a basic block BB_b of a program, denoted as RCS_b, is the set of possible cache states when BB_b is reached via any incoming program path. This notion captures the possible cache content when the task is preempted at BB_b.
- Live Cache State (LCS). The LCS at BB_b, denoted as LCS_b, is the set of memory blocks that may be referenced in the future in any outgoing program path from BB_b. This notion captures the potential reuse of memory blocks after a preemption point occurring at BB_b.

Cache Utility Vectors (CUV) as defined in [19] are then computed, and correspond to the cache blocks that may be in the cache (in RCS) and may be reused (in LCS). As our method does not make any assumption on the higher priority tasks, no information on the task(s) that can preempt a given task is exploited. The CRPD at BB_b is then simply the delay to reload all the cache blocks in CUV_b, without considering cache usage in the preempting task(s) as done by Negi et al. [19] and Altmeyer et al. [20].

B. Computing execution intervals

Computing $f_i(t)$ for every task τ_i, represented by its control flow graph (see left part of Figure 1), requires to obtain, the interval of time during which every basic block b of τ_i might execute. The minimum and maximum execution time of every basic block b, noted respectively e_b^{min} and e_b^{max} have to be known. The cache analysis method used to classify every memory access uses the following categories:

- AH (Always Hit) when the access will always result in a hit
- AM (Always Miss) when the access will always result in a miss
- FM (First Miss) the access could neither be classified as hit or miss the first time it occurs but will result in cache hits afterwards (this category is used for code inside loops)
- NC (Not Classified) when no more precise categorization can be achieved

Given the classification of every reference, respective values of e_b^{min} and e_b^{max} are easily generated by considering the lower and higher delays allowed by the category (e.g. hit delay and miss delay for the category NC). The first iteration of every loop is virtually unrolled before applying the analysis in order to limit the number of references classified FM and NC.

Then, computing execution intervals on loop-free code requires to know for every basic block b its earliest and latest start offsets

s_b^{min} and s_b^{max}. This can be done by a breadth-first traversal of the CFG, applying to every traversed basic block b the following formulas:

$$s_1^{min} = s_1^{max} = 0 \quad (1)$$

$$s_b^{min} = \min_{x \in pred(b)} \{s_x^{min} + e_x^{min}\} \quad (2)$$

$$s_b^{max} = \max_{x \in pred(b)} \{s_x^{max} + e_x^{max}\} \quad (3)$$

with *pred(b)* the direct predecessor(s) of a basic block b in the CFG. Values for e_x^{min} and e_x^{max} can be produced by standard WCET estimation tools (variations between e_x^{min} and e_x^{max} come for example from memory references that can not be determined statically as hitting of missing the cache).

The right part of Figure 1 shows for every basic block its earliest and latest start offset after applying the above formulas. Then, the time interval within which every basic block b may execute is $[s_b^{min}, s_b^{max} + e_b^{max}]$.

The method was generalized to cope with natural loops and function calls as follows. Regarding loops, computation of execution time intervals is done on every loop starting from the innermost one, and then considering every loop as a single node with known earliest and latest start offsets. Similarly, in case of function calls, each function is analyzed for every call context, starting from the leaves in the acyclic call graph.

C. Computation of $f_i(t)$

Knowing the possible execution interval $[s_b^{min}, s_b^{max} + e_b^{max}]$ of every basic block b, the set of basic blocks that might execute at a given time instant t, noted $BB(t)$ is known. For each basic block b in this set, $f_i(t)$ can then be computed as follows, with $CRPD_b$ the CRPD paid when preempting the task at basic block BB_b:

$$f_i(t) = \max_{\forall b \in \mathrm{BB}(t)} \{\mathrm{CRPD}_b\}$$

D. Extrinsic Cache Miss Function

Until now the preemption delay is assumed to be paid immediately following a preemption [1]. By analysing the task code it is possible to extract information on when the preemption delay may be paid. This is captured through the concept of *Extrinsic Cache Miss*.

Definition 1 (Extrinsic Cache Miss)**.** *A memory access resulting in a cache miss due to the prior eviction of the requested cache line by code not belonging to the current task is termed extrinsic cache miss*

The extrinsic cache miss function $G_i(t)$ is an upper bound on the number of extrinsic cache misses a task might suffer in the interval $[0, t]$. $G_i(t)$ is then propostional to the upper-bound on the preemption delay that might have been paid in the same interval. As we will see, using the $G_i(t)$ information it is possible to provide preemption delay estimations in the presented framework for situations where $Q_i \leqslant \max_t(f_i(t))$, whereas in the previous work [1] this was not possible. This subsection is devoted to the explanation of the procedure to compute this $G_i(t)$ function.

Each basic block has a fixed number of memory requests which may lead to extrinsic cache miss during its execution. For each BB_i this number is given by

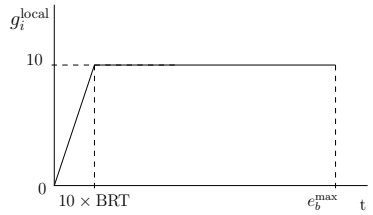


Fig. 2. BB_b Example g_b^{local} Function Where BB_b Execution May Generate at Most 10 Extrinsic Cache Misses

$$\text{extrinsic-miss}_b = |RCS_b \cap gen_b| \quad (4)$$

Where gen_b is defined as the set of memory blocks accesses generated during the execution of BB_b and RCS_b is the set of memory blocks that may be in the cache while BB_b is being executed. The intersection of both sets at each BB_b effectively yields the upper bound on the number of extrinsic cache misses that may occur while BB_b is executed after a preemption.

For each BB_b a function can then be constructed. This function g_b^{local}, has all the memory requests that may lead to an extrinsic cache miss in BB_b. These requests are assumed to occur as early as possible and at the maximum rate at which they can occur, as is displayed in Figure 2. This function is defined in the interval $[0, e_b^{\max}]$, which is the maximum length execution time interval for BB_b. Let us define the maximum time for memory block reload operation to complete:

$$\mathrm{BRT} \stackrel{\text{def}}{=} \frac{T_{\mathrm{HIT}} + T_{\mathrm{MISS}}}{T_{\mathrm{HIT}}}$$

where T_{HIT} is the time to serve a cache hit, and T_{MISS} is the time to serve a cache miss. The constant BRT imposes a restriction on the maximum rate in comparison to progression that preemption delay may be paid. Assuming $\mathrm{BRT} \neq \infty$ ensures that while paying preemption delay progression is still occurring, albeit at a slower pace. Intuitively BRT is an upper bound on the maximum rate at which cache miss penalty may be generated while executing the program. Formally the g_b^{local} function is defined per basic block as:

$$g_b^{\text{local}} \stackrel{\text{def}}{=} \begin{cases} \mathrm{BRT} \times t & , 0 \leqslant t \leqslant \frac{\text{extrinsic-miss}_b}{BRT} \\ \text{extrinsic-miss}_b & , \frac{\text{extrinsic-miss}_b}{BRT} < t \leqslant e_b^{\max} \end{cases} \quad (5)$$

A task-wide G_i function is constructed starting from the first BB_1 of the CFG. An initial function g_1^{in} is fed into the CFG. Where $g_1^{\text{in}}(t) = 0, \forall t$. The input functions for each BB_b are then merged together with g_b^{local} as is shown in Figure 3. Two merge operations are defined to aid the computation of the task-wide extrinsic cache misses function. Each block outputs a function g_b^{out}. Accordingly each basic block has one input function g_b^{in}. This input function is a product of the merging of all the output functions of the BB_b parent nodes. The input merging operation is defined by:

$$g_b^{\text{in}} \stackrel{\text{def}}{=} \bigoplus_{x \in pred(b)} g_x^{\text{out}}(t) = \max_{t \in [0, C_i], x \in pred(b)} \{g_x^{\text{out}}(t)\} \quad (6)$$

The input function is then merged with the corresponding node-specific g_b^{local} function using the merge-at-node operation defined

in the following way:

$$g_b^{\text{local}} \otimes g_b^{\text{in}} \stackrel{\text{def}}{=} \begin{cases} g_b^{\text{in}}(t) & , t \leqslant s_b^{\min} \\ g_b^{\text{in}}(t) + g_b^{\text{local}}(t - s_b^{\min}) & , s_b^{\min} < t \leqslant s_b^{\min} + e_b^{\max} \end{cases} \quad (7)$$

This assumes the knowledge of the earliest time the node may be accessed ($s_b^{\min}$). The g_n^{out} where BB_n is the last basic block in the CFG (return block) is the task-wide G_i function. iIn the case that several return blocks exist then the task-wide G_i function is obtained by combining the g_n^{out} functions of all the return blocks using the merge-at-edge operator.

$$G_i \stackrel{\text{def}}{=} \bigoplus_{j \in \text{RET}} g_j^{\text{out}} \quad (8)$$

The procedure described is graphically explained in Figure 3. In this figure only one BB_b is portrayed for clarification. The procedure repeats for all the BB_b of the CFG.

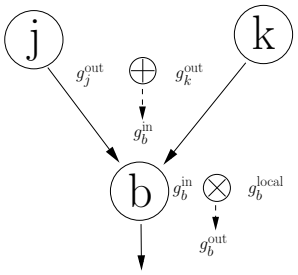


Fig. 3. g_b^{out} Computation for BB_b, Where BB_j and BB_k are BB_b Predecessors.

Theorem 1. *The function $G_i(t)$ is an upper-bound on the extrinsic cache misses occurring during the execution of task τ_i for any time t.*

Proof: The function g_1^{in} is an upper bound on the number of extrinsic cache misses for the time instant 0 since there can occur zero extrinsic cache misses until this time instant. For each basic block all the memory accesses which may generate an extrinsic cache miss are considered to occur as early as possible (s_b^{min}), at the maximum possible rate (BRT). Hence g_b^{local} is an upper bound on the extrinsic cache misses occurring in BB_b. Then, the function g_1^{out} is an upper-bound on the number of extrinsic cache misses until the execution of the task exits BB_1. The same function g_1^{out} is then the g_b^{in} for all the BB_b child nodes of BB_1.

Since the merge-at-node operation integrates the maximum number of extrinsic cache misses occurring in BB_b at the earliest time that these could occur (s_b^{min}) and at the maximum rate (BRT), it holds true that merge-at-node operation carried out in each of BB_1 child preserves the property that each g_b^{out} is itself an upper-bound on the number of extrinsic cache misses occurring from the start of task execution until it took the path leading to BB_b.

If a node BB_b has more than one parent then g_b^{in} is constructed using the merge-at-edge operator over all the node predecessors output functions (g_j^{out}). Since the merge-at-node operation takes the maximum value for all the $t \in [0, C_i]$ of the parents g_j^{out} output functions then it holds true that g_b^{in} is still an upper-bound on the extrinsic cache misses that could occur from the start of task execution until it took the path leading to BB_b. This reasoning holds true for all nodes in the CFG.

Lastly to compute the $G_i(t)$ function the merge-at-edge operator is applied across all the return edges of the CFG. Hence the $G_i(t)$ function is an upper bound on all the extrinsic cache misses occurring for all the possible execution paths in task τ_i. ■

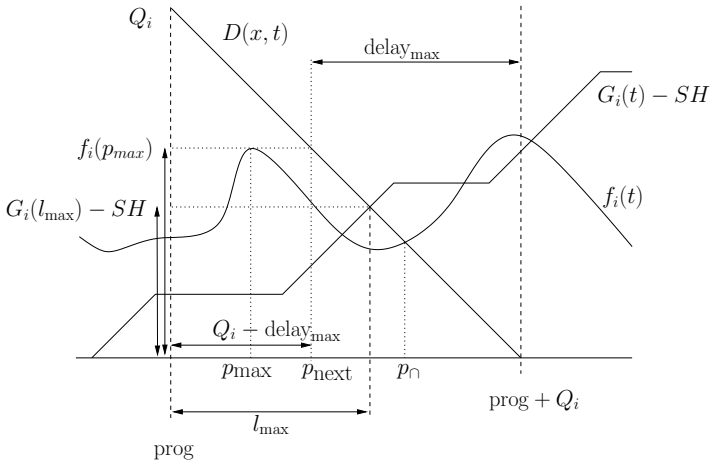


Fig. 4. Algorithm Iteration Sketch

V. PREEMPTION DELAY COMPUTATION

For the preemption delay upper bound computation two functions ($f_i(t)$ and $G_i(t)$) are considered. The $f_i(t)$ function represents an upper bound on the preemption delay a task may face for any point in time, whereas the G_i function yields the upper bound on the amount of preemption delay in the time interval $[0, t]$.

Using the knowledge provided by $f_i(t)$ function itself prevents the application of the method in scenarios where $Q_i \leqslant \max_t(f_i(t))$. This was one of the major limitations of the method presented in [1] which is now relaxed in this work. This fact arises from a situation where no progression can be performed since there is always more preemption delay to be paid than the length of the considered non-preemptive execution region. As stated in Section III a task will always execute non-preemptively for at least Q_i time units before a preemption occurs, unless it completes before the end of the non-preemptive region.

Description of the Preemption Delay Estimation Algorithm 1. Initially we will explain the intuition of the approach on Figure 4 before presenting the actual algorithm. Suppose that prog is the current progression in the task τ_i execution. Considering the next preemption point, the approach is looking for the lower bound on the progression which will be achieved within the next Q_i time units in any preemption scenario. For this, functions $f_i(t)$ and $G_i(t)$ are investigated from the current prog to $\text{prog} + Q_i$. On the ordinate also at length Q_i a line $D(x,t)$ is drawn to $\text{prog} + Q_i$.

Two intersection points with the $D(x,t)$ function are obtained for $f_i(t)$ and $G_i(t)$ with an offset of SH units. One is the point $p_\cap$ where $f_i(t)$ first crosses $D(x,t)$. It limits the range of values which need to be considered for the $f_i(t)$ function. A preemption past this value would lead to a situation where this point would again be considered in a subsequent iteration, since then prog would not pass this point in the current iteration.

Within the interval prog to $\text{prog} + l_{\max}$ or to p_{next}, $\text{delay}_{\max}$ is determined simply by choosing the minimum value between the maximum values of the functions $f_i(t)$ and $G_i(t) - SH$ in the given interval. That means in an interval Q_i under any preemption scenario at least $Q_i - \text{delay}_{\max}$ progress in program execution will be achieved. It is a conservative bound as a later preemption

means that also the non preemptible region will only start then. This point $\text{prog}+Q_i-\text{delay}_{\max}$ will serve as new starting point.

The second point is $l_{\max}$. At $l_{\max}$ the function $G_i(t)-\text{SH}$ intersects $D(x,t)$. At this point $G_i(l_{\max})-\text{SH}$ cache misses have occurred in the $[\text{prog}, l_{\max}]$ interval. Using $G_i(t)$ information only a progression of $\text{prog}+Q-G_i(l_{\max})-\text{SH}$ is then carried out where $\text{prog}-G_i(l_{\max})-\text{SH}$ are spent reloading cache content.

Algorithm 1: Upper-Bound the Preemption Delay

Input : $f_i(t)$: preemption delay function of task τ_i.
$G_i(t)$: extrinsic cache misses function of task τ_i.
Q: length of the non-preemptive region
Output: total_delay: cumulative preemption delay suffered by τ_i

```
1  total_delay ← 0;
2  delay_max ← 0 ;
3  p_next ← Q ;
4  SH ← 0 ;
   /* While the next progression is not beyond C_i  */
5  while p_next < C_i do
       /* Update time, progression and delay       */
6      prog ← p_next ;
       /* Compute the next progression step and the
          next delay to account for, based on the
          f_i(t) function                          */
7      p_∩ ← min{p_x} such that
8              p_x ∈ [prog, prog+Q]
9              and  p_x = prog+Q − f_i(p_x) ;
10     if p_∩ = null then  p_∩ ← prog+Q;
11     p_max ← arg max_{p_x ∈ [prog, p_∩]} {f_i(p_x)};
12     l_max = min((t|g^local(t) − SH = −t + prog+Q), prog+Q);
13     if  G_i(l_max) − SH ⩽ f_i(p_max) then
14         delay_max ← G_i(l_max) − SH;
15         SH = G_i(l_max);
16     else
17         delay_max ← f_i(p_max);
18         SH = SH + f_i(p_max);
19     delay_max ← min(f_i(p_max), G_i(l_max));
20     p_next ← prog+Q − delay_max;
21     total_delay ← total_delay + delay_max ;
22 return total_delay ;
```

Returning to the Algorithm 1: Lines 1–4 initialise the variables. The variable prog memorizes the current progression in the task's operations while total_delay records the cumulative preemption delay accounted for up to the current progression. As the task τ_i executes, it accounts for progressing in its execution (and the variable prog is increased) and for the preemption delay (which updates the variable total_delay). The algorithm is iterative, and at each iteration the variables $\text{delay}_{\max}$ and p_{next} are the preemption delay taking place only in the current iteration and the next progression point in τ_i's execution at which the next iteration will start, respectively. Lastly the variable SH, which represents the offset of the $G_i(t)$ function, is initialised. Lines 1–4 can be seen as the first iteration of the algorithm. $\text{delay}_{\max}$ is set to 0 and p_{next} to Q_i, because no preemption can occur during the first Q_i time units of τ_i's execution. The algorithm starts iterating at line 5, and it iterates as long as the next computed progression point p_{next} does not fall beyond τ_i's execution boundary. Line 6 shifts the current progression point of τ_i to the computed value p_{next}. In lines 7 – 11 the $f_i(t)$ preemption delay computation for the current iteration is carried out. Subsequently in lines 11 and 12 the the preemption delay computation is carried out with the $G_i(t)$ function information. From line 13 to 19 the a decision is carried out on the amount of preemption delay to consider in the current iteration. This value is the minimum between the one estimated with the $G_i(t)$ function and the $f_i(t)$ function, since both functions hold an upper bound on the preemption delay for a given interval. THen the next progression point p_{next} is computed with the knowledge of the maximum delay that τ_i could suffer while progressing in its operations from its current progression point to p_{next}. Finally, line 21 adds this maximum delay to the current cumulative delay accounted so far.

In the following Theorem 2, we prove that the value returned by Algorithm 1 is an upper-bound on the cumulative preemption delay that the given task τ_i might suffer during its execution. This implies that the WCET of τ_i (while taking into account all the possible preemption delays that τ_i might suffer during its execution) is given by

$$C_i' \stackrel{\text{def}}{=} C_i + \text{total\_delay} \tag{9}$$

where total_delay is the value returned by Algorithm 1.

Theorem 2. *Algorithm 1 returns an upper-bound on the preemption delay that a given task τ_i can suffer during the execution of any of its jobs.*

Proof: Algorithm 1 computes the maximum cumulative preemption delay iteratively, by progressing step by step through the execution of the task τ_i. Hereafter, we use the notation $\text{prog}^{(k)}$ to denote the progression through τ_i's execution at the beginning of the k^{th} iteration of the algorithm. Similarly, $\text{total\_delay}^{(k)}$ will be used to denote the cumulative preemption delay that τ_i has suffered until it reached a progression of $\text{prog}^{(k)}$. In this proof, we show that at each iteration $k > 0$, $\text{total\_delay}^{(k)}$ actually provides an *upper-bound* on the cumulative preemption delay that τ_i might suffer before reaching a progression of $\text{prog}^{(k)}$ in its execution. The proof is made by induction.

Basic step.

Algorithm 1 first considers that τ_i progresses by Q_i time units in its execution without suffering any preemption delay (since it cannot get preempted during these first Q_i time units). We consider this first step as the first iteration of the algorithm. That is, straightforwardly, $\text{total\_delay}^{(1)} = 0$ is an upper (and even exact) bound on the cumulative preemption delay that τ_i may suffer before reaching a progression of Q_i time units in its execution.

Induction step.

We assume (by induction) that $\text{total\_delay}^{(k)}$, $k \geqslant 1$, is an upper-bound on the cumulative preemption delay that τ_i might suffer before reaching a progression of $\text{prog}^{(k)}$ time units in its execution.

During the k^{th} iteration, Algorithm 1 computes $\text{prog}^{(k+1)}$ and $\text{total\_delay}^{(k+1)}$ as follows:

$$\text{prog}^{(k+1)} = \text{prog}^{(k)} + Q_i - \text{delay}_{\max} \tag{10}$$

$$\text{total\_delay}^{(k+1)} = \text{total\_delay}^{(k)} + \text{delay}_{\max} \tag{11}$$

where

$$\text{delay}_{\max} = \min(f_i(p_{\max}), G_i(l_{\max})) \tag{12}$$

$$p_{\max} = \underset{p_x \in [\text{prog}^{(k)}, p_\cap]}{\arg\max} \{f_i(p_x)\} \tag{13}$$

$$p_\cap = \min\{p_x\} \text{ such that } p_x \in [\text{prog}^{(k)}, \text{prog}^{(k)} + Q_i] \text{ and } p_x = \text{prog}^{(k)} + Q_i - f_i(p_x) \tag{14}$$

$$l_{\max} = \min((t | g^{\text{local}}(t) - SH = -t + \text{prog} + Q), \text{prog} + Q) \tag{15}$$

Equations 10 and 11 can be interpreted as follows. During the k^{th} iteration, Algorithm 1 assumes that τ_i executes for Q_i time units during which τ_i progresses by $Q_i - \text{delay}_{\max}$ units of time in its execution and incurs a delay of $\text{delay}_{\max}$; The algorithm assumes that τ_i gets preempted when its progression reaches $p_{\max}$ given by Equation 13. Below we show that choosing any other preemption point $p_{\text{other}} \neq p_{\max}$ would ultimately[1] result in a cumulative preemption delay lower than the one returned by Algorithm 1, thus showing that the value returned by Algorithm 1 is an upper-bound. Two cases may arise: $p_{\text{other}} > p_{\text{next}}$ or $p_{\text{other}} \leq p_{\text{next}}$.

Case 1: $p_{\text{other}} > p_{\text{next}}$. This means that τ_i progresses in its execution until it reaches p_{next} *without being preempted*, i.e., from a progression of $\text{prog}^{(k)}$, τ_i reaches a progression of p_{next} by being executed only for $(p_{\text{next}} - \text{prog}^{(k)}) \leqslant Q_i$ time units, and with an unchanged cumulative preemption delay of $\text{total\_delay}^{(k)}$. On the other hand, in the execution scenario built by Algorithm 1, τ_i's execution reaches a progression of $\text{prog}^{(k+1)} = p_{\text{next}}$ by being executed for Q_i time units, and with a cumulative preemption delay of $\text{total\_delay}^{(k+1)} = \text{total\_delay}^{(k)} + \text{delay}_{\max} \geq \text{total\_delay}^{(k)}$. In other words, Algorithm 1 manages to progress slower in τ_i's execution while suffering from a greater preemption delay. Furthermore, p_{other} is still a candidate preemption point for a further iteration of Algorithm 1.

Case 2. $p_{\text{other}} \leq p_{\text{next}}$. *After executing τ_i for Q_i time units*, we have that

1) the delay of the preemption that occurs when τ_i's progression reaches p_{other} has been totally accounted for (since $p_{\text{other}} < p_{\text{next}} \leq p_{\cap}$).
2) the progression of τ_i in this scenario becomes

$$\begin{aligned} \text{prog}_{\text{other}} &= \text{prog}^{(k)} + Q_i - f_i(p_{\text{other}}) \\ &\geq \text{prog}^{(k)} + Q_i - f_i(p_{\max}) \\ &\geq \text{prog}^{(k+1)} \end{aligned} \tag{16}$$

3) the cumulative preemption delay becomes

$$\begin{aligned} \text{total\_delay}_{\text{other}} &= \text{total\_delay}^{(k)} + f_i(p_{\text{other}}) \\ &\leq \text{total\_delay}^{(k)} + f_i(p_{\max}) \\ &\leq \text{total\_delay}^{(k+1)} \end{aligned} \tag{17}$$

Thus, after executing τ_i for Q_i time units Algorithm 1 progressed less in the execution of τ_i (Inequality 16) while suffering from a higher preemption delay (Inequality 17). As a consequence of Cases 1 and 2, it holds at each iteration of Algorithm 1 that choosing to preempt the task when it reaches a progression of $p_{\max}$ ultimately leads to an upper-bound on its cumulative preemption delay. ■

A. Reducing the pessimism of $f_i(t)$

For computation of the preemption delay estimation the obtained $f_i(t)$ function is pessimistic. The main source of pessimism is the fact that the earliest possible entry time for each basic block is considered in the function computation.

Even though the $f_i(t)$ function is an upper bound on the preemption delay that might be paid at any progression point t in the task a less pessimistic function may be extracted.

In fact if a basic block is accessed earlier than its worst-case access time then the execution is in a situation which will never lead to the WCET. If the task is preempted in this basic block at an earlier time than the worst-case access time, the preemption delay that has to be paid is already doubly accounted on the WCET computation.

To exemplify this intuition consider the example provided in Figure 1. For the case of BB_4, $s_b^{min} = 35$, $s_b^{max} = 65$ and $e_b^{max} = 20$. Let $\text{CRPD}_4 = 5$. If this BB_4 is accessed at $t = 40$, and a preemption occurs at $t' = 45$, when the task resumes its execution at most 5 units of time will be spent regaining cache state. The BB_4 exit time would be $t'' = 65$ whereas the worst-case exit time computed statically is $s_b^{max} + e_b^{max} = 85$. One can then observe that $t'' < s_b^{max} + e_b^{max} = 85$, and derive the conclusion that preemptions occurring in BB_4 at an earlier time distant from the WCET behaviour are already accounted for in the WCET computation. Clearly t'' is smaller than the statically computed worst-case exit time and hence it doesn't make sense to consider a preemption delay of 5 units when the basic block is accessed at time $t = 40$. A more thorough definition and explanation of the concept follows.

The optimised $f_i(t)$ function is then defined as:

$$f_i(t) = \max_b \{f_b^{\text{local}}(t)\}$$

where $f_b^{\text{local}}(t)$ is defined in the following manner:

$$f_b^{\text{local}}(t) \stackrel{\text{def}}{=} \begin{cases} 0, 0 \leqslant t < \max(s_b^{max} - \text{CRPD}_b, s_b^{min}) \\ \text{CRPD}_b \times (t - s_b^{max} - \text{CRPD}_b) \\ , \max(s_b^{max} - \text{CRPD}_b, s_b^{min}) \leqslant t \leqslant s_b^{max} \\ \text{CRPD}_b, s_b^{max} < t \leqslant e_b^{max} \\ 0, t > e_b^{max} \end{cases} \tag{18}$$

Lemma 1. *Assuming an entry time t_1 such that $s_b^{max} - CRPD_b \leqslant t_1 < s_b^{max}$, and a preemption occurring at t_1 The progression point achieved in a Q_i length time window assuming a preemption delay payment of $f_b^{\text{local}}(t_1)$ is equal to the progression point assuming an entry time $t_2 = s_b^{\max}$.*

Proof: $f_b^{\text{local}}(t_1) = t_1 - (s_b^{\max} - \text{CRPD}_b)$
$p_1 = Q_i - t_1 + (s_b^{\max} - \text{CRPD}_b) + t_1$
$p_2 = Q_i - \text{CRPD}_b + t_2$
$p_1 = p_2 \Leftrightarrow Q_i - (t_1 - (s_b^{\max} - \text{CRPD}_b)) + t_1 = Q_i - \text{CRPD}_b + t_2 \Leftrightarrow Q_i + s_b^{\max} = Q_i + t_2$.
Since $t_2 = s_b^{\max}$ the Lemma is proved ■

There exits a set of possible paths from BB_1 until BB_b such that for all these possible program paths BB_b will be accessed at a progression point p' in the program where $p' \in [s_b^{min}, s_b^{max}]$. For all possible paths where the entry time of BB_b for which $p' < s_b^{max}$ then the execution is earlier in relation to the WCET by $s_b^{max} - p'$ time units.

When computing the $f_i(t)$ function for each progression point considered the point with the maximum preemption delay is chosen. It is the case that for the paths which diverge from the WCET behaviour then paying preemption delay in this paths would just bring them closer to the WCET behaviour. As long as the progression is sufficiently distanced from the worst-case pattern for BB_b execution then no preemption delay has to be considered, since this would lead to a double accounting of the preemption delay.

[1] when τ_i's execution will be completed

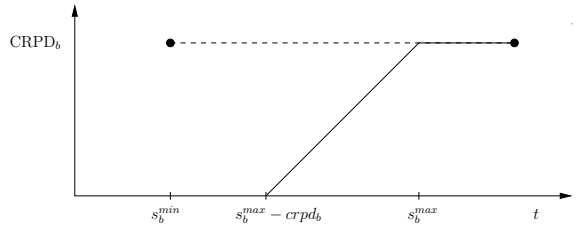


Fig. 5. $f_b^{\text{local}}(t)$ Graphical Example

Consider a situation where $s_b^{max} - p' > \text{CRPD}_b$, if the task would be executing in BB_b, by paying the preemption delay of CRPD_b would not bring this execution to the WCET scenario. In the same way if $0 < s_b^{max} - p' < \text{CRPD}_b$ then paying a portion of the preemption delay payment given by $s_b^{max} - p'$ would bring the execution exactly into the worst-case scenario. From this point onwards the maximum preemption delay for the basic block should be paid since the progression on BB_b is already in a worst-case scenario.

Lemma 2. *For all the* BB_b *that a given progression point* p *might belong to, the worst-case preemption delay is given by* $\max_b \{\max(\min(\text{CRPD}_b - (s_b^{max} - p), \text{CRPD}_b), 0)\}$.

Proof: For a given BB_b *and a current actual progression point* p*, if* $\text{CRPD}_b - (s_b^{max} - p) < 0$ *then the current execution in* BB_b *is* $s_b^{max} - p$ *time units earlier in relation to the worst-case entry time behaviour for the given block. Paying* CRPD_b *in this instance is not putting it in a worse situation than the worst-case entry* $s_b^{max} - p$ *and hence is not leading to an optimistic analysis result.* ■

Theorem 3. *The function* $f_i(t)$ *provides a safe upper bound on the preemption delay to be paid during a preemption of* BB_b *in a WCET context*

Proof: In Lemma 1 it is shown that any scenario where a basic block is considered to be accessed between $s_b^{max} - \text{CRPD}_b$ *and* s_b^{max} *will be treated such that is equivalent of accessing* BB_b *at time* s_b^{max}*. Additionally, in Lemma 2 it is proven that* BB_b *arriving before* $s_b^{max} - \text{CRPD}_b$ *the actual progression in the real execution is at least as good as arriving at* s_b^{max}*. Hence, using* $f_i(t)$ *provides a safe upper bound for the preemption delay to be taken into account during analysis.* ■

Figure 5 presents a graphical representation of the $f_b^{\text{local}}(t)$ function. From s_b^{min} to $s_b^{max} - CRPD_b$ the value of the function $f_b^{\text{local}}(t)$ is zero. In the interval $s_b^{max} - CRPD_b$ to s_b^{max} the function $f_b^{\text{local}}(t)$ has a first derivative of one. Lastly, from s_b^{max} to $s_b^{max} + e_b^{max}$ the function tops at the $CRPD_b$ value.

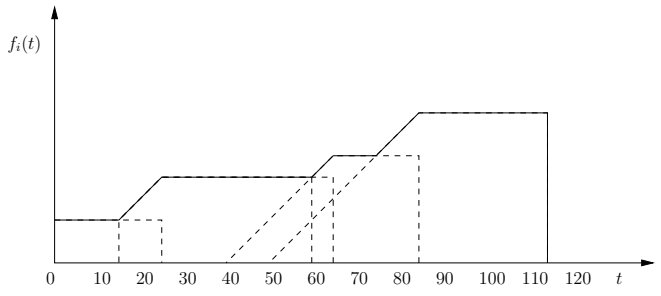


Fig. 6. Task Wide $f_i(t)$ Obtention

In Figure 6 the procedure to compute the task-wide $f_i(t)$ function is graphically portrayed. Several $f_b^{\text{local}}(t)$ functions are displayed with dashed lines. For all the instants in time, the maximum value of all the local $f_b^{\text{local}}(t)$ is taken as the value of $f_i(t)$.

B. Reducing the pessimism of $G_i(t)$

Following the same reasoning as in section V-A, the $G_i(t)$ function holds pessimism for the preemption delay computation in a WCET analysis context.

The merge-at-node operation is then defined as:

$$g_b^{\text{local}} \otimes g_b^{\text{in}} \stackrel{\text{def}}{=} \begin{cases} g_b^{\text{in}}(t) \\ , t \leqslant \max(s_b^{\max} - g_i^{\text{local}}(e_b^{\max}), s_b^{\min}) \\ \\ \max(g_b^{\text{in}}(t) + g_b^{\text{local}}(t - \max(s_b^{\max} - g_i^{\text{local}}(e_b^{\max}), s_b^{\min}))) \\ , \max(s_b^{\max} - g_b^{\text{local}}(e_b^{\max}), s_b^{\min}) < t < C_i \end{cases} \tag{19}$$

In this way the preemption delay is only accounted for when the task execution is in a path that would lead into the WCET.

Theorem 4. *The function* $G_i(t)$ *is an upper-bound on the extrinsic cache misses occurring during the execution of task* τ_i *for any time* t *in a WCET context.*

Proof: The redefinition of the merge-at-node operation integrates the g_b^{local} *function only at the time instant* $t = \max(s_b^{\max} - g_b^{\text{local}}(e_b^{\max}), s_b^{\min})$*. For all the time instants* $t' < t$ *if the* BB_b *is executing and generates an extrinsic cache miss then this value was already accounted for in the WCET analysis since:*
$t' + g_b^{\text{local}}(e_b^{\max}) + e_b^{\max} < s_b^{\max} + e_b^{\max} \Leftrightarrow t' + g_b^{\text{local}}(e_b^{\max} < s_b^{\max} \Leftrightarrow t' < s_b^{\max} - g_b^{\text{local}}(e_b^{\max})$
The only change to the computation procedure of $G_i(t)$ *lies on the merge-at-node operation. Hence proving the correctness of the merge-at-node operation redefinition along with Theorem 1 proves the present theorem.* ■

C. Experimental Evaluation

In order to experimentally validate both the new preemption delay estimation Algorithm 1 and the usefulness of its input functions, the $f_i(t)$ and $g^{\text{local}}(t)$ function extraction procedures were implemented in Heptane. Results are also provided for the previous function $f_i(t)$ [1]. The framework is trialled on a set of benchmarks available online [18]. From the set of available benchmarks only the results for acquisition task, autopilot task 6, autopilot task 9 and interrupt handler routine are shown. Due to space constraints, not all the benchmark results are presented.

All the results are given for a direct-mapped instruction cache of 4KB, with a line size of 32 bytes (8 instructions). The data cache is perfect (i.e. data cache misses are assumed never to occur). Only results regarding instruction caches are presented at this time since data caches were not yet addressed theoretically. The analyzed code is mips code (fixed-size 4B instructions). The code also works with set-associative cache. There is only one level of cache, with $T_{\text{HIT}} = 1$ cycle when the fetched instruction hits the cache, $T_{\text{HIT}} = 10$ cycles in case of a miss.

D. $f_i(t)$ *functions*

A comparative evaluation between the prior $f_i(t)$ function extraction procedure and the more recent one is presented. As is apparent in all the benchmarks the current method enables a

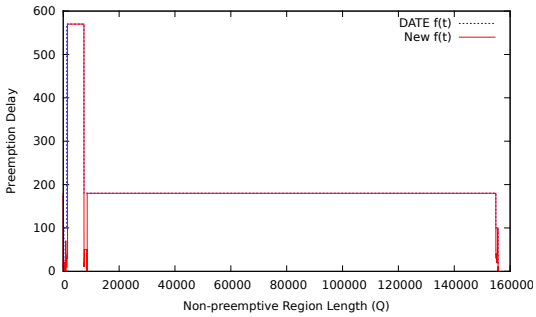


Fig. 7. [1] and Current $f_i(t)$ Function for Acquisition Task

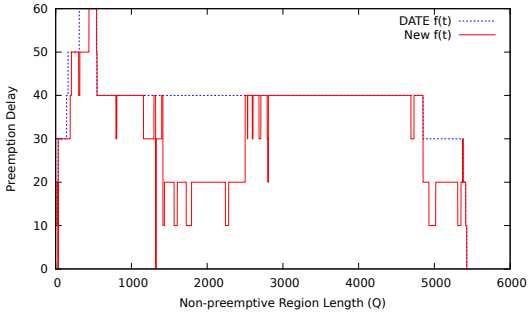


Fig. 8. [1] and Current $f_i(t)$ Function for Autopilot.t6

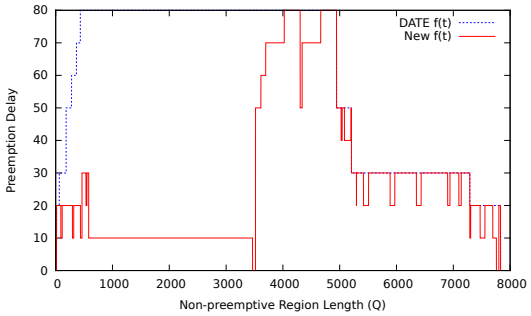


Fig. 9. [1] and Current $f_i(t)$ Function for Autopilot.t9

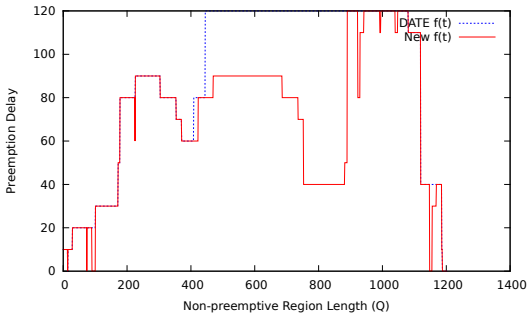


Fig. 10. [1] and Current $f_i(t)$ Function for Interrupt Handler

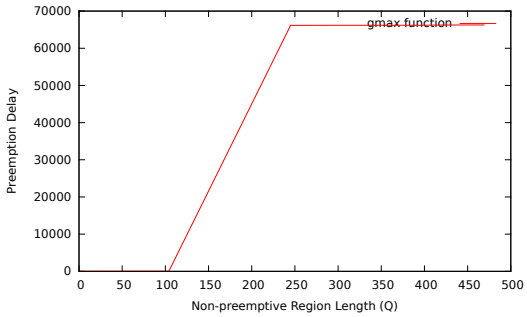


Fig. 11. g^{local} Function for Acquisition Task

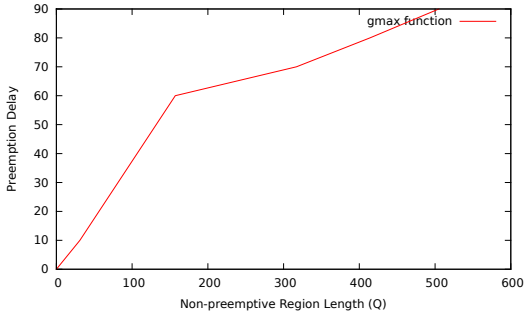


Fig. 12. g^{local} Function for Autopilot.t6

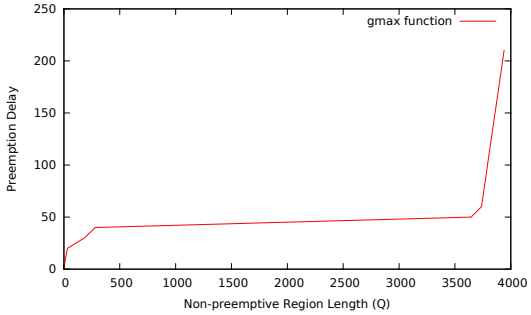


Fig. 13. g^{local} Function for Autopilot.t9

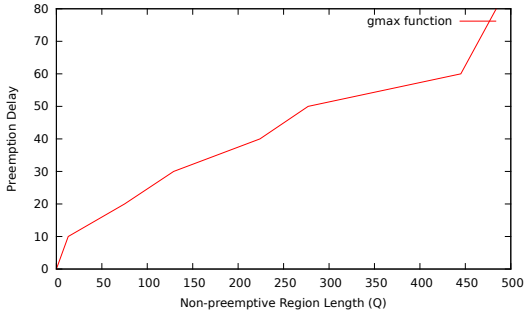


Fig. 14. g^{local} Function for Interrupt Handler

considerable increase in the amount of information present in the new $f_i(t)$ functions. The previous $f_i(t)$ [1] is always greater than the improved version. With this new concepts it is then possible to decrease the level of pessimism present in the analysis of the preemption delay. The major differences are present in the results shown in Figures 8, 9 and 10. For these benchmarks the results using the new $f_i(t)$ function are much less pessimistic than the prior method presented in [1].

E. $G_i(t)$ Functions

The $G_i(t)$ function enables the preemption delay to be computed whenever Q_i is smaller than any $f_i(t)$ value. In this subsection results for the $G_i(t)$ function extraction on the several selected benchmarks are presented. Only the less pessimistic version is presented. From Figures 12 to 14 it is apparent that the $G_i(t)$ function does not reach its peak value too soon. In Figure 11 the peak is reached much sooner in relation to the C_i of the task contrary to the other benchmarks.

F. Preemption Delay Estimations

The preemption delay estimations are presented in this subsection. Only the improved version of the $f_i(t)$ is used, since it is obvious that the results could never be worse than using the old version of the function. In all the results present in Figures 15 to 18 it is apparent that the results are much less pessimistic both for the previous method (which only uses $f_i(t)$ information) and for

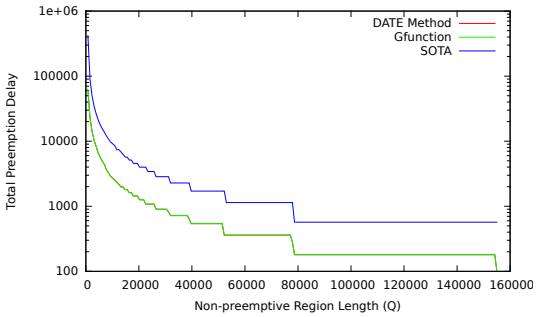


Fig. 15. Preemption Delay Estimations Function for Acquisition Task

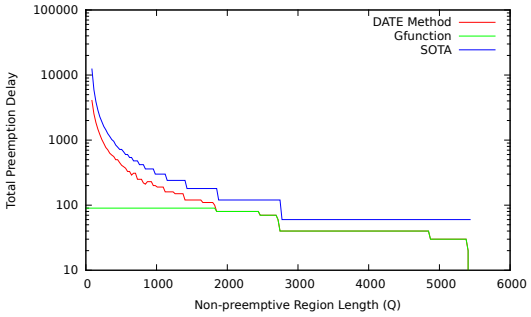


Fig. 16. Preemption Delay Estimations Function for Autopilot.t6

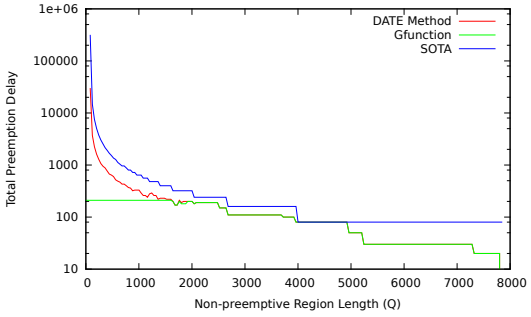


Fig. 17. Preemption Delay Estimations Function for Autopilot.t9

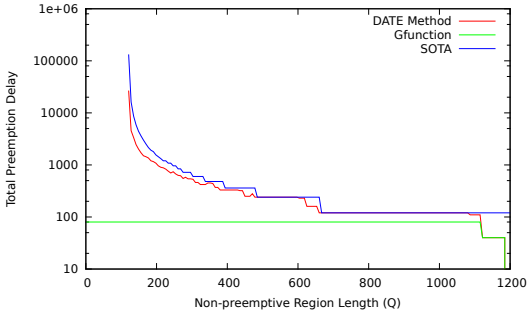


Fig. 18. Preemption Delay Estimations Function for Interrupt Handler

the new method with the $G_i(t)$ knowledge in comparison to the intuitive state of the art method. The state of the art method was published in [1] and is an intuitive upper bound on the preemption delay for the floating non-preemptive regions scheduling. The method using the $G_i(t)$ function information is never worse than the one which relies purely on the $f_i(t)$ function, and enables solutions to be obtained for situations where Q_i is lower than some value of $f_i(t)$ as is shown in figures 15 to 18.

VI. Conclusions and Future Work

This paper presented a refined model for calculation of the preemption delay. The $f_i(t)$ function is subject to a major optimisation considering the WCET context. More information is provided as well in the form of $G_i(t)$ function which enables the method to provide a solution in situations where the Q_i length is smaller than some $f_i(t)$ value. This was one of the major drawbacks of the previous work presented in [1]. The final contribution is the implementation of all the proposed methods and the extraction of preemption delay estimations for real world benchmarks. In this way we have shown that the proposed method enables considerable savings on the pessimism of the preemption delay estimation for floating non-preemptive region scheduling. Data cache analysis will be the subject of future work.

Acknoledgements

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within REPOMUC project, ref. FCOMP–01-0124-FEDER-015050, and by FCT and the EU ARTEMIS JU funding, within RECOMP project, ref. ARTEMIS/0202/2009, JU grant nr. 100202.

References

[1] J. Marinho, V. Nélis, S. M. Petters, and I. Puaut., "Preemption delay analysis for floating non-preemptive region scheduling," in *49th DATE*, Mar 2012.
[2] S. M. Petters and G. Färber, "Scheduling analysis with respect to hardware related preemption delay," in *Workshop on Real-Time Embedded Systems*, London, UK, Dec 2001.
[3] J. Marinho and S. M. Petters, "Job phasing aware preemption deferral," *EUC 2011*, Oct 2011.
[4] M. Bertogna and S. Baruah, "Limited preemption edf scheduling of sporadic task systems," *IEEE Trans. on Industrial Informatics*, vol. 6, no. 4, Nov 2010.
[5] C.-G. Lee, J. 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," *IEEE Trans. on Computers*, vol. 47, 1998.
[6] H. Ramaprasad and F. Mueller, "Bounding preemption delay within data cache reference patterns for real-time tasks," in *12th RTAS*, Apr 2006.
[7] J. Busquets-Mataix, J. Serrano, R. Ors, P. Gil, and A. Wellings, "Adding instruction cache effect to schedulability analysis of preemptive real-time systems," in *17th RTSS*, Jun 1996.
[8] J. Staschulat and R. Ernst, "Multiple process execution in cache related preemption delay analysis," in *Proceedings of the 4th ACM international conference on Embedded software*, ser. EMSOFT '04, 2004.
[9] S. Altmeyer, C. Maiza, and J. Reineke, "Resilience analysis: tightening the crpd bound for set-associative caches," ser. LCTES '10, 2010.
[10] S. Altmeyer, R. I. Davis, and C. Maiza, "Pre-emption cost aware response time analysis for fixed priority pre-emptive systems," in *32nd RTSS*, Nov 2011.
[11] L. Ju, S. Chakraborty, and A. Roychoudhury, "Accounting for cache-related preemption delay in dynamic priority schedulability analysis," in *44th DATE*, April 2007, pp. 1–6.
[12] A. Burns, "Preemptive priority-based scheduling: an appropriate engineering approach," in *Advances in real-time systems*, S. H. Son, Ed., 1995.
[13] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *6th RTCSA*, 1999.
[14] G. Yao, G. Buttazzo, and M. Bertogna, "Comparative evaluation of limited preemptive methods," in *15th ETFA*, Sep 2010.
[15] ——, "Feasibility analysis under fixed priority scheduling with limited preemptions," *Journal Real-Time Systems*, vol. 47, no. 3, 2011.
[16] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazz, "Optimal selection of preemption points to minimize preemption overhead," in *23rd ECRTS*, 2011.
[17] A. Colin and I. Puaut, "A modular and retargetable framework for tree-based wcet analysis," in *ECRTS 2001.*, 2001, pp. 37 –44.
[18] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. D. Michiel, "Papabench: a free real-time benchmark," in *6th WCET*, F. Mueller, Ed., 2006.
[19] H. S. Negi, T. Mitra, and A. Roychoudhury, "Accurate estimation of cache-related preemption delay," in *CODES+ISSS 2003*, Oct 2003.
[20] S. Altmeyer and C. Maiza, "Cache-related preemption delay via useful cache blocks: Survey and redefinition," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 707 – 719, 2011.