Improving the Worst-Case Execution Time Accuracy by Inter-Task Instruction Cache Analysis

Fadia Nemer, Hugues Cassé, Pascal Sainrat*
Institut de Recherche en Informatique de Toulouse(IRIT)
Université Paul Sabatier de Toulouse
Toulouse, France
{nemer, casse, sainrat}@irit.fr
* Hipeac Network of Excellence

Ali Awada
Computer Science Department
Lebanese University
Hadath, Lebanon
al_awada@ul.edu.lb

Abstract—**In hard real-time applications, WCET is used to check time constraints of the whole system but is only computed at the task level. While most WCET computation methods assume a conservative approach to handle the processor state before the execution of a task, the inter-task analysis of long effect hardware facilities should improve the accuracy of the result. As an example, we developed an analysis of a direct-mapped instruction cache behavior, that combines inter- and intra- task instruction cache analysis to estimate more accurately the number of cache misses due to task chaining by considering task Entry and Exit states along the inter-task analysis. The initial tasks WCET can be computed by any existing single-task approach that models the instruction cache behavior.**

Keywords – Worst Case Execution Time, Data Flow Analyses

I. Introduction

Checking the temporal behavior of critical hard real-time systems requires sound and accurate timing analyses: (1) an underestimation of the tasks execution time may cause catastrophic effects, and (2) an overestimation may waste hardware resources. The Worst Case Execution Time (WCET) analysis is a part of the timing analysis of a system. It computes an upper bound of a task execution time for a specific run-time environment.

In the last decade, an extensive research has developed methods and techniques for evaluating a WCET approximation based on static analysis approach [1, 2, 3, 4, 5, 6]. Usually, single-task based analyses manage the hardware state in a conservative way, a usual example of such state is the empty cache, ensuring an overestimation of the WCET. Although they provide safe approximations, a lot of factors in a multi-tasking system are not taken into consideration which definitely affect the accuracy of such timing estimates. For example, lots of works have been performed to analyze the cache behavior inside tasks [1, 2, 3, 4] in order to predict their timing properties but very few to handle the cache between tasks [5, 6, 15] and not directly for the sake of WCET.

In this paper, we present an approach to analyze the instruction cache behavior of tasks as well as the inter-task cache state in a real-time system with a static schedule of tasks to give an approximation of cache hits due to task chaining. This critical real-time system is made of a set of tasks that are subject to real-time constraints (i.e. deadlines) and that are sharing processing resources. To assert hard real-time properties and WCET computability, we are considering tasks running in a non-interruptible environment and statically scheduled.

The results are obtained by analyzing the addresses that are first loaded in the cache when executing a task. They are used to reduce the miss count evaluation induced by conservative approaches which leads to the optimization of the initial tasks WCET. These initial WCETs can be computed using any existing timing analysis modeling the instruction cache behavior. To analyze the inter-task cache states, we use a cycle of the static task schedule. A cycle is the sequence of tasks that are repeated in the same order in the schedule while the system is running. This sequence contains one or more occurrences of each task according to their periods.

The remaining of this paper is organized as follows. Section 2 introduces existing cache timing analyses. The problem is analyzed in section 3. Section 4 describes the characteristics of an Iterative Data Flow Analysis (DFA) and the details of the intra- and inter-task cache analyses. Experimental results are shown in section 5 and section 6 concludes the paper.

II. Related Works

Most WCET analyses are performed at the task level where a task is a non-interruptible code sequence. Caches are complicating timing analysis because their content depend on the program control flow. They improve the average memory access time as they are implemented with a faster memory than the main memory but they suffer from a smaller size due to cost consideration. So, they only store a small part of the memory and require special policies to select dynamically which part of the main memory to store and to wipe out during the execution of the program. Consequently, their behavior is control flow sensitive and hard to predict.

While both instruction and data caches have effects on the WCET estimation, we only consider in this paper, like most of WCET cache surveys, the instruction cache. The data cache causes even worst WCET pessimism and is often managed by direct program control [14].

Li and Malik [1, 2] propose a WCET analysis approach for a non-interruptible task called Implicit Path Enumeration Method (IPET). The program and the cache behavior are modeled as a set of integer constraints and a function whose maximization gives the WCET. They only consider inter-task cache effects by taking into account the possibility of the presence of the blocks in the cache before the task starts.

F. Mueller [4] presents a framework to handle WCET prediction. This static cache simulation method uses data-flow information to categorize instructions according to their caching behavior (always hit, always miss, first hit, first miss). The WCET is computed by traversing the CFG paths and propagating timing predictions within a tree in a bottom-up traversal. It seems there is no support for inter-task cache state management.

Ferdinand et al. [3] describe semantics-based analysis methods using abstract interpretation that predicts the cache behavior of programs. *MUST* and a *MAY* sets are provided to compute the set of memory blocks that, respectively, must/may be in the cache at a given program point under all circumstances. These sets are then used to assign to each memory reference one of the following categories: always hit, always miss, persistent, not classified. To our knowledge, this approach does not take in account the inter-task state.

All the aforementioned approaches focus on a single task timing analysis. To ensure a conservative computation of the WCET, they consider either an empty or an undefined cache state before the task. Yet, they may miss some opportunities to record hits in the WCET computation due to persistence of some blocks between different activations of the same task. This issue is discussed in [5, 6] that propose a timing analysis approach to compute Worst Case Response Time for (WCRT) in preemptive multi-tasking systems with caches. The approach focuses on cache reload overhead caused by preemptions. A path analysis is performed on the preempted and preempting tasks. The WCRT of each task is estimated by analyzing intra- and inter-task cache eviction.

Staschulat and Ernst, in [15], analyze the cache effects in a multi-process real time systems with preemptive static priority schedule to evaluate WCRT. They conduct experiments to quantify the cache effects and use the process response time for comparison. The core execution time of each process is simulated with an empty cache at startup to deliver conservative results, then the response time for multiple preemptions is computed. To lessen the complexity of the computation, only interruptions at basic block bounds were considered. Consequently, the response time is computed ignoring basic block overlapping causing a very approximative value on modern pipelined and superscalar processors.

[5, 6] and [15] computes the WCRT and not the WCET of the tasks and hence the approaches are applied mostly to soft real time systems and is bound to simple processors.

In this paper, we propose a task timing analysis for a multi-tasking critical real-time system, thus assuming no preemption between and inside tasks. As opposed to [1, 2, 3, 4], our analysis takes in account the execution context of the tasks. We compute the inter-task cache state and the set of memory addresses that will be first loaded into the cache when executing a task, in order to compute the hits count due to task chaining. The obtained results are straightforwardly used to improve the estimated WCET. While the approach in [5, 6] requires the computation of a list of sets for each instance of the tasks in the schedule, our approach uses only five sets for each task, four of these sets are computed once and used for every instance of the same task in the schedule and whatever the schedule. As opposed to [5, 6, 15], the simplicity of the analysis makes it applicable to complex processors.

Our approach is bound to non-preemptive tasks as found in usual critical industrial real-time systems (avionics for example). So, we have experimented it with PapaBench [10], a whole real-time application benchmark driving an Unmanned Aerial Vehicle. On the contrary, the method presented in [5, 6] and the simulations in [15] were tested on simple tasks configurations (two tasks for [15]) that (1) do not exhibit the properties of a complete real-time system environment and (2) do not provide any evidence of computation complexity scaling with bigger realistic systems.

Our approach guarantees the validation of the system timing constraints without wasting time clearing the cache memory .

III. Problem Definition

In this section, we detail the behavior of the cache all along the execution of the control loop of a real-time system.

A. Context

Let be a real-time system composed of n non-interruptible tasks named T_1, T_2, ... T_n. Each task T_i has a period P_i. T_i is ready to run at the beginning of its period and the deadline of T_i is at the end of its period. No constraint holds on the task-scheduling algorithm apart from being static, that is, it produces the schedule as a static table containing the order and the activation date of the tasks before the system start. Usually, the scheduling algorithm requires the period and the WCET of each tasks. The latter parameter is estimated by the OTAWA framework [8, 9] assuming a conservative hardware state at the entry of the tasks, i.e. an empty cache.

According to the static schedule of the real-time applications, we will analyze the instruction cache state between tasks in order to reduce the pessimism induced by the conservative WCET computation. To this end, we need to analyze the cache behavior inside and between the tasks. According to the tasks period, a task may be activated many times and each instance is named $T_{i,j}$, the j^{th} instance of task T_i in the control loop.

B. Cache Behavior in the task

We propose to improve the WCET of statically-scheduled tasks involved in an application by taking into account the hardware state between tasks. As the tasks may execute many instructions, only the hardware features, whose state depends on old instruction executions, should be taken into account. Consequently, we focus on the instruction cache.

Although static analysis approaches managing the instruction cache effects only apply to single tasks, they provide an insight about the cache behavior in the task. We face two problems (1) how to turn false miss induced by the conservative state of the cache to hits and (2) how to use the intra-task results to model the cache behavior all around the application control loop. We have based our intra-task analysis on the *MAY* / *MUST* method developed by Ferdinand [3].

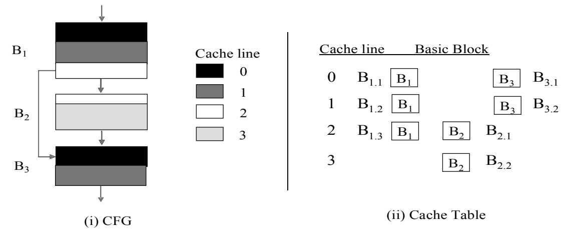

Figure 1. *L-block* construction

This approach is based on a Control Flow Graph (CFG) which describes the control structure of each task. As a function may be called many times from different locations of the program, every call is treated as an in-line during the CFG construction. This improves the sensitivity to the context and makes easier the analyses implementation. A CFG is considered as a pair (*N, E*) where the nodes, *N*, represent basic blocks[1] denoted by B_i and the edges from *E,* the control transfer between basic blocks.

The execution time of an instruction depends upon the behavior of the instruction cache. Whether it has been recently executed, it may be still stored in the cache, resulting in a fast access called a cache *hit*, or it may have been wiped out from the cache resulting in a long memory access called a cache *miss*. Usually, the memory is composed of cache blocks of the same size. Each cache block may be stored in one of the cache lines. When a cache block is accessed, either it is already in the cache, or a block in its matching line need to be wiped out in order to make place for the new block loaded from the memory. In our experimentation, we are using direct-mapped cache whose lines can only contains one cache block and do not need any complex replacement policy.

As the cache memory accesses are performed by blocks of constant size, we use the *line-block* unit, or simply *l-block*, to consider the cache effects on the instruction execution time. A *l-block* is a maximal contiguous sequence of instructions contained in a basic block that are mapped to the same line in the instruction cache. Two *l-blocks* that map to the same cache line conflict with each other if the execution of one displace the cache content of the other. Although a cache line can hold one cache block, this one may be composed of several *l-blocks*. These *l-blocks* are mapped to the same cache line but they are not conflicting with each others because they are contained in the same cache block. For example, *figure 1* shows a CFG with three basic blocks. We assume that the cache has four lines and suppose that a basic block B_i is partitioned into *s l-blocks* denoted by $B_{i,1}, \ldots, B_{i,s}$. The *l-blocks* of each basic block mapping to the same cache line are painted with the same pattern. One can remark that $B_{1.1}$ and $B_{3.1}$ are conflicting *l-blocks* while $B_{1.2}$ and $B_{2.1}$ are not.

From such a CFG with partitioned basic blocks, the algorithm provided in [3] computes by abstract interpretation and for each point *L* of the program two sets:

- ***MUST(B_i)***: list of *l-blocks* that must be in the cache whatever the execution path, after $\boldsymbol{B_i}$ execution.
- ***MAY(B_i)***: list of *l-blocks* that may be in the cache for one execution path at least, after $\boldsymbol{B_i}$ execution.

1. A basic block is a maximum sequence of consecutive instructions which control flow enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

These sets may be used to categorize the *l-blocks* and to derive cache miss count and the time penalty for each instruction.

C. *Cache behaviour between the tasks*

For all single task based analyses, the cache are considered empty at the start of the task to ensure an overestimation of WCET. Yet, according the actual state of the cache before the task, some hits have been considered conservatively as misses. Now, we examine the cache state behavior between tasks in order to remove these false misses.

We compute for each task an interval of the number of cache hits which are considered as misses by the conservative approaches. The upper bound of the interval is the maximum possible number of hits (depending on the control flow of the task) while the lower bound is the number of hits forgotten whatever the real execution path of the task. To obtain such a result, we need to estimate the cache state before each task, which cache blocks of this state may cause hits and which cache blocks stay in the cache after the task execution according to the schedule chaining.

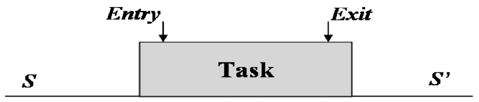

Figure 2. Task Characteristics

In our analysis, each task will be characterized by four cache block sets as shown in *figure 2*

- the cache state *S* before the task execution,
- the cache state *S'* after the task execution,
- *Entry*, the set of cache blocks that will be first loaded in the cache when executing the task and
- *Exit*, the set of cache blocks that will be in the cache after the task execution.

Entry and *Exit* are computed ignoring the inter-task cache state and assuming an empty cache or an undefined cache state at the start of the task. *S'* is a function of *S* and *Exit*. As *S'* is the set of cache blocks after the task execution, it contains the cache blocks produced by the task (*Exit*) and the cache blocks that were in the cache before the task execution and were not replaced by the task cache blocks (which is a subset of *S*). The *Entry* and the *Exit* sets derive from the intra-task cache analysis. However, *S* and *S'* proceed from an inter-task cache behavior analysis. We need to compute *Entry* and *Exit* separately because the intersection of *Entry* and *S* produces the number of false misses, while the *Exit* set is prominent to compute *S'* that will be used as the input cache state (*S*) for the next task in the schedule.

Ferdinand proposes only an *Exit* analysis in [3] that predicts the cache behavior using *MUST* and *MAY* analyses. Same as Ferdinand, we propose *MUST* and *MAY* analyses to compute the *Exit* set. However, we have performed our computation using an iterative Data Flow Analysis (DFA) [13] and we are only interested in the *Exit* set at the end of the task execution and not to evaluate the WCET.

Yet, we provide in this paper a *MUST* and a *MAY* analyses for the task *Entry*. The computed *MUST* and *MAY Entry* sets holds the list of *l-blocks* that are (respectively may be) first loaded in the cache when a task executes. The *Entry* analysis was not considered before because the majority of WCET computation methods are single-task based. We analyze the *Entry* of a task *T* as only these task *l-blocks* are affected by the task chaining. In other words, we have conservatively forgotten cache hits of *l-blocks* that are in the *Entry* set and in the cache state before the execution of the task. The remaining *l-blocks* of the task are only influenced by the *l-blocks* that are loaded in the cache during execution of the task.

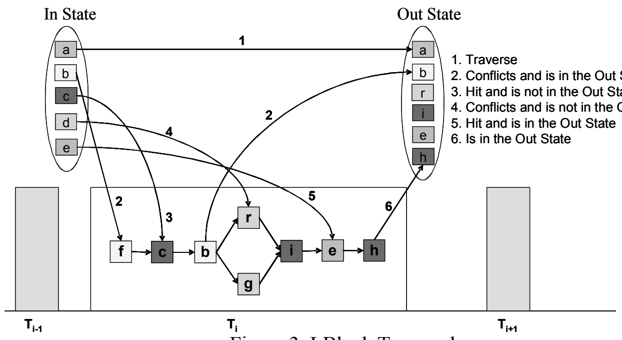
In State
Out State
a
b
c
d
e
a
b
r
i
e
h
1
2
3
4
5
6
1. Traverse
2. Conflicts and is in the Out State
3. Hit and is not in the Out State
4. Conflicts and is not in the Out State
5. Hit and is in the Out State
6. Is in the Out State
f
c
b
r
g
i
e
h
T_{i-1}
T_i
T_{i+1}

Figure 3. LBlock Traversal

Figure 3 exhibits several behaviors of instruction cache *l-blocks*. It displays an example of an application tasks schedule: the time runs from left to right. We focus on the internal CFG of the task T_i drawn horizontally. According to the given *In* cache state, we can compute the *Out* state after T_i execution whatever the actual execution of the CFG. The *l-blocks* having the same color are mapped to the same cache line and as we consider a direct mapped instructions cache these *l-blocks* conflict with each other.

The *l-blocks* that are affected by the task chaining are *{f, c, g, r, e, h}* because they are the first encountered *l-blocks* in their own cache line. *r* and *g* provides very simple cases (edge 4): they are never in the *In* state and consequently cause ever a *miss*. Because *f* and *c* are first accessed in the CFG (2, 3), *b* and *i* are never affected by the task chaining. When T_i is running, *f* evicts *b* from the cache generating a cache miss. As opposed to *c* and *d*, *b* is found in the Out state because it is reloaded after *f* eviction and no other *l-block* wipes it out (2). On the other hand, *c* and *e* is ever in the *In* state so that a hit is generated when the *l-block c* is accessed (3, 5): this is one of the forgotten hits considered as a miss with a conservative empty cache state. Finally, we can notice that the line containing *a* is not changed (1) by the task making the task execution transparent for this line. The case of *h* (6) is trivial too: it fills a non-used cache line.

As the *Exit* analysis, the *Entry* analysis does not model the instruction cache behavior to compute the task WCET but is dedicated to the evaluation of the forgotten hit count in the start cache state *S*. The details of the *Entry*, *Exit* and cache state analyses are shown in the following sections.

IV. Cache Analyses

We begin this section with a general overview of the used Data Flow Analysis and then we explain the characteristics of our intra- and inter-task cache analysis.

A. Data Flow Analyses

A Data Flow Analysis (DFA) is a process to collect run-time information about data items in programs without actually executing them. DFA algorithms are frequently defined using operations on control flow graphs (CFG). We use an Iterative DFA algorithm that combines and manipulates information, added to the CFG nodes and edges, using a set of equations that relate information at a given node to the information at other nodes. A typical equation is *Out(n) = (In(n) - Kill(n)* ∪ *Gen(n)*, which can be read as "*the information at the output of a node n is either generated in n or it enters at the input and is not deleted in n*". In general *In(n)* is a function that collects information from a subset of neighbors of *n*, either the successors or the predecessors of *n* in the graph.

```
/* Initialisation */
$\text{In}[B_i] = \emptyset$ ; / $\forall B_i$
$Out[B_n] = \emptyset$ ;
/* Main Loop */
do { $convergence = true$ ;
for $B_i$ do {
  $\textbf{In}[B_i] = \cap\ \textbf{Out}[s]$ ; // $s \in Predecessor[B_i]$
  $OldOut = Out[B_i]$; $\textbf{Out}[B_i] = (\textbf{In}[B_i] - \textbf{kill}[B_i]) \cup \textbf{Gen}[B_i]$;
  if $Out[B_i] \neq OldOut$ then $convergence = faux$ ;}}
while ( $!convergence$ )
```

Figure 4. General DFA algorithm (A1)

The DFA is a static program analysis and therefore, makes an approximation of the program behavior and performs a fixed- point computation for the body of the repetition statements. Such a fixed point always exists because (1) the data flow equations compute sets of variables in a monotonic way and (2) there is only a finite number of variables available since we consider only programs with finite number of statements. Therefore, there is a finite upper limit to the number of elements of the computed sets which means that a fixed-point always exists. *Figure 4* shows the general Iterative DFA algorithm. It consists of an initialization part where we must initialize the set to compute (*In* and *Out*) and the main loop which iterates until the convergence of the *In* sets (and the *Out* sets as a consequence). Therefore, we use a boolean variable *convergence* that will not be set to true unless all the *Out* sets remain unchanged. This algorithm may be applied forward or backward according the execution order depending on the extracted information. Evaluation order of basic blocks in the CFG has no effect on the result except to speed up the convergence to the fix point.

B. Intra-Task Cache Analysis

In both *Entry* and *Exit* analyses, we use the function *lines (l-block set)* that returns the sets of all *l-blocks* of the CFG mapped to the same line than an *l-block* of the considered set. Let *LB* be the set of the CFG *l-blocks* and *line(l)* computing the cache line containing *l-block l*. For each basic block B_i, we use four sets: ***In*** is the set of *l-blocks* available at B_i's entry, ***Out*** is the *l-blocks* set after B_i execution, ***Gen*** is the set of *l-blocks* accessed by B_i and ***Kill***. Equations (1), on the next page, shows how *Gen* and *Kill* are computed for a basic B_i.

$$Gen[B_i] = \{l \in B_i\}$$
$$Kill[B_i] = \{l \in LB \ / \ \exists l' \in B_i \ . \ line(l) = line(l')\} \quad (1)$$

Same as Ferdinand, we propose a *MUST* and a *MAY* analysis to compute the *Exit* set. However, we have performed our computation using an iterative DFA and we are only interested in the *Exit* set at the end of the task execution. The *MUST* analysis provides the set of *l-blocks* that must be in the cache after the execution of a task T_i while the *MAY* analysis gives the set of *l-blocks* that may be in the cache after the task execution.

```
/*Initialisation*/  $Out[B_1]=Gen[B_1]; In[B_i]=LB;$
/* Main Loop */  $In[B_i]=\bigcap Out[p]; // p \in Predecessors[B_i]$
```

Figure 5.*MUST Exit* analysis

```
/*Initialisation*/  for $B_i$ do {  $Out[B_i]=\emptyset; In[B_i]=\emptyset;$}
/*Main Loop*/  $In[B_i]=\bigcup Out[p]; // p \in Predecessor[B_i]$
```

Figure 6. *MAY Exit* analysis

The CFG is traversed in a descending order starting from B_1. We use the algorithm A1 with the changes shown in *figure 5* for the *MUST Exit* analysis or with the changes shown in *figure 6* for the *MAY Exit* computation. Note that the *Exit* analysis results are used to compute the cache state after the task execution.

Then, we propose a *MUST* and a *MAY* analysis to compute the task *Entry* set. The *Must* analysis gives the set of *l-blocks* that must be first loaded in the cache when executing the task. And the *May* analysis produces the set of *l-blocks* that may be first loaded in the cache.

```
/* Initialisation */
$Out[B_n]=Gen[B_n]; In[B_i]=LB; / \forall B_i$
for $B_i \neq B_n$ do  $Out[B_i]=LB-Kill[B_i];$
/* Main Loop */
$In[B_i]=\bigcap Out[s]; // s \in Successor[B_i]$
```

Figure 7. *MUST Entry* analysis

```
/*Initialisation*/  for $B_i$ do {  $Out[B_i]=\emptyset; In[B_i]=\emptyset;$}
/*Main Loop*/  $In[B_i]=\bigcup Out[s]; // s \in Successor[B_i]$
```

Figure 8. *May Entry* analysis

Having a CFG with *n* basic blocks, we apply the algorithm *A1* with the changes shown in in *figure 7* for the Must analysis and those shown in *figure 8* for the *MAY* Analysis. *A1* goes through the CFG in an ascending order starting from the last block B_n and using a basic block iterator on the CFG. *Must* and the *May* analysis to compute the *In* and the *Out* sets using an iterative approach.

There are only a finite number of cache lines and for each task, we have a finite number of memory blocks. Additionally, the union (intersection) operators applied to decreasing (increasing) *Out* sets produce a decreasing (increasing) set at each iteration of the main loop. This guarantees the termination of the analyses.

C. Inter-Tasks Cache Analysis

Our objective is to compute the WCET of a complete application cycle. Hence, we need to analyze the inter-task cache effects to have a tight value of this WCET.

Previous section has shown how we can characterize the behavior of the cache in the task using *MUST* and *MAY* cache states on exit, called respectively $MUST_{Exit}$ and MAY_{Exit}, and first accessed blocks on entry with sets called $MUST_{Entry}$ and MAY_{Entry}. Moreover, the cache state before the execution of a task is the cache state after the execution of the previous task in the schedule (as the task schedule produces a cyclic graph, there is always a predecessor to each task). To perform the analyses of the cache between tasks, we apply the iterative DFA algorithm to the graph formed by the application scheduling whose nodes are the tasks instances. One may observe that this graph, formed by an unterminated loop containing a sequence of nodes, is simpler than a usual CFG.

For each task T_i and each analysis, we use four sets: ***In*** is the set of *l-blocks* available at T_i entry, ***Out*** is the *l-blocks* set after T_i execution, ***Gen*** is the set of *l-blocks* produced by T_i and ***Kill*** is the set of *l-blocks* that were in the *In* set and were replaced by T_i *l-blocks*. In the *MUST* analysis, *Kill* contains the list of *l-blocks* that may be evicted, that is, the MAY_{Exit} *l-blocks*, while for the *MAY* analysis, it contains the list of *l-blocks* evicted whatever the execution path as in the $MUST_{Exit}$ *set*.

In the *MUST* analysis, the computed cache state holds the *l-blocks* that will be in the cache whatever the execution control flow of T_i. The T_i execution paths give different exit sets. These sets includes at least the $MUST_{Exit}$ set as it is the minimal exit set. Hence, to build the *MUST* cache state after T_i execution, we must evict the list of *l-blocks* that conflict with the elements of the MAY_{Exit} set. In the other hand, to compute the *MAY* cache state, holding the *l-blocks* that will eventually

```
/*Initialisation*/
$In[T_1]=\emptyset;$
$Out[T_i]=Must_{Exit}[T_i];$
$Gen[T_i]=Must_{Exit}[T_i];$
$Kill[T_i]=lines(May_{Exit}[T_i]);$
/*Main Loop*/
$In[T_i]=\bigcap Out[T]; // T \in Predecessors[T_i]=T_{i-1}$
```

Figure 10: Cache State MUST Analysis

```
/*Initilisation*/
$In[T_1]=\emptyset;$
$Out_{initial}[T_i]=May_{Exit}[T_i];$
$Gen[T_i]=May_{Exit}[T_i];$
$Kill[T_i]=lines(Must_{Exit}[T_i]);$
/*Main Loop*/
$In[T_i]=\bigcup Out[T]; // T \in Predecessors[T_i]=T_{i-1}$
```

Figure 9. Cache State May Analysis

be in the cache after T_i execution, we must use the $MUST_{Exit}$. Hence we must evict from the cache the *l-blocks* conflicting with its elements.

To perform the computation, we assume an empty cache to initialize the input state $In[T_i]$ of each task T_i. We use A1 with the definitions shown in of *figure 10* to implement the *MUST* analysis or those shown in *figure 9* for the *MAY* analysis.

V. Experimental Results

Our approach aims to tighten the WCET of tasks involved in a complete application cycle taking into account the inter-task instruction cache state. With such a scheme, the WCET of each task depends on the task itself but also on its location in the task chain. Therefore, a different WCET is computed for each instance of the task in the application execution cycle. To this end, we evaluate the lower bound of cache hits (from the *MUST* analysis) used to fix the task WCET and an upper bound (from the *MAY* analysis) that can be used to recompute the WCET of the task in a context-sensitive way. We have experimented our approach on a real-time benchmark modeling a full application and the results are exposed here.

A. *Benchmarks*

To get objective results from our experimentation, we need a whole real-time application and we want also to avoid unrealistic benchmark made of unrelated benchmark pieces. As no such a benchmark was existing to our knowledge, we have derived one from an actual application driving an UAV, called PapaBench [10].

ID	***Description***	***Processor***	***Period***
T3	Receive MCU0 values	MCU1	20Hz
T4	Transmit Servos	MCU1	20Hz
T5	Check Failsafe	MCU1	20Hz
T7	Stabilization	MCU0	20Hz
T8	Send Data to MCU1	MCU0	20Hz
T9	Receive GPS Data	MCU0	4Hz
T10	Navigation	MCU0	4Hz
T11	Altitude Control	MCU0	4Hz
T12	Climb Control	MCU0	4Hz
T13	Reporting Task	MCU0	10Hz
I1	Transmission Servos interrupt	MCU1	20Hz
I2	SPI interrupt of MCU1	MCU1	20Hz
I4	SPI interrupt of MCU0	MCU0	20Hz
I5	Modem interrupt	MCU0	10Hz
I6	GPS interrupt	MCU0	4Hz

TABLEI. Autopilot Tasks

PapaBench has 13 tasks and 6 interrupts. We use an AADL [12] model of this application assuming that tasks and interrupts are periodic (consequently, in the following, we do not distinguish explicitly between tasks and interrupts and use the task term for both). The AADL models the benchmark as a bi-processor architecture separating the radio/servo command management, handled by the processor MCU1, from the autopilot task managed by the processor MCU0. Notice that processor and their memory are independent and only linked by a serial interface : actually, PapaBench contains two embedded real-time applications, one for each processor.

As PapaBench has two operation modes, we choose to analyze the automatic mode as it exhibits the most critical part of the application. *Table I* shows the task list of the automatic mode schedule and provides for each one an identifier, a description, the corresponding processor and a period. The deadline of a task is at the end of its period. As a sample of the application cycle, each time T4 is executed, six instances of interrupt I1 are also executed on the same processor, one per servo and the execution time will vary between 1ms and 2ms.

We analyze the task scheduling of each processor alone because the separation of tasks execution makes the propagation of tasks execution effects impossible between the two processors. Hence, the cache behavior is only affected by the list of tasks executed on the corresponding processor.

B. *Tools*

Our cache static analysis has been performed in the OTAWA [8, 9] framework. In this tool, the WCET computation is viewed as performing a chain of analyses that use and produce annotations hooked to the code until getting the WCET evaluation. We choose the Implicit Path Evaluation Technique [1] in our experiments to compute the WCET and the CAT approach to estimate the effects of the instruction cache on execution time. CAT is the adaptation of the Muller's categorization approach [4] to IPET. To perform this work, OTAWA builds the CFG of the tasks and performs several analyses to finally represent the WCET as a set of linear constraints. The maximization of a cost function according to the constraints by an Integer Linear Programming solver (lp_solve) gives the WCET value [1].

The static schedules of each processor tasks used in our experiments have been generated by CHEDDAR [11], a resource requirements analyzer. It provides analytical and simulation performance analysis methods / tools and it focuses on tasks, processors, shared resources, buffers and task dependencies. It fits well the requirement of our experiments because (1) it supports AADL used to model the real-time application of PapaBench and (2) it supports the task periods and the precedence rules of the original application to perform its schedule.

We used a fixed-priority scheduling called “Posix Highest Priority First” where task priorities range from 1 to 255. The algorithm schedules the tasks according to their priority and their precedences. In the future, we plan to analyze the effects of the scheduling protocols to the cache state. So, we will consider these protocols in our future experimentations.

To perform the experimentation, we have tested different instruction cache configuration with the same processor pipeline. As our IPET approach uses processor simulation to compute the times of program parts, we have chosen a generic

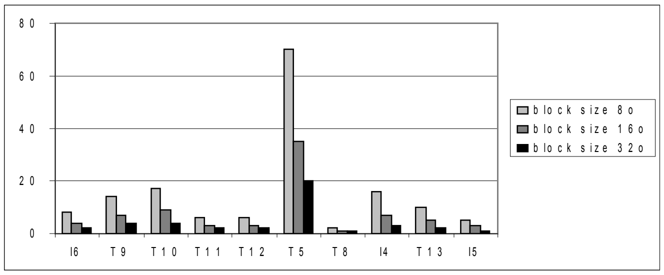

Figure 11. MCU0 Hit count (cache size 64KB)

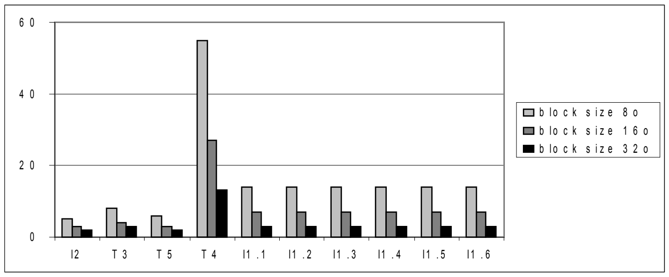

Figure 13.MCU1 average hit count (cache size 64KB)

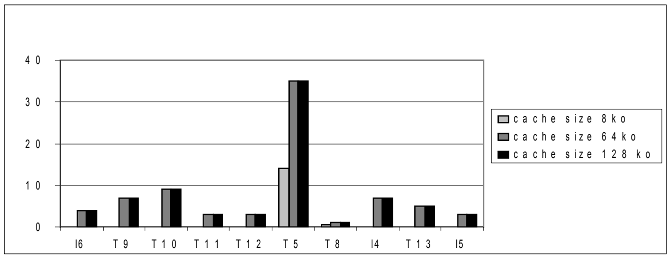

Figure 12: MCU0 average hit count (cache block size 16 B)

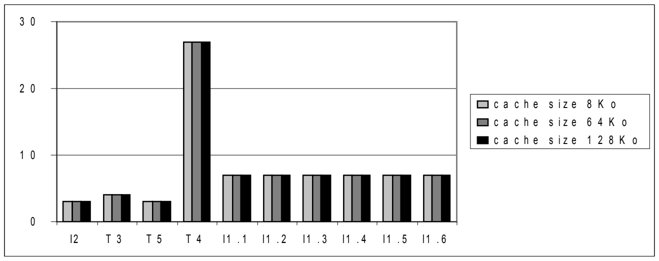

Figure 14.MCU1 average hit count (cache block size 16 B)

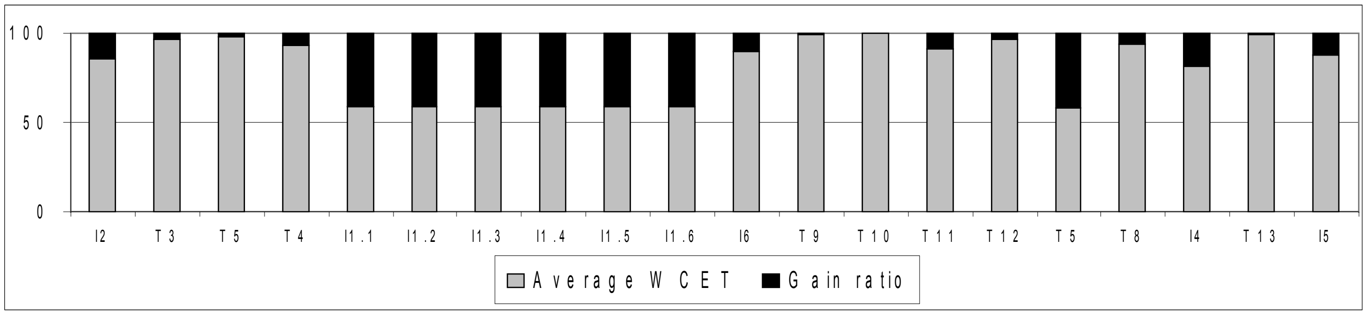

Figure 15. WCET Improvement

four stages in-order pipeline with execution in-order. All stages process two instruction by cycle with two ALU and two float computation functional units. The used binary programs was compiled for a PowerPC ISA but the used generic simulator supports any architecture available on OTAWA. The program running on MCU1 processor has a code size of 320 Kb while the program on MCU0 is a bit bigger with 370 Kb.

C. *Results*

In our experiments, we analyze different configurations of a direct-mapped instruction cache to evaluate their impact on the WCET estimation with our method. The cache size and the block size effects are taken into consideration.

First, we have fixed the cache size to 64 Kb and tested different block sizes: 8 bytes, 16 and 32 bytes. The results are shown for the two programs in *figure 11* and *13*. Vertically, we measure the average number of forgotten cache hits per task instance, that is, the number of *l-block* in the *MUST* causing hits whatever the execution path. Horizontally, we have the list of tasks with one bar for each configuration. The number of hits represents a lower bound of the pessimism in the WCET evaluation with a conservative approach and, conversely, the minimal gain of our approach.

The average count of hits – 1304 / 663 / 334 per block size – is significant and enforces the interest in our approach. Then, one may notice that little cache block size exhibits best inter-task analysis results. Small cache basic blocks benefit from the random distribution of code in memory and create less cache block aliasing effect on cache wiping between tasks.

In the second set of measures, we have fixed the cache block size to 16 bytes and we have performed our analysis for different cache sizes: 8Kb, 64Kb and 128Kb. *Figures 12* and *14* use the same notation as previous figures. Notice that, in all cases, the cache is much smaller than the code size of each program inducing a realistic load on the cache use. First, both cache sizes of 16 Kb and 32 Kb provide the same benefit in number of hits whatever the task. Yet, the 8 Kb cache size seems to be too much small for the MCU0 program, 20% bigger than the MCU1 program where it produces the same measures. While the cache size should improve the WCET of the tasks, it seems to have very few effects on the inter-task analysis.

Lastly, we have computed the tasks WCET for each instruction cache configuration for the processor described in the previous section with a miss penalty of 10 cycles. Then we have improved the tightness of the WCET with the results of the hit count of our inter-task analysis by simply subtracting 10 cycles by each forgotten hit, counted as a miss by the conservative WCET analysis. Indeed, the CAT method just counts the number of misses and add the resulting penalty in cycles to the WCET.

Figure 15 shows the results of our computation. Horizontally, we have the list of tasks of MCU0 and MCU1 tasks while the vertical bars represent 100% of the WCET of each task. The black part of each bar represents the number of penalty cycles subtracted to the conservative WCET and, consequently, the gray part represents the WCET after fix. This figure shows that our approach can tighten the WCET estimate by 20% over approaches that does not consider the inter task cache analysis, and this is only the minimum improvement ratio. It also shows that the amelioration can reach 40% for some tasks.

In this experimentation, we have only used the results of the *MUST* analysis as it is the improvement that we provide and do not depend on the task flow execution. In the opposite, the *MAY* analysis provides an upper bound of the inter-task hits but it depends on the flow execution. Yet, this kind of information should be useful to provide a non-empty conservative state to intra-task instruction cache WCET analyses.

VI. Conclusion

In this paper, we have presented an analysis of a direct-mapped instruction cache behavior combining inter- and intra-task instruction cache analysis to estimate the number of cache hits due to task chaining.

A *MAY* and a *MUST* analysis are provided to compute the cache states between tasks. As the *MUST* analysis gives the lower and safe bound of the hit count due to task chaining, we have presented its results considering that the *MAY* analysis will mostly be useful to tighten the WCET intra-task estimation. Although we consider the *MUST* bound a lowest it is not insignificant, it can tighten the WCET estimate by an average of 20% over approaches that does not consider the inter-task cache analysis and the improvement can reach 40% for some tasks. It is very important to mention that the initial task WCET provided to our approach can be computed using any WCET computation methods bound to intra-task cache analysis. Our approach can be viewed as a lens filtering the cache hits due to task chaining to reduce the WCET estimation deduced from conservative methods.

There are many axes of improvement or extensions for our method. First, we are going to extend our method to set-associative caches. While it should more benefit from the persistence of blocks in the cache, it could make fuzzier the identification of blocks accessed at the entry of tasks. Another improvement would be to use the results of the intra-task analysis, that is, the cache state at the entry of each task, to improve the accuracy of instruction cache algorithms to perform intra-task WCET computation. As, in our current experiment, we have only exploited the *MUST* analysis, this would allow to benefit from the *MAY* analysis results too. Then, in our experimentation, we have considered a static task schedule that is not changed by the reduction of the WCET of the tasks. In a first time, we plan to experiment different scheduling policies with our inter-task analysis. Then, we want to examine the impact of the WCET reduction on the schedule and the back effect of the new schedule on the inter-task analysis results. We hope to find a strategy to converge toward a static schedule with a minimal overall WCET of the application. Moreover, it will be also interesting to experiment our analyses on dynamic scheduling policies.

As a last word , we believe that it should be interesting to adapt our approach to hardware devices with long time effects whose behavior depends on the task schedule such as branch prediction and data caches.

References

1] *Y.-T. S. Li, S. Malik. "Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software". 16th IEEE Real-Time Systems Symposium, pp. 298-307, 1995.*

2] *Y.-T. S. Li, S. Malik. "Cache Modeling for Real-Time Software:Beyond Direct Mapped Instruction Caches". 17th IEEE Real-Time Systems Symposium, 1996.*

3] C. Ferdinand et al. "Applying Compiler Techniques to Cache Behavior Prediction". *ACM SIGPLAN Workshop on Languages, Compilers, and Tools Support for Real-Time Systems,* pp. 37– 46, june 1997.

4] F. Mueller. "Timing analysis for instruction caches". Real-Time Systems, 18(2/3):209–239, May 2000.

5] Y. Tan, V. Mooney. " Integrated Intra- and Inter-Task CacheAnalysis for Preemptive Multi-Tasking Real-Time Systems". *8th International Workshop, SCOPES*, in Lecture Notes on Computer Science, LNCS3199, pp. 182–199, 2004.

6] Y. Tan, V. Mooney. " Timing Analysis for Preemptive Multi-Tasking Real-Time Systems with caches". Design, Automation and Test in Europe Conference and Exibition, Vol 2, pp. 1034 – 1039, 2004.

7] I.Wenzel, B.Rieder, R. Kirner, P. Puschner. "Automatic Timing Model Generation by CFG Partitioning and Model Checking". *Design, Automation and Test in Europe*, Vol. 1, pp. 606-611, 2005.

8] H. Cassé, C. Rochange, P. Sainrat. "An open Framework for WCET Analysis". *IEEE Real-Time Systems Symposium-WIP session*, pp. 13-16, 2004.

9] H. Cassé, C. Rochange, P. Sainrat. "OTAWA, a framework for experimenting WCET computations". *3rd European Congress on Embedded Real-Time*, 2005.

10] F.Nemer, H. Cassé, P. Sainrat, J.P. Bahsoun, M. De Michiel. PapaBench: a free real-time benchmark. *6th WCET Workshop*, 2006.

11] F. Singhoff, J. Legrand, L. Nana. AADL resource requirement analysis with CHEDDAR. *LYSIC/EA 3883*.

12] P. Feiler, D. P. Glush, J. J. Hudak, B. A. Lewis. "Embedded System Architecture Analysis Using SAE AADL", 2004.

13] A.V. Aho, R. Sethi, J.D. Ullman, "Compilers: Principles, Techniques, and Tools", Addison Wesley, 1986.

14] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems.In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS02)*, 2002.

15] J. Staschulat, R. Ernst. Cache Effects in Multi Process Real-Time Systems with Preemptive Scheduling. Technical report, IDA, TU Braunschweig, Germany, November 2003.