Achieving Timing Composability with Measurement-Based Probabilistic Timing Analysis

Leonidas Kosmidis[*,†], Eduardo Quiñones[†], Jaume Abella[†], Tullio Vardanega[‡], Francisco J. Cazorla[†,§]
[*]Universitat Politècnica de Catalunya, Spain
[†]Barcelona Supercomputing Center, Spain
[‡]University of Padua, Italy
[§]Spanish National Research Council (IIIA-CSIC), Spain

***Abstract*—Probabilistic Timing Analysis (PTA) allows complex hardware acceleration features, which defeat classic timing analysis, to be used in hard real-time systems. PTA can do that because it drastically reduces intrinsic dependence on execution history. This distinctive feature is a great facilitator to time composability, which is a must for industry needing incremental development and qualification. In this paper we show how time composability is achieved in PTA-conformant systems and how the pessimism of worst-case execution time bounds obtained from PTA is contained within a 5% to 25% range for representative application scenarios.**

I. Introduction

Embedded systems industry [1] faces a steadily growing demand for greater computing power and stricter cost containment [2]. One response to this challenge is to use less hardware with more performance-aggressive capabilities that can accommodate more complex software functions of mixed criticality. But going this way is far from immediate.

Placing multiple functions, possibly at different criticality levels, on the same hardware, requires *space isolation* and *time isolation* [3]. The former prevents any data-related misbehaviour of one function from affecting the data sources of other functions. The latter, which is the focus of this paper, ensures that the worst-case execution time (WCET) bound determined for one function is guaranteed (hence can never be exceeded) in the face of other functions competing for execution on the same processor. This interpretation of time isolation directly follows from the property known as *time composability* (TC). Achieving both forms of isolation enables a drastic reduction of development costs as each subsystem can be independently developed and then incrementally integrated and qualified in the system without risks of regression at system level.

Classic timing analysis and schedulability analysis assume time composability in the software components that constitute the system. Their assumption however is safeguarded by conservative and pessimistic assumptions, which fatally defeat the industrial goal stated above.

State-of-the-art static timing analysis techniques are intrinsically limited by the complexity of constructing a sufficiently accurate model of the hardware and of the software executing on top of that hardware. Any inaccuracy or lack of the required knowledge about the hardware timing or the software execution behaviour may have an inordinate impact on the tightness of the resulting WCET estimates. As the hardware becomes increasingly complex and software functions increase in number and size, building accurate models of the hardware and software to determine tight WCET bounds becomes prohibitive, if at all feasible [4] [5] [6].

Probabilistic Timing Analysis (PTA) [7] [8] has recently emerged as an alternative to classic timing analysis. PTA provides WCET estimates with an associated probability of occurrence, what is known as probabilistic WCET (pWCET) estimates. PTA extends to timing correctness the notion of probability of failure, which is an acquired concept in the analysis of reliability. PTA aims to obtain pWCET estimates for arbitrarily low probabilities (e.g., in the region of 10^{-20} per hour of operation) that meet the requirements on the probability of hardware failures. The key benefit of embracing PTA is that execution timing becomes dramatically less dependent on execution history, with drastic reduction in the amount of information required to obtain tight WCET estimates in comparison to other timing analysis approaches. PTA can be applied either in a static (SPTA) [8] or measurement-based (MBPTA) [7], though the latter is considered in this paper as it is much easier to use for industry.

MBPTA is a variant of PTA based on statistical reasoning, which requires that the random variables describing the events to be analysed – measurements of end-to-end runs – are independent and identically distributed (i.i.d.). Independence is obtained by opportune statistical techniques. Identical distribution is best ensured by construction; otherwise it is confirmed a posteriori by specific statistical tests.

MBPTA has been proven to be a viable method to perform timing analysis of programs in hard real-time systems. However there is still a lack of understanding of what *PTA-conformance* – the property of a system to produce i.i.d. timing events – offers in the way of time composability. The question is how MBPTA tight yet safe WCET bounds can be in comparison to those obtained by state-of-the-art techniques.

The contributions of this paper are as follows:

1) We identify how time composability can be achieved in probabilistic systems while keeping pWCET estimates tight. We focus on the effect of cache memories [9], as they are the typifying example of a hardware acceleration feature known that severely challenges classic WCET analysis.
2) We describe the information required from software components to be able to time compose their pWCET estimates. In particular, we show that the *reuse distance*[1] of memory accesses is enough to fully characterise the time composability properties of a software component.
3) We illustrate how to compose pWCET estimates in a processor set-up with complex cache configurations and

[1]Reuse distance stands for the number of memory accesses in-between two consecutive accesses to a particular address.

provide methods to generate pWCET estimates suitable for composition in PTA systems. We demonstrate that smart compositions allow using tighter pWCET estimates of the components.

We run our experiments on a simulation environment in which we model a pipelined processor that features data and instruction caches. The results we present show that PTA makes pWCET estimates attractive to time composability. In particular, we observe up to 25% reduction in our pWCET bounds with respect to the WCET obtained by the flushing processor state prior to program execution, which is the standard industrial practice to attain time composability.

II. Background

A. Incremental qualification

For many real-time embedded systems, the hardware and software components of the system have to be developed in parallel and integrated incrementally. A key design principle to contain verification costs in Integrated Architectures such as AUTOSAR for automotive [10] and Integrated Modular Avionics (IMA) for avionics [11] is to secure the possibility of *incremental qualification*, whereby each software component can be subject to verification and validation – including timing analysis – in isolation. This goal is achieved by guaranteeing that no interaction occurs between isolated functions, in time and space, in their sharing of execution resources. At functional level, this translates into providing for space isolation, such that no misbehaving function may corrupt the data used by other functions. At timing level, this requires *time composability*, such that the timing behaviour of a function (in terms of its known lower and upper bounds) is not affected by the execution of other functions, whereby software functions enjoy time isolation. This property determines that the timing behaviour of an individual component does not change in the face of composition with other components.

B. Probabilistic Timing Analysis

PTA has emerged as an alternative to current timing analysis techniques [8] [7]. Both SPTA [8] and MBPTA [7] provide a cumulative distribution function, or pWCET function, that upper-bounds the execution time of the program under analysis, guaranteeing that the execution time of a program only exceeds the corresponding execution time bound with a probability lower than a given target probability (e.g., 10^{-15} per hour). The probabilistic timing behaviour of a program (or an instruction) can be represented with Execution Time Profiles (ETP). An ETP defines the different execution times of a program (or latencies of an instruction) and its associated probabilities. That is, the timing behaviour of a program/instruction can be defined by the pair of vectors $(\overrightarrow{l}, \overrightarrow{p}) = \{l_1, l_2, ..., l_k\}\{p_1, p_2, ..., p_k\}$, where p_i is the probability the program/instruction taking latency l_i.

With MBPTA, used in this paper, given a set of R runs of a program, one could compute the pWCET function of the program as the *exceedance cumulative distribution function* (ECDF). ECDF provides the probability of occurrence of each of the observed execution times based on the histogram of execution times. Unfortunately, ECDF can only provide execution time estimates for probabilities down to $\frac{1}{R}$ in the best case. For smaller probabilities, techniques such as *Extreme Value Theory* (EVT) [12] [7] are used to project an upper-bound of the tail of the exceedance function, enabling MBPTA techniques to provide pWCET estimates for *target probabilities* largely below $\frac{1}{R}$.

PTA techniques require that the events under analysis, program execution times for MBPTA and instruction latencies for SPTA, can be modelled with *i.i.d.* random variables [8]: two random variables are said to be independent if they describe two events such that the occurrence of one event does not have any impact on the occurrence of the other event. Two random variables are said to be identically distributed if they have the same probability distribution. The existence of ETP ensures that each execution time of the program (for MBPTA) or instruction (for SPTA) has an actual probability of occurrence: this is a sufficient and necessary condition for the execution time behaviour to be modelled probabilistically [8].

Timing events are given a i.i.d. nature by ensuring that all lower-level events that may contribute to the jitter observed at the granularity level of timing analysis (e.g., hits or misses in cache for memory accesses) have true probabilities of occurrence. To meet this requirement the timing behaviour of some processor resources is to be time randomised [8] [7] [9].

III. Time Composability

Time composability applies to the interaction between the Operating System (OS) and the user-level application or between the user level application themselves.

Regarding the interaction between applications and the OS, in [13] the authors present the design of a time composable OS. In the cited work, the latency of every system call that may affect the pWCET of application-level program units is designed to be constant and to not perturb the state retained in the processor at the time of the call: in this manner the execution of the OS has no effect on pWCET bounds of the caller program unit.

In this paper we focus on time composability (TC) at application level. The TC we seek (i) allows tight pWCET estimates to be obtained for program units and (ii) incurs affordable design, implementation and verification costs. We dismiss approaches to TC that hinder functional design and scalability: for example, the use of static (table-driven) scheduling for the execution of program units and making pessimistic allowances for the execution slots assigned to them.

Next we describe the typical structure of industrial-quality user-level hard real-time applications and then discuss how state retention in hardware, with focus on cache memories, and in software components threatens TC. Finally, we state the problem and the assumptions on top of which we build our case for probabilistic time composability.

A. Software Structure of Real-Time Functions

We regard the software design to include a collection of independent *main procedures* each of which in turn contains a number of *inner procedures*. Main procedures are called indefinitely as long as the system is operational according to some system level schedule (see Algorithm 1). We consider inner procedures to be the unit of software development and verification. Integration and validation occur incrementally.

The system-level schedule determines how the execution of main procedures interleave with one another. The Control-Flow Graph of each main procedure determines the way inner

Algorithm 1: Static schedule of main procedures with run-to-completion execution semantics (top) and Example control flow of a $Main\_procedure_i$

```
void Main_procedure_i() {
Inner_procedure_{i,1}()
Inner_procedure_{i,2}()
Inner_procedure_{i,1}()
for (j = 0; j ≤ 5; j++) do
    Inner_procedure_{i,3}()
    Inner_procedure_{i,4}()
    Inner_procedure_{i,2}()
end for
}
```

```
while true do
    Main_procedure_1()
    Main_procedure_2()
    ...
    Main_procedure_n()
end while
```

procedures are called and what interleaving occurs between any two successive executions of any of them. However, as inner procedures are the unit of parallel development hence also the Unit of Composition (UoC), their exact internal structure is only known late in the development process.

We term *Program Unit* the software component on which pWCET analysis is to be performed. This can be at the granularity of either the main procedure or the inner procedure. The pWCET value that matters for system-level scheduling is that of main procedures, which in turn results from the pWCET bounds determined for its inner procedures. It is therefore safe to assume that pWCET analysis operates on inner procedures and proceeds upward from them.

B. Problem Statement and Assumptions

The main factor affecting TC in the execution of a UoC is the state retained in the processor hardware (e.g., cache contents) and in the software (e.g., data on which path decisions are dependent). In this paper we ignore software state issues because MBPTA is capable of handling the impact of software state in the determination of pWCET estimates [7].

We concentrate on the processor state whose retention may affect TC. Given a UoC B representing an inner procedure to which PTA is to be applied to obtain a pWCET estimate, we assume that (i) a time-composable OS is used such that the latency of each OS service call is constant and its execution has no effect on the state retained in the target processor [13]; and (ii) run-to-completion semantics is granted for all UoC so that the interference effects resulting from preemption do not need to be accounted for in the analysis. Assumption (ii) matches current IMA practice in civil avionics [14].

Let us now consider two consecutive execution instances of B. Let us call them B_p and B_q respectively, with the subscript representing the ordinal execution number of the UoC instance, with $q > p$, so that B_p precedes B_q. The amount of useful processor state that B_q can reuse from B_p and how much benefit this reuse can cause on the execution time of B_q, and thus on its pWCET, depends on: (1) How much internal reuse B makes, which we call *intrinsic reuse* and which depends on the size of B's working set. (2) The execution 'duration' of B. The longer the execution the lower the relative effect that different initial conditions can have on B_q. (3) The size of the cache(s) that retains B_p's state in hardware. And (4) what portion of B's working set is not evicted by the code executed between B_p and B_q. We term that foreign code *disturbing code* and its effect *state disturbance*.

Standard analysis of the Control-Flow Graph of the main procedure to which B belongs allows determining when different inner procedures are called (see Algorithm 1).

An easy yet pessimistic way to attain time composability for B is to compute its pWCET assuming that all processor state is flushed prior to its every execution. This assumption has the advantage that it makes no assumption on the code executed before B is called. Yet, as we show later in Section V, this approach to time composability may introduce significant degradation in both average and guaranteed performance (pWCET) for B. Achieving time composability in this manner allows too little load in the system, which goes counter the industrial need presented in Section I.

We want to allow useful processor state retention to be considered in the determination of tight pWCET estimates and want to do so in a manner that achieves time composability and that is economically viable to implement. The particular problem we solve in this paper can be formulated as follows: *provide hardware/software mechanisms such that the pWCET estimate for a UoC stays valid in the face of composition with any other UoC during system integration. These hardware/software mechanisms must not require any change in existing PTA techniques, whether MBPTA or SPTA, applied to PTA-conformant processor architectures.*

IV. Probabilistic Time Composability

Previous studies [15] show that when flushing the processor state, programs recover their core (pipeline) state – including the branch predictor state – in a few hundred cycles. This is not true for caches instead: depending on the cache size in fact, it may take an amount of time several orders of magnitude higher than core state recovery, before the working set is restored in the cache after a flush. For this reason in this work we focus on on-chip cache resources and instead simply flush the core (pipeline) on every entry to a UoC.

We assume time-randomised caches [7] [9]. Time randomised caches ensure that there is a probability for each cache access to hit or miss in cache, as needed by PTA, so its timing behaviour can be modeled with an ETP: $ETP(i) = \{t_{hit}, t_{miss}\}, \{p_{hit}, p_{miss}\}$, where t_{hit}, t_{miss} express the latency in case of a hit and an miss respectively, and p_{hit}, p_{miss} the associated probabilities. This can be achieved for both set-associative and fully-associative caches implementing random placement and replacement policies [9] [7].

On time randomised caches, on every access that incurs a miss, a cache line is randomly evicted from cache as the new location in which the fetched line is placed, i.e the cache set and the cache way, is randomly selected. [9] shows that given any two accesses to a given address $@_A$, the number of evictions occurring between those two accesses affects the hit probability of the second instance of $@_A$. In fact, the higher the number of misses, the lower the hit probability of the second occurrence of $@_A$. The reason is that the probability of selecting (and so evicting) the cache set and the cache way in which $@_A$ resides increases.

We note that the number of evictions is upper-bounded by the number of accesses occurring both accesses to $@_A$. However, upper-bounding the number of misses of the disturbing code with the number of cache accesses is overly pessimistic

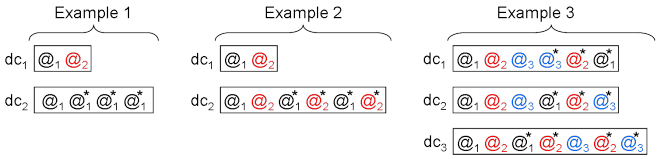


Fig. 1. The effect of the number of accesses, the number of unique addresses and reuse distances. Accesses marked with * have non-infinite reuse distance.

since only a subset of the accesses are misses. For the specific sake of time composability, we seek a set of metrics me that characterise a program unit such that given two disturbing codes dc_1 and dc_2, we can prove that the cache ageing caused by dc_1 is higher than that caused by dc_2 if dc_1 produce worse effects than dc_2 on the same program unit for all me. That is, for $1 \leq i \leq M$ where M is the number of metrics to consider of a program unit, if $me_i^1 \geq me_i^2$ the effect of dc_1 is worse than the effect of dc_2. Under this situation, if we obtain a pWCET bound for B_q under the composition scenario $B_p dc_1 B_q$ we know that such pWCET bound is a true upper-bound value for the $B_p dc_2 B_q$ composition scenario.

In Figure 1 we show several examples that illustrate which metrics (me) we use.

a) Example 1: A large number of accesses of a given disturbing code does not guarantee that it is an upper-bound for any other disturbing code with fewer accesses. In particular dc_1 performs only 2 accesses that produce 2 misses, whereas dc_2 performs 4 accesses that produce just 1 miss since the last 3 accesses are guaranteed to hit. The number of unique addresses[2] is the critical factor here since the first access to any given address will miss and produce an eviction, whereas other accesses may miss (marked * in the figure) or hit if the data they look for are still in cache.

b) Example 2: While the number of unique addresses matters, the number of accesses also does. In particular, dc_2 will produce at least as many evictions as dc_1. In fact, both dc_1 and dc_2 can evict up to 2 of the cache lines left in cache by B_p. We deepen on this issue later in this section.

c) Example 3: Reuse distance also matters. In particular, dc_1 and dc_2 access the same addresses the same number of times, but in a different order. However, reuse distances are different. Their respective reuse distances are $(\infty, \infty, \infty, 0, 2, 4)$ and $(\infty, \infty, \infty, 2, 2, 2)$ respectively[3], which have different impact on the hit probability of accesses. It is therefore unclear how to determine which disturbing code upper-bounds others. Similarly, it is unclear whether dc_3 bounds dc_1 and dc_2 even if it has a larger number of accesses and the same number of unique addresses.

Reuse distance. Given a sequence of accesses to cache $@_A @_B @_C @_D ... @_A$, the hit probability of the second instance of $@_A$ depends on the *number of intermediate accesses occurring in between the first and the second occurrence of* $@_A$ *and the reuse distance of each of those accesses*: $rd_B, rd_C, rd_D, ...$ We define the reuse distance for an access $@_A$ as the number of memory accesses occurred since the last access to the same address. For instance, in dc_1 in Example 3 of Figure 1, the reuse distance is 4 for the second instance of $@_1$. The reuse distance of an access determines its hit probability: the higher the reuse distance, the lower the hit probability of that access. In the extreme case, when the reuse distance an access is infinite its hit probability is 0. We call such an access a *unique access*. The hit probability of the second instance of $@_A$ is inversely proportional to the miss probability of intermediate accesses, which in turn depends on the reuse distance of those accesses.

[2]Unique addresses are those remaining once repetitions are removed.

[3]∞ denotes the first access to a given memory location in the sequence,

Algorithm 2: Checking whether disturbing code dc_1 upper-bounds another disturbing code dc_2 for MBPTA with EoM

```
Check if dc1 upper-bounds dc2
C = set of caches in the processor
for all c_i ∈ C do
    r1 = reuse distances for dc1 in c_i
    r2 = reuse distances for dc2 in c_i
    if |r2| > |r1| then
        return false
    end if
    r1sort = sort r1 from higher to lower
    r2sort = sort r2 from higher to lower
    for all r2sort_j ∈ r2sort do
        if r2sort_j > r1sort_j then
            return false
        end if
    end for
end for
return true
```

Determining the reuse distance of two subsequent executions of a UoC B allows obtaining the pWCET estimates of the second execution of B such that it benefits from the cache state left by the previous execution of it. These pWCET estimates must be obtained making the least assumptions on the disturbing code executed in between those instances of B.

Hence, our first approach considers the reuse distances of the instruction and data accesses of the disturbing code: me = $\{rd_i, rd_d\}$. Given two disturbing codes, dc_1 and dc_2, we say that the former produces worse interference than the latter if each instruction and data access of dc_2 can be paired up with an instruction and data access respectively of dc_1 with higher reuse distance. The rationale behind this approach is that the higher the reuse distance of an access, the higher its miss probability thus the higher the probability of an eviction, which reduces the survivability of cache contents. In other words, if dc_1 performs at least as many instruction and data accesses as dc_2 and the miss probabilities for dc_1 accesses are higher than those for dc_2, dc_1 upper-bounds dc_2 cache ageing. An easy mechanism to check how the reuse distances for data and instructions for dc_1 upper-bound their counterparts for dc_2 is shown in Algorithm 2.

Below we show some examples of reuse distance vectors for dc_1 and dc_2. In each case we show when dc_1 upper-bounds dc_2. For this example, we only consider data accesses, but the same considerations apply to instruction accesses.

- dc_1 upper-bounds dc_2: $rd_1 = \{7, 5, 3, 2\}$, $rd_2 = \{6, 5, 2\}$. $7 > 6$, $5 \geq 5$, $3 > 2$, $2 > \emptyset$. Hence, dc_1 upper-bounds dc_2.
- dc_1 does *not* upper-bound dc_2: $rd_1 = \{9, 8, 7, 0\}$, $rd_2 = \{1, 1, 1, 1\}$. $9 > 1$, $8 > 1$, $7 > 1$, $\emptyset < 1$. No reuse distance of dc_1 is paired up with the last one of dc_2, so dc_1 cannot be proven to upper-bound dc_2.

Although the approach described so far is conceptually applicable and can produce tight upper-bound values, modelling the reuse distances of all the disturbing code that may possibly execute between B_p and B_q is too complex. It would be interesting, therefore, to find alternative approaches to bound the effect that a given dc can cause on different instances of

a UoC B, which can reduce the characterisation information that the user has to provide about the disturbing code. Next we present one such approach.

Using the number of unique accesses One way to achieve tight and safe bounding consists in having the user provide the number of unique accesses u_i, u_d that the *dc* to be bound may perform. Let's assume that dc_1 performs the following sequence of accesses $@_A@_B@_A@_B$, so the number of accesses (N) is 4 and the number of unique accesses (u) is 2. Next, we show that such a program can evict at most u cache lines of the contents stored prior to its execution. The first time $@_A$ is accessed it misses in cache evicting data of B_p, which we call *prior data*. The first access to $@_B$ will also miss but it can evict data of B_q or $@_A$. The second access to $@_A$ will only cause a miss if $@_B$ evicted A and no element of B_p is in cache. Analogously, the second access to $@_B$ will only evict data in cache from B_p only if the first instance of $@_B$ evicted the first instance of $@_A$ and the second instance of $@_A$ evicted the first of $@_B$. *Overall, a disturbing code of N accesses and u unique accesses can evict at most u elements that were in cache prior to its execution*. The reuse distance of the N accesses will determine the actual probability that u prior data are evicted. For example, the sequence $@_A@_A@_A@_B@_B@_B$ that has the same N and u that the sequence $@_A@_B@_A@_B@_A@_B$ is much less likely to evict u prior data.

Given a dc_1 with u_i, u_d unique accesses, we want to synthetically generate a disturbing code dc_2 whose probability of evicting u_i and u_d prior data in the instruction and data cache respectively is higher than for any other dc_1 with u_i, u_d unique accesses regardless of the number of times they are accessed and the reuse distance of those accesses.

Note that the user is only asked to provide u_i, u_d for the disturbing code, which can be easily obtained by means of profiling at the integration stage.

Next, we must derive how a dc must look like so that it bounds a program whose number of unique instruction and data cache lines accessed is u_i, u_d. To that end, it can be proven that for a cache with S entries, the number of distinct entries evicted (dee) after l random evictions is as follows [16]:

$$dee = \left[1 - \left(1 - \frac{1}{S}\right)^{l}\right] \times S \tag{1}$$

In other words, if we want to evict at least u distinct entries so that $dee = u$, the number of evictions required can be obtained as follows:

$$l = \left\lceil \frac{\log\left(1 - \frac{u}{S}\right)}{\log\left(1 - \frac{1}{S}\right)} \right\rceil \tag{2}$$

Therefore, a dc causing at least l evictions bounds the impact of any program with up to u unique accesses.

A. Software support

With MBPTA, we make observation runs on the target system that capture the effect of disturbing code on the UoC of interest. Hence, disturbing code cannot be solely a conceptual artefact but it has to be concretely created at either hardware or software level.

We need to implement simple programs, which we term *micro-benchmarks*. A micro-benchmark produces a sequence of accesses to each cache memory such each access addresses a different cache line. As obtaining that effect for data and instruction caches simultaneously is difficult, a first part of the micro-benchmark can produce evictions for data and a second part of it can cause evictions for instructions. The number of evictions required in each cache is determined with equations 1 and 2. Data evictions simply require a loop with a load instruction whose stride is equal or higher than the largest cache line in any data cache. For instance, if we use a data cache with 32-byte cache lines and a data TLB (translation look-aside buffer) with 1KB pages, performing max_d loads with a distance of at least 1KB ensures that no reuse occurs in any of the data caches across those max_d accesses. A similar effect can be produced for instructions by executing branches with a stride equal or higher than the largest cache line in any instruction cache.

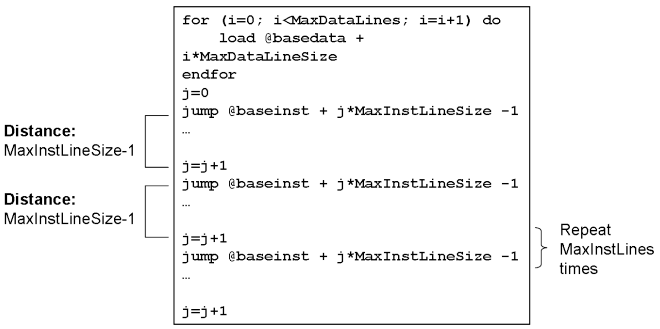


Fig. 2. Example of a micro-benchmark

An example of how to build such a micro-benchmark is shown in Figure 2. First, a loop performs all the needed data cache evictions. The micro-benchmark code iterates as many times as data cache evictions are required, max_d, ($MaxDataLines$ in the figure) accessing data with a stride matching the maximum data cache line size ($MaxDataLineSize$). Note that data space must be allocated to guarantee that the micro-benchmark does not make any access beyond the program bounds. The second section of the micro-benchmark consists of linear code that jumps as many times as instruction cache evictions are required, max_i, ($MaxInstLines$ in the figure) with a stride matching the maximum instruction cache line size ($MaxInstLineSize$).

In order to produce composable pWCET estimates of $Main\_procedure_i()$ in Algorithm 1 therefore, we must analyse each instance of $Inner\_procedure_{i,j}$ against its disturbing code. For instance, if we focus on the second instance of $Inner\_procedure_{i,2}$, its disturbing code is $Inner\_procedure_{i,3}$ and $Inner\_procedure_{i,4}$.

The pWCET estimate for the second instance of $Inner\_procedure_{i,2}$ is determined by MBPTA [7]) as follows. We run $Inner\_procedure_{i,2}$ alone, then a particular instance of the micro-benchmark and finally the p$Inner_procedure_{i,2}$ again measuring its execution time in that last run. To simplify that process, several values are chosen for max_d and max_i for the micro-benchmark are used. The higher the number of combinations considered, the tighter the pWCET, but for more experiments to run.

With that data, MBPTA produces the pWCET estimate considering the interference effect of the assumed disturbing code. This process is repeated for all disturbing codes considered. At integration time the user must select the lowest pWCET for those scenarios where the micro-benchmark considered bounds the real disturbing code, such that $max_d \geq l_d$ and $max_i \geq l_i$, where l_i and l_d are obtained with equation 2.

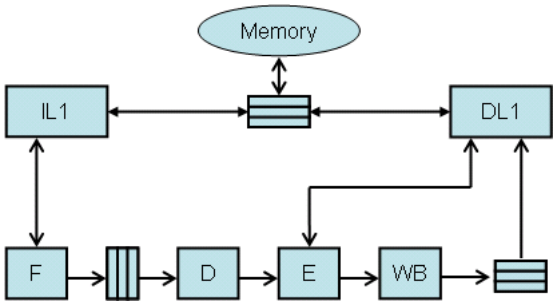


Fig. 3. Processor architecture

This approach to timing composability has the key property of being *oblivious to the particular location in memory where the data and instructions of the disturbing code are* because probabilistically analysable caches such as random placement and replacement caches break the structural relation between the address and location of cache contents.

V. Experimental results

This section evaluates our Time Composability approach. First we introduce the experimental framework and show how cache size and disturbing code characteristics impact the survivability of cache lines. Then, pWCET estimates are obtained for several relevant benchmarks under different scenarios.

A. Experimental Framework

We use SoCLib [17] with PowerPC binaries [18] to implement a cycle-accurate processor simulator. In particular, we implement a 4-stage pipelined in-order core architecture (fetch, decode, execute, write-back) similar to the LEON4 processor [19], in use at the European Space Agency. The processor architecture is depicted in Figure 3. The memory hierarchy consists of separate instruction (IL1) and data (DL1) caches, and main memory. Both instruction and data caches are 8-way set associative with 16-byte lines. The hit latency for all caches is 1 cycle. The miss latency is set to 100 cycles, including the time to access main memory. This value is arbitrary but it serves the purpose of producing a significant jitter without being unrealistic.

We used a sample of Mälardarlen benchmarks [20], which are commonly used in the hard real-time community to evaluate WCET analysis tools and methods. Of those, we used: bs (BS), crc (CRC), qsort (QSO) and select (SEL). BS and CRC illustrate extreme cases with very large reuse and almost no reuse across executions respectively. QSO and SEL correspond to intermediate cases with moderate reuse across executions.

We obtained pWCET values using the MBPTA method described in [7]. Analogously, we used the methods reported in that work to determine the minimum number of runs and to prove that all execution time traces fulfil the required independence and identical distribution (i.i.d.) properties. The minimum number of runs we needed was always below $1,000$, and all execution time traces passed the i.i.d. tests successfully.

B. Results

1) Survivability: We saw earlier that the potential reuse that B_q can make of the data and instruction state left by B_p is inversely proportional to the size of the disturbing code. To illustrate this point, we assume a cache in which after the execution of B_p all cache lines have reuse distance 0. Figure 4

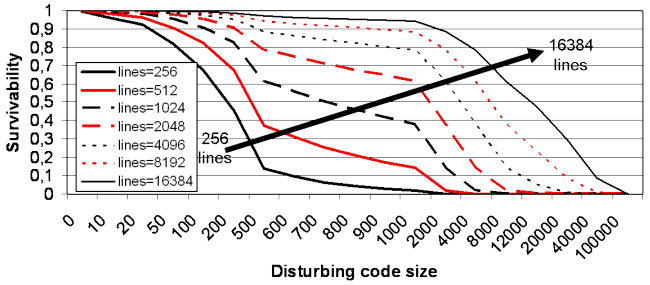
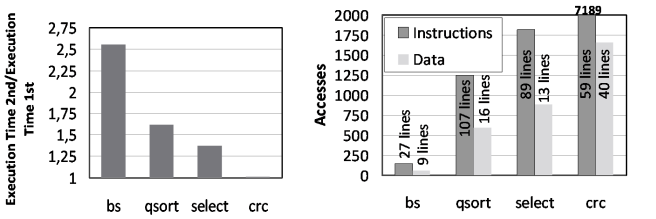


Fig. 4. Survivability as a function of the number of unique accesses in the disturbing code for caches with different number of lines.



(a) pWCET speedup (b) Instruction and Data access count

Fig. 5. Characterisation of the Mälardarlen benchmarks used

shows the survivability of each line for a range of different cache sizes as we increase the number of unique addresses (and therefore also accesses) to the cache in the disturbing code. The cache size range corresponds to the typical number of cache lines for L1 and L2 caches.

Survivability with small-sized caches is low if all accesses of the disturbing code cause evictions. Conversely, for relatively large caches (e.g., in the order of some thousands of cache lines, which match the size of L2 caches) survivability increases noticeably and data in cache can be reused across execution instances. This is true even if the disturbing code accesses thousands of different cache lines, which is very unlikely for our target scenarios where small inner procedures are executed several times within their enclosing procedure (cf. Section III).

2) Characterising the maximum benefit of TC: The potential benefit we can get with our approach to time composability depends, in addition to the intrinsic reuse of the application as shown in previous section, on the number of instruction and data accesses the functions in the disturbing code have.

Figure 5 shows the instruction and data accesses of the selected Mälardarlen benchmarks as well as the execution time reduction due to data and instruction reuse. The number of different cache lines accessed by each benchmark are shown on top of the corresponding columns for data and instructions. The experiments show that BS achieves the highest pWCET reductions ($\frac{pWCET\ empty\ cache}{pWCET\ zero\ disturbing\ code}$) since the number of accesses per cache line is low, hence the relative benefit of finding those lines in cache is significant. QSO and SEL show lower yet significant pWCET reductions because the relative impact of the lines reused across executions is lower. CRC shows the smallest reductions due to the relative low impact of the 99 (59+40) potential extra hits in a program performing almost 9,000 (7,189+1,655) total cache accesses.

3) TC of pWCET estimates: As mentioned in Section IV, flushing the processor state prior to the analysis of the UoC is the easiest way to TC. This solution prevents the execution time of the UoC under analysis from being affected from

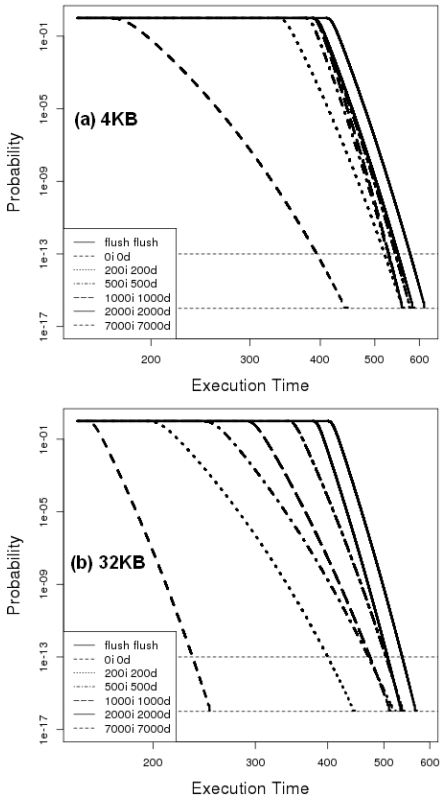


Fig. 6. pWCET estimates obtained with MBPTA for different (u_i, u_d) values for the *bs* benchmark

previous history of execution, which makes its WCET bound time composable, but at the cost of unnecessary pessimism. We refer to this approach as (*flush*,*flush*), meaning that we flush the instruction and data caches prior to execution.

Figure 5(b) shows that instruction access counts vary from a few hundreds to around $7,000$ while the data access counts range between a few dozens to almost $1,700$. Based on these values we use 6 different sets of values for u_i and u_d to build the micro-benchmark as described in Section IV: (u_i, u_d) = (*100*,*100*), (*200*,*200*), (*500*,*500*), (*1000*,*1000*), (*2000*,*2000*) and (*7000*,*7000*).

Figure 6 shows the pWCET distribution for BS for various micro-benchmark settings. pWCET improvements diminish as the number of evictions performed increases, especially for small caches. Larger caches (e.g., 32KB) still provide significant pWCET improvements over the empty cache case despite the large number of evictions. Figure 7 details the results for all benchmarks for an exceedance probability of 10^{-13} per run. pWCET reductions are significant for some benchmarks as long as the number of evictions per cache does not exceed 1,000 evictions. Beyond that point benefits quickly diminish. However, given the context where probabilistic time composability is exploited (see Section III), the disturbing code consists of small inner procedures.

Our representative benchmarks show that the number of unique cache lines accessed is 70 instruction lines and 20 data lines on average (see Figure 5 (b)).

By using the method in Equation 2 and a 4KB cache, the number of evictions required in instruction and data caches to safely upper-bound the effects of disturbing code would be 82 and 21 for instruction and data caches if only one inner procedure with 70 and 20 unique instruction and data lines accessed respectively is executed in between two consecutive

	4KB cache				32KB cache			
evicts	bs	select	qs	crc	bs	select	qs	crc
(*0, 0*)	31.8	35.5	28.3	4.3	56.5	33.4	27.0	6.4
(*200,200*)	10.0	10.7	2.4	0.4	25.6	20.0	17.9	4.4
(*500,500*)	8.6	4.2	2.2	0.4	16.0	8.5	11.6	1.5
(*1,000,1,000*)	8.4	3.8	0.9	0.3	12.0	5.4	9.0	0.6
(*2,000,2,000*)	5.9	2.4	0.6	0.1	6.3	2.3	1.9	0.4
(*7,000,7,000*)	4.8	1.9	0.1	0.0	5.6	0.5	0.9	0.4

Fig. 7. pWCET percentage improvement (reduction) of (u_i,u_d) against (*flush*,*flush*) for 10^{-13} cutoff probability

cutoff probability	($flush$,0) icache flushing	(0,$flush$) dc flusing
10^{-13}	5.9%	22.8%
10^{-16}	6.4%	19.9%

Fig. 8. Effect of instruction data and caches flushing (4KB cache)

executions of the UoC under analysis. If the number of such inner procedures increases, so does the number of evictions required. For instance, two such inner procedures would require 203 and 44 instruction and data unique accesses to bound their effect. As the number of inner procedures grows, their effect also grows, thus decreasing the pWCET improvement due to time composability. For instance, after 5 inner procedures we would not expect any cache line to be still in the instruction cache based on the method in equation 2 so we should simply assume that the instruction cache has been flushed. Conversely, the data cache would still provide some pWCET improvement as only 127 evictions would be needed.

The results are much better for the 32KB cache as it mitigates the impact of larger disturbing pieces of code. For instance, if only one inner procedure is executed in between two consecutive executions of the UoC under analysis, the number of instruction and data evictions required would be 72 and 21 respectively. Five inner procedures would only require 384 and 103 evictions respectively.

In summary, if the number of inner procedures between consecutive executions of the UoC under analysis is below 10, then the pWCET improvements will be in the range dictated by the (*200*,*200*) or (*500*,*500*) cases. Thus, significant pWCET improvements of 5%-10% can be observed for some inner procedures if caches are small (e.g., 4KB) and 10%-25% if caches are larger (e.g., 32KB) as shown in Figure 7.

4) Breaking down TC benefits across data and instruction caches: Flushing instruction and data caches has different effects on pWCET estimates. Figure 8 shows the average relative pWCET reduction across all benchmarks when only one cache is flushed, with respect to the pWCET estimate when both caches are flushed. As we can observe, when the instruction cache is empty, the pWCET reduction is quite small, in the range of 6%. On the other hand, if instruction cache contents are preserved, even in the complete absence of data in the data cache, the pWCET reduction is around 20%.

Thus, the maximum benefit is provided by the reuse of instructions, which is explained by the structure of the Mälardarlen benchmarks and is the typical behaviour of inner procedures in our context. In particular, their code consist of linear code with few small loops. Thus, the number of instruction cache lines fetched is relatively large (quite linear code) and cold misses account for most of the misses. This opens the door to a significant reuse across executions. Data sets are relatively small and highly reused inside inner procedures. Thus, there are fewer data cold misses, which decreases the relative impact of data reuse across executions.

VI. Related Work

In current industrial practice Time Composability of WCET bounds is attained by flushing prior every execution all processor resources that may introduce jitter. Based on such bounds, each application service is assigned a time budget [21] [10], which ensures exclusive access to shared resources and identifies the point in time in which processor resources needs to be flushed.

Several works [22]–[25] achieve time composability in the presence of processor shared resources (e.g., memory controllers) by computing the maximum delay a request can suffer because of interferences when accessing to them. Such maximum delay is used to compute trustworthy WCET bounds.

The use of specialised micro-benchmarks [26]–[28] has also been considered to stress each processor resource close to its worst-case scenario, so that the highest interferences are observed. Their impact in execution time is used to compute WCET bounds, achieving some degree of time composability.

Previous works achieve time composability by considering the worst-case scenario, i.e. by either flushing the processor resource or forcing the maximum response time of processor requests or reproducing a processor resource access close to the worst-case situation. To the best of our knowledge, this is the first work that benefits from previous processor state to reduce the pWCET while still guaranteeing an economically viable attainment of Time Composability.

VII. Conclusions

PTA has been proven to enable the use of complex hardware acceleration features such as caches while providing tight guaranteed execution time bounds (WCET). The Time Composability properties of a PTA-conformant system however were not yet understood. As Time Composability is needed by embedded systems industry to enable incremental development, failing to provide arguments about how software components can be time composed defies the benefits of PTA.

In this paper we have shown how program units executed on a PTA-conformant processor can be time composed. We have focused on the challenges to time composability caused by caches. In particular, we have considered time-randomised cache memories . We have shown that the amount of information required to characterise the disturbing effect of foreign code execution, which is needed to make the program unit time composable, is relatively low: the number of unique data and instruction accesses of the disturbing code.

This is in contrast with approaches based on deterministic architectures that require knowledge of all addresses for any potential disturbing code to determine which cache contents may be reused across executions.

Acknowledgments

This work has been mainly supported by the PROARTIS FP7 European Project under grant agreement number 249100. It has also been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557 and the HiPEAC Network of Excellence. Leonidas Kosmidis is funded by the Spanish Ministry of Education under the FPU grant AP2010-4208. Eduardo Quiñones is partially funded by the Spanish Ministry of Science and Innovation under the Juan de la Cierva grant JCI2009-05455.

References

[1] P. Clarke, "Automotive chip content growing fast, says gartner," in *http://www.eetimes.com/electronics-news/4207377/Automotive-chip-content-growing-fast*, 2011.

[2] R. Charette, "This car runs on code," in *IEEE Spectrum online*, 2009.

[3] APEX Working Group, "Draft 3 of Supplement 1 to ARINC Specification 653: Avionics Application Software Standard Interface," 2003.

[4] E. Mezzetti and T. Vardanega, "On the industrial fitness of wcet analysis," *WCET Workshop*, 2011.

[5] R. Kirner and P. Puschner, "Obstacles in Worst-Case execution time analysis." *11th IEEE International Symposium on Object-oriented Real-time distributed Computing*, pp. 333–339, 2008.

[6] P. Puschner, R. Kirner, and R. Petit, "Towards composable timing for real-time software," in *Workshop on Software Technologies for Future Dependable Distributed Systems*, 2009.

[7] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzeti, E. Quinones, and F. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs," in *ECRTS*, 2012.

[8] F. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim, "PROARTIS: Probabilistically analysable real-time systems," To appear in ACM TECS, Tech. Rep. 7869 (http://hal.inria.fr/hal-00663329), 2012.

[9] L. Kosmidis, J. Abella, E. Quinones, and F. Cazorla, "A cache design for probabilistically analysable real-time systems," in *DATE*, 2013.

[10] AUTOSAR, "AUTomotive Open System ARchitecture," 2012, http://www.autosar.org.

[11] Aircraft And Sys Dev And Safety Assessment Committee, "Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment," *ARP4761*, 2001.

[12] S. Kotz and S. Nadarajah, *Extreme value distributions: theory and applications*. World Scientific, 2000.

[13] A. Baldovin, E. Mezzetti, and T. Vardanega, "A time-composable operating system," *WCET Workshop*, 2012.

[14] C. Watkins and R. Walter, "Transitioning from federated avionics architectures to Integrated Modular Avionics," 2007.

[15] F. J. Cazorla, P. M. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero, "Predictable performance in SMT processors: Synergy between the OS and SMTs," *IEEE Transaction on Computers*, vol. 55, no. 7, 2006.

[16] W. Feller, *An introduction to Probability Theory and Its Applications*, J. Willer and Sons, Eds., 1996.

[17] SoCLib, "SoCLib," 2012, http://www.soclib.fr/trac/dev.

[18] J. Wetzel, E. Silha, C. May, B. Frey, J. Furukawa, and G. Frazier, *PowerPC User Instruction Set Architecture*, IBM Corporation, 2005.

[19] Aeroflex Gaisler, *Quad Core LEON4 SPARC V8 Processor - Data Sheet and User's Manual*, 2011. [Online]. Available: http://microelectronics.esa.int/ngmp/LEON4-NGMP-DRAFT-1-6.pdf

[20] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," in *WCET Workshop*, 2010.

[21] ARINC, *Specification 651: Design Guide for Integrated Modular Avionics*, Aeronautical Radio, Inc, 1997.

[22] M. Paolieri, E. Quinones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware support for WCET analysis of hard real-time multicore systems," in *ISCA*, Austin, TX, USA, 2009.

[23] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero, *An Analyzable Memory Controller for Hard Real-Time CMPs.*, Embedded System Letters (ESL), 2009.

[24] B. Akesson, A. Hansson, and K. Goossens, "Composable resource sharing based on latency-rate servers," in *Proc. DSD*, Aug. 2009.

[25] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable SDRAM memory controller," in *CODES+ISSS*. USA: ACM, 2007.

[26] P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla, "On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 34:1–34:25, Jan. 2012.

[27] M. Fernández, R. Gioiosa, E. Quiñones, L. Fossati, M. Zulianello, and F. J. Cazorla, "Assessing the suitability of the ngmp multi-core processor in the space domain," in *ACM international conference on Embedded software (EMSOFT)*, 2012.

[28] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," *European Dependable Computing Conference*, vol. 0, pp. 132–143, 2012.