A First Step Toward Using Quantum Computing for Low-level WCETs Estimations

STEPHANE LOUISE, CEA, LIST

29

Low-Level analysis of Worst Case Execution Time (WCET) is an important field for real-time system validation. It stands between computer architecture and mathematics, as it relies strongly on variants of abstract interpretation. One of the features that causes the largest uncertainty regarding WCET evaluation for low-level analysis of sequential execution on a single processor is taking Cache Memory-related Delays (CMRD) and Cache-related Preemption Delays (CRPD) correctly into account. Research work from the 1990s provides a good basic framework for this problem as long as a task runs without preemption. But when preemption of tasks is allowed, although several formalisms exist, their predictive power is lower and the usual approach relies on analyses of NP-hard problems.

In this article, we want to show some potential advantages of using a formalism inspired by Quantum Computing (QC) to evaluate CMRDs with preemptions while avoiding the NP-hard problem underneath. The experimental results, with a classic (non-quantum) numerical approach, on a selection of Malardalen benchmark programs display very good accuracy, while the complexity of the evaluation is a low-order polynomial of the number of memory accesses. While it is not yet a fully parallel quantum algorithm, we provide a first roadmap on how to reach such an objective.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; *Real-time systems software*; Real-time schedulability;

Additional Key Words and Phrases: Worst-case execution, cache memories, real-time, cache related preemption time

ACM Reference format:

Stephane Louise. 2019. A First Step Toward Using Quantum Computing for Low-level WCETs Estimations. *ACM Trans. Archit. Code Optim.* 16, 3, Article 29 (July 2019), 22 pages.
<https://doi.org/10.1145/3335549>

1 INTRODUCTION

Validating a real-time system means to ensure all time constraints can be met. This involves a careful analysis of the system design, including its scheduling policy. But when it is required that the system always meet its time constraints, whatever the odds, one thing is important: to know how much processing time any given real-time task will take on the system in the worst case. The so-called Worst Case Execution Time (WCET) analysis is, therefore, an important step to validate a given real-time application on a given system. This is a complex field of research between processor architecture and mathematical models. It relies heavily on variants of abstract

New article, not an extension of a conference paper.

Authors' address: S. Louise, CEA-Saclay, Nano-Innov, PC172, Bat. 862. 91191 Gif-sur-Yvette, France; email: stephane.louise@cea.fr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2019/07-ART29

<https://doi.org/10.1145/3335549>

interpretation (Cousot and Cousot 1977) and can be divided into two subfields: High-Level WCET analysis is mostly focused on the analysis of the control and data flows and does not require a fine knowledge of the target processor architecture (knowing the ISA¹ and its semantic is sufficient); and Low-Level analysis, which would give how much time it would take to execute a given instruction in the specific context of the analyzed program, on the target processor.

As the design and the feature of modern processors make the execution time of single instructions heavily dependent on their execution context and history, models used for Low-Level analysis of WCETs also become more elaborate as time passes. However, WCETs are never absolute, because execution time is so context and architecture dependent that some simplifications are required. This leads to overestimations of WCETs, which can lead to over-engineered and more expensive systems.

One of the modern features of single core processors that holds most of the uncertainties at the low-level analysis is cache memories, even more so when preemptive scheduling are allowed. As a matter of fact, the execution time of a memory related instruction depends heavily—usually by one or two orders of magnitude—on the presence of the given data in the cache. In turn, this depends on the possible histories of execution of the analyzed task, but the analysis is jeopardized by the possibility of preemption. Current approaches require the WCET analysis tool to find the worst places in the analyzed task to preempt it (i.e., find the most disruptive preemption points). This problem is NP-hard, and usually engineers do not have an interest in these points of preemptions, but only on the associated WCET. While knowing these preemption points is sufficient to calculate the WCET, as the problem is NP-hard, it could be better to directly evaluate the WCET without having to find these points of preemption.

Our contribution in this article is to present a new approach to evaluate cache-related contributions to WCET for single core execution, for in-order processors, especially in the case of preemptions. It also features a new way to take into account indexed accesses. It is shown on LRU policy and validated on a subset of Mälardalen University WCET benchmark programs. Our approach is inspired by the linear operations of Quantum Mechanics and use the Markovian background of the method to obtain a good estimation in case of preemption for a negligible computing cost compared to the usual approach. While it is not a full parallel quantum algorithm yet, it is a first step in that direction and shows how such a method bypasses the NP-hard characteristic of the usual approaches to the problem.

Most of the discussion is focused on Data cache memory within the article, because it is the difficult case, compared to Instruction cache, and it is easier for the reader to focus on only one type of cache in the explanations. Nonetheless, all the exposed work is valid for Instruction caches, and Instruction caches are taken into account in our experiments. It can also be applied to several levels of memory cache hierarchy.

This article introduces the context and a selection of most important publications of the state-of-the-art in Section 2. Section 3 presents the mathematical base of our formalism, Section 4 discusses the computing cost of the approach, and Section 5 shows the results of experimental evaluation on several programs from the Mälardalen WCET benchmark suite. Finally, in Section 6, we provide a first limited quantum algorithm bases on our approach before concluding the article.

2 ON CACHE MEMORY AND WCETS

In this section, a quick introduction to the base of cache memories, a very simple overview of the principles of quantum computing and some key elements of the related work on WCET and CRPD are provided.

¹Instruction Set Architecture.

Fig. 1. Access to memory with a four-way associative 1KB cache memory with a line size of 32B. If the data or code corresponding to the accessed address is not in the cache, then the cache fetches it from main memory (including the whole line in which it will be copied) and replaces the least recently accessed of the four possible lines.

2.1 Cache Memories

Cache memories provide a means to mitigate the so-called memory wall: The speed of processors increases yearly about an order of magnitude faster than the speed of main memory. Hence, since the '80s, little amounts of high-speed memories have been added close to the processing parts, to provide a local copy of often or recently used values in memory and to provide a speedup on average access time to memory. If the content of the local memory is managed by the hardware, then this is a cache memory, by definition.

The allocation of memory copies in a cache is done in amounts called lines whose size s_l is usually a few dozens of bytes (e.g., 32B or 64B for L1 cache memories). The location of the copy is settled by several parameters including the cache memory associativity A , which determines the number of lines in which a particular copy of the memory can reside, its size, and its replacement policy. For an A -way associative cache with l lines of b bytes per line, then each set defines a $l \times b$ space of addresses. The cache has total number of lines $L = l \times A$ and a total size of $S = b \times L = b \times l \times A$. The replacement policy determines which previously used line of the cache must be replaced when a new snippet from the memory must be copied in the cache after a miss. An illustration of a usual replacement policy (the Least Recently Used—LRU—policy) is shown in Figures 1 and 2.

2.2 A Laymen Introduction to Quantum Computing

While the model shown in this article is not yet a full Quantum Algorithm (see Section 6 for an overview on how a real quantum algorithm would be achievable), it is still inspired by the type of computations done in quantum mechanics. Future research work will focus on transforming the formalism shown in the article to a full quantum algorithm that would also add the power of quantum parallelism. Nonetheless, to show how our model relates to quantum computing, we now introduce a few elementary concepts from quantum computing and see how to build a computing space and find a solution.

Computing Space. In quantum computing the most elementary computing space is composed by a set of two orthogonal vectors denoted $|0\rangle$ and $|1\rangle$ (Dirac notation) and a *qubit* (for Quantum Bit)

is a normalized linear combination of these two vectors. Therefore, while a regular bit can only take two values, 0 or 1, a *qubit* is a continuum of values in the *Bloch* sphere of radius 1 (coefficients can be complex numbers). Any computing space is a tensor product of several qubits in the general case.

For any part of a quantum algorithm, an operation can be made on one or several of the total number of qubits in the system. Basically, there are two types of quantum operations:

- reversible operations using unitary operators,
- irreversible operations called “*measurements*” or “*observations*,” which are projections onto Eigen-vectors of the observation operator.

In the latter case, the probability to observe a given state ($|0\rangle$ or $|1\rangle$ for a single qubit) is given by the module of the projection onto the observation space. The state afterward will be modified as an Eigen-vector of the operator. Therefore, in the general case, observation is not a linear operation, although in particular configurations or when the normalization factor is not used, it can be linear.

It is worth noting the dimension of the computing space in the case of several qubits is the tensor product of the spaces, and therefore it is exponentially larger than the Cartesian product we use in our current approach, but the Cartesian product can be seen as a subset of the tensor product.

For a complete introduction on quantum computing, refer to Nielsen and Chuang (2011).

Approach within the Work. The aim is not to provide a full quantum algorithm yet. Hence, for this first step, we will accept to lose some quantum information in exchange for a simpler model with only linear operations and a state with lower dimensions. This means the formalism within this article will use only linear operations on vectors (not unitary in the general case) and the Dirac notation for state vectors. A first limited quantum algorithm based on our approach is given in Section 6.

2.3 State of the Art and Related Work

The research problem is to find a way to take into account the effects of preemption in the calculation of WCET (or “*cache related preemption delays*”—CRPD). This is an issue for the low-level analysis of WCETs. Without preemptions, the problem is already evaluated with an acceptable accuracy by using set arithmetic applied to cache policies to determine two extreme possible contents of any cache: *must analysis* provides what is certain to be in the cache at any given point of execution, whatever the previous execution path, and *may analysis* provides the complement, i.e., all the content of what may be in the cache at the same point of execution.² The base principle is to provide an abstract operation on the sets of the cache for any access, and one other for path recombination (see Alt et al. (1996) and Ferdinand and Wilhelm (1999)). Over the years several related methods have been designed and improved upon one another. Huynh et al. (2011) offer a nice point of view of a more recent work, and we will use some of their examples to compare our approach to theirs. Once the possible states of the cache are statically evaluated for any possible path, a classical pipeline analysis utilizes the results to calculate the worst-case execution times. However, this approach does not work with preemption, because preemption can occur at any point in the execution, and the results on cache contents and timing analysis depend strongly on the exact point of preemption. The base problem on which relies the usual approaches is NP-hard: Finding the worst point of preemption in a given piece of code (Busquets-Mataix et al. 1996; Lee et al. 1998) is a problem equivalent to graph coloring.

²Equivalently, the set of what cannot be in the cache at this point can also be provided, i.e., the set of all accesses that are sure to trigger a cache miss.

The current state-of-the-art focuses on methods to evaluate in a quicker way (basic block level rather than instruction level) the worst possible preemption points (Negi et al. 2003; Ramaprasad and Mueller 2006). To comment on the method developed in Ramaprasad and Mueller (2006), the authors require establishing a string of all accesses in order, and link accesses done on the same cache set, then find the partition that cut the maximum number of lines, thus maximizing cache misses. Iterate on the obtained set for the second, third, and so on, worst preemption points. But even though, these methods mitigate the complexity of the problem, it is still NP-hard, and requires exact solutions (because the worst-case execution path may vary with the number of preemptions). Hence, for large enough problems, resolution becomes non feasible. More recent works like Altmeyer and Burguière (2011) try to figure a way to make the problem more accurate, nonetheless, all these approaches relies on a computation of useful cache blocks, which still remain difficult to figure out. By contrast, our approach does not rely on this concept and does not require knowing the future accesses to memory at any point in the execution. Some elements of their approaches on abstract interpretation could be used into our own approach, because it relies on the same ground of abstract interpretation. This was not yet fully analyzed or developed yet.

Other approaches try to mitigate the problem by reducing uncertainties on the content of the cache memory. A first approach is “*cache segmentation*”: by using task specific segments of cache, preemption delays are avoided (see, e.g., Vera et al. (2003)); hence single-task WCET static analysis is sufficient. But of course, reducing cache size by partitioning the cache has a significant impact on single task performance. A closely related method is to use a mode available with some processors called “*cache locking*.” When this mode is activated, lines present in the cache cannot be flushed anymore, and no new line will be fetched into the cache either. Schematically speaking, this mode transforms the cache in a software transparent scratch-pad (see Puaut and Pais (2007)), and (after the initial prefetch phase) renders memory access time deterministic. This has also a significant impact on tasks performance, as in the case of cache partitions (even more so because the cache address content is fixed once for all, with few possibilities of dynamic adaptation).

For a different approach, an initiative held initially by Cazorla et al. (2012), about probabilistic approaches to WCET, started as the European working group *Proarstist* after the publication of a foundation paper at ECRTS 2009 (Quinones et al. 2009). This has led to several recent publications (Davis et al. 2013; Kosmidis et al. 2013; Maxim et al. 2012). Nonetheless, all these approaches are based on long-tailed distribution theories and require statistical analysis of timing measurements, whereas ours is a mathematical construction that is deterministic and does not require measurements.

Our approach is developed on the base of an abstract space with a quantum computing analogy so that to avoid the underlying NP-hard problem: Determining the most disruptive points of preemption (the NP-hard problem) is a sufficient condition to solve the problem, but it is not necessary. In this work, the algorithm evaluates a WCET without knowing the worst-case preemption points of the program under evaluation.

3 A NEW FORMALISM

Our formalism has its bases on a good choice of the modeling space. Contrary to the other approaches where the state of the hardware component is what is being traced, in this approach, we keep trace of every element that can interact with the hardware component: The principle is to update their states regarding their interaction with this hardware component. Hence, the considered space is (sometimes much) larger than the set of what can interact at a given time with the hardware component, but the operations are much simpler (most of the time linear) and are only applied to a subset of the considered space.

Because our formalism is not restricted to cache-related Worst Case delays, in this section, we will propose a formal approach, which is then applied to the particular case of cache memories. For a concrete illustration, simple illustrative examples follow each theoretical explanation in the following.

It is worth noting that we will use Abstract Interpretation as in the general approach defined by Cousot and Cousot (1977), i.e., as operations on an abstract state to model some properties of a concrete system, in a simplified way, not in the sense that became prevalent in the WCET community. What we use here is a pseudo-execution of binary code in an abstract state. It is also worth noting that the approach developed in this article is not yet meant to be applied to safety-critical real-time systems; although we are confident it works reliably when correctly used, a formal demonstration is not provided here. Still, the formalism can be inspirational to other researchers, and we plan to improve on it.

3.1 Model for Memory Accesses

Concretely, the considered space-state for a data cache with a line size of l , a total number of lines of N , and an associativity A is defined by considering the total memory space addressed by the piece of code under analysis: For any topologically compact set of addresses $[a_{start}, a_{end}]$, let us consider the mapping in lines in the cache, in the interval $[\lfloor \frac{a_{start}}{l} \rfloor, \lfloor \frac{(a_{end}+l-1)}{l} \rfloor]$ in $\mathbb{N}$, and for each of these lines, a vector is defined.

Example. For the case represented in Figure 1, the interval of addresses from 0xFAA to 0xFC2 would be translated as a set of two state-vectors (one for the line 0xFA0 to 0xFBF and one for the line 0xFC0 to 0xFDF).

It can be repeated to all compact subsets of the address-set potentially accessed by the considered piece of code, and the final vector-set is the union of all these: They represent the set of lines in memory that can be mapped on the cache when the piece of code is running.

The goal to achieve with this space is that any interaction of the space with the hardware component can be modeled as a Finite State Machine (FSM, or an automaton). Once this goal is achieved, the result is quite simple: We can associate a component of the vector state with each state of the automaton (the order of the vector is irrelevant as long as it is coherent for all spaces), and then operations for going from one state to another are simple transition matrices. The operation is a matrix product with a state-vector, which is linear. As updates of states are made with a simple matrix product, the formalism can be considered as a Markov chain³ (Basharin et al. 2004; Markov 1906). For deterministic automata/FSMs, Markov chains are also deterministic.

Optional elements of the state space can keep track of the relevant parts of the state history (e.g., the number of hits and misses in the cache). It is updated by the state-vector before state update. This is also a linear operation. As the usual notation for state-vectors is the Dirac notation from quantum mechanics as seen in Section 2.2, where a state-vector associated with state s is noted $|s\rangle$, if the history state-vector is noted $|h\rangle$, then an update of state-vectors can be written as the follows:

$$\begin{cases} |h'\rangle = |h\rangle + P |s\rangle \\ |s'\rangle = O |s\rangle \end{cases}, \quad (1)$$

where primed state-vectors are state-vectors after update, P is usually a projector for the interesting parts of the state for the history state-vector, and O is the state-transition matrix.

³This means that this formalism possesses all the properties of Markov chains, including a stationary state.

Fig. 2. State machine for LRU replacement policy: H_0 is the youngest access in history, H_1 the second youngest access, and so on. M is the miss state, which means that the considered line in memory is not in the cache. The Least Recently Used (LRU) policy means that when an access occurs to a given line in memory, it is copied in the cache and its history level is set to 0. When another line that has the same set of address modulo the size of cache divided by its associativity is accessed, a concurrency operation occurs: the considered line history counter is increased by one until it reaches $A - 1$. If another concurrency occurs, then the line is expelled from the cache, and a later access would be a miss.

3.1.1 Application to Cache with LRU Policy. The Least Recently Used policy (LRU)) is the paragon of deterministic policies. The principles of selection for the set chosen to override a previously stored line of cache are based on how recent the last access to a data in a line was. The state machine for any line in memory that can be present in the cache looks as shown in Figure 2.

The first step is to model the LRU replacement policy without preemption and then use the properties of the Markov chain to create an attenuation-driven model of preemption on top of the single process model.

According to the automaton presented in Figure 2, we can define two types of state-transition operations: One for access in memory, called $a_{LRU}^\dagger$, and one for concurrency in the cache, called β_{LRU} . By definition, the transition matrices are

$$a_{LRU}^\dagger = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \end{pmatrix}, \quad \beta_{LRU} = \begin{pmatrix} 0 & 0 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & 0 & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 1 \end{pmatrix}.$$

Simply speaking, $a^\dagger$ is the projector on state H_0 (i.e., next access would be “hit” as first in LRU history) and β is the second diagonal plus a self-coupling of the M (Miss) state.

To model a load or store operation at a given address in memory, the simplest way to do so is to use an $a^\dagger$ operation on the state-vector corresponding to the memory line where this address appears, and perform a β operation to any memory line that can share the same set in the cache. Using the operators this way means that the operations have little dependence on execution history of the considered piece of code. Nonetheless, this is overly pessimistic in the general case (although the impact on the number of misses and the execution time is low, usually less than 2%). But, as this is a lot of potential operations (even if most of them are quite trivial), we want to restrain the set of operations to consider for β by using *may analysis* (Ferdinand and Wilhelm 1999) (i.e., only considering the set of memory lines that may be in the cache at the considered time).

The global operator can be chosen to be

$$\mathbb{A}_i = a_i^\dagger \otimes_{j \in K} \beta_j,$$

with K being the set of lines such that *may analysis* find out they are in the cache and such that $j \equiv i \pmod N$ and $j \neq i$ (i.e., in the set of lines that are concurred by access i). What can be

This means we lose information in this process. This would not happen in a fully developed quantum algorithm, and future work will provide an improvement through the use of a quantum operator in this case. Such a development would achieve a goal of actual quantum-parallelism.

Although the *join* operation loses information, it is a simple operation, and we can find a linear operator that simulate the behavior of the loop on the cache as there is always a fixed point in all Markovian models.

Illustration on a Simple Example. Let us consider a fully associative cache with only two lines and a simple program with a loop and a sequence of accesses represented in the Control Flow Graph (CFG) of Figure 3.

The first path of the loop (blocks B1, B3) conveys two accesses a and b . The second path in the loop (blocks B2 and B4) conveys two other accesses c and a . The first execution of the loop can choose any of the two paths. The first access for path (B1, B3) is a and the operator is $\mathbb{A}_a = a_a^\dagger$, since the *may set* is $\{a\}$. The initial vectors are $t(a, b, c) = ((0 \ 0 \ 1), (0 \ 0 \ 1), (0 \ 0 \ 1))$, so it is a first miss for a . The same for second access on b (block B3). This is a miss for b and the operator is $\mathbb{A}_b = a_b^\dagger \otimes \beta_a$. After that the state space is $((0 \ 1 \ 0), (1 \ 0 \ 0), (0 \ 0 \ 1))$.

The same can be done on the other path (B2, B4) in the loop. This path conveys the accesses c followed by a . The same way as previously, the access operators will be $\mathbb{A}_c = a_c^\dagger$ for the first access, and $\mathbb{A}_a = a_a^\dagger \otimes \beta_c$ for the second, respectively, and the final state will be $(a, b, c) = ((1 \ 0 \ 0), (0 \ 0 \ 1), (0 \ 1 \ 0))$.

Applying the fusion operation before the block B5 gives $(a, b, c) = ((0 \ 1 \ 0), (0 \ 0 \ 1), (0 \ 0 \ 1))$ and the set of concurrence is fully populated with the 3 accesses a , b and c .

For the subsequent loop iterations, the access operators for the path (B1, B3) will be $\mathbb{A}_a = a_a^\dagger \otimes \beta_b \otimes \beta_c$ and $\mathbb{A}_b = a_b^\dagger \otimes \beta_a \otimes \beta_c$ with an unchanged final state $(a, b, c) = ((0 \ 1 \ 0), (0 \ 0 \ 1), (0 \ 0 \ 1))$. The same way for the second possible path (B2, B4) with accesses c and operator $\mathbb{A}_c = a_c^\dagger \otimes \beta_a \otimes \beta_b$ and access a with operator $\mathbb{A}_a = a_a^\dagger \otimes \beta_b \otimes \beta_c$. The final state will still be $(a, b, c) = ((1 \ 0 \ 0), (0 \ 0 \ 1), (0 \ 1 \ 0))$. As both states are unchanged, so is the state after the join operation, which is $(a, b, c) = ((0 \ 1 \ 0), (0 \ 0 \ 1), (0 \ 0 \ 1))$. And so on for any number of loop increments after the first.

The number of misses will correctly be computed as $1 + n$ in the worst case, with n being the number of iterations of the loop. Access a is always *hit* except the first time, and there can be either a b or a c miss in the worst case of execution for each loop iteration.

It is worth noting that our model already encompasses the notion of *Younger Set* from Huynh et al. (2011) and that the use of *May analysis* is a way to optimize the operators, but it is not mandatory as it is possible to use the total set of accesses as references.

3.3 Indexed Accesses

The case of indexed accesses where static analysis cannot determine exactly where in memory a given access is done (often only a range of addresses is known, e.g., access in an array whose index is data dependent), can be treated for a pessimistic analysis as only concurrent accesses (β operators) on all the possible sets that can be in concurrence. If the data set⁵ is larger than $(A - 2)l$ in some cases or always if it is larger than $(A - 1)l$, then the whole set of considered memory is impacted. We can do better. For a less pessimistic operator, we use partial concurrence operations and partial access operations so that the number of accesses is one, and the number

⁵If all that we are discussing here can be applied indifferently to data or instruction caches, then the case of unknown access in a given range only applies to data accesses, because by construction of low-level WCET analysis, the address of the current instruction is always known.

of concurrences is also one. We distribute the accesses in priority on the least favorable accesses and the concurrences on the best possible ones. These approaches are consistent with the usual *may/must* analysis.

With this modeling of the set of vector states for the whole considered memory, it is easy to feed a pipeline analysis of a low-level WCET analyzer to deduce possible execution times of basic blocks.

In the following, we introduce a new solution that is pseudo-probabilistic, by distributing the access over all the set. The theoretical justification of how it can work is the same way as the preemption case as shown in Section 3.4.

Illustration on a Simple Application Case. Let us consider the case of a direct mapped cache memory with two lines. An indexed access is considered over two lines $\{E, F\}$ mapped on the two different lines of the cache memory. Let us call E^* the indexed access (so either over E or F), and the following sequence of accesses:

$ABABE^*A$

Then, at the first access, the *may sets* are $\{\{A\}, \{\}\}$, the operator is $a_A^\dagger$, which conveys a miss and the state-vector after that is $\sigma_A = (1 \ 0)$. For the second access (B), the *may sets* become $\{\{A\}, \{B\}\}$, the operator for the access is $\mathbb{I}_A \otimes a_B^\dagger$, which conveys also a miss for B , and the states vectors after that are $\sigma_A = (1 \ 0)$ and $\sigma_B = (1 \ 0)$.

The two following accesses of A and B are processed the same way, and convey cache hits for both. The state-vectors and the *may sets* are unchanged.

The interesting part is the indexed access E^* , and we illustrate it on the case of the pseudo-probabilistic range operator. Then, the operator we apply with the indexed access is

$$\frac{1}{2}\beta_A \otimes \frac{1}{2}\beta_B \otimes \frac{1}{2}a_E^\dagger \otimes \frac{1}{2}a_F^\dagger.$$

Because both accesses are distributed over E and F and concurrences are distributed over A and B (because of the *may sets*). The *may sets* are now $\{\{A, E\}, \{B, F\}\}$. The vector states after the operator is then: $\sigma_A = \sigma_B = \sigma_E = \sigma_F = (\frac{1}{2} \ \frac{1}{2})$. The associated number of misses of access E^* is $\frac{1}{2} + \frac{1}{2} = 1$.

For the last access, things are also interesting. The *may sets* are unchanged, and the operator is $a_A^\dagger \otimes b_E \otimes \mathbb{I}$. The result is an unusual “half-miss” and the state-vectors after the access are $\sigma_A = (1 \ 0)$, $\sigma_B = \sigma_F = (\frac{1}{2} \ \frac{1}{2})$ and $\sigma_E = (0 \ 1)$.

As the sample program ends here, we must take care to normalize the pending misses. Looking at the state-vectors after the program, we still have $\frac{1}{2} + \frac{1}{2} = 1$ pending miss (respectively, for B and F), so we can estimate at most 4.5 misses for the whole program (as always, we seek an overestimation of the value if the exact value is difficult to obtain).

It is worth noting that if the program had one B access after that, then we would have found the correct number of misses, i.e., 4. Therefore, depending on the access pattern, the result will be exact or an overestimation.

3.4 Preemptive Multitasking

Preemption means that the considered piece of code can be interrupted at any point in its execution (after the execution of any instruction, since instruction executions are always atomic) and this can have a catastrophic impact on the content of caches as cached data or code lines are likely to be flushed out of the cache. Sometimes it has little impact on execution time, e.g., a point where the content of the cache is not really relevant for the remaining part of code execution. But at other points, it can produce a very large overhead, when lots of data or code elements that were planned

Fig. 4. State machine for pseudo-random preemption effect on cache content. The *Hit* state maps all the possible hit states $(H_0, H_1, \dots, H_i, \dots, H_{A-1})$.

to be used by the remaining part of the program are flushed out of the cache, and a considerable amount of time is necessary to fetch them again in the cache.

For this problem, we considered a pseudo-probabilistic approach also inspired by quantum computing in which the Markov formalism fits well, as shown in Figure 4. The simple idea is to find an operator that can model that any element in memory could be removed from the cache at any time. It is easy to write it as a linear combination of the identity and of the B operator, i.e., the projector on the miss state⁶:

$$P_r = (1 - p)\mathbb{I} + pB, \quad (2)$$

where p is the attenuation factor, analogous to the mass of a pseudo-particle. It can be evaluated from parameters that must be known beforehand in a safety critical real-time system:

- n_p is the maximum number of preemptions that the piece of code may undergo. This number is known for safety critical systems because of the programming constraints, which forbid, e.g., dynamic creations of tasks or uncontrolled rates of interruptions (all task awakening have a maximum period).
- N_i the lowest number of instructions run by the processor when the analyzed piece of code is executed.

Then, for LRU policy, it is easy to find that $p_0 = \frac{n_p}{N_i}$ is an overestimation of the attenuation factor of Equation (2).

It is worth noting that the state-vector associated with a memory line *after* an access to this line is a perfect *memory-cache hit* state, as if it were the result of an observation in the sense of quantum mechanics, and as it was the case without preemptions.

PROOF. Let us suppose the worst case (the shortest path) program is N_i sequentialized instructions and all of them access to a given location in memory (mapped to a cache line). Since any instruction is executed as if it was atomic for any kind of single-core processor, then, the equivalent operator for the whole program is $a^\dagger \times P_r$ at the power of N_i , since there are N_i accesses in this worst case program. The number of instruction is what matters, because it is the number of possible preemption points (all instructions are atomic). All matrices $\mathbb{I}$, $a^\dagger$, and B are idempotent operators, and $\mathbb{I}$ is the neutral element of the matrix product. Therefore,

$$(a^\dagger \times P_r)^{N_i} = (1 - p)a^\dagger + pB.$$

This operator is also idempotent. Hence, for the number of misses, we have $p \cdot N_i$ at the end of the program. Since we want that to be n_p , then $p \leq \frac{n_p}{N_i}$ (QED). Using N_i ensures the attenuation factor calculated this way is an overestimation. $\square$

⁶That is, $B = \begin{pmatrix} 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \\ 1 & 1 & \cdots & 1 \end{pmatrix}$.

In this model, the behavior is deterministic on the results and can be seen as an analogy to quantum mechanics. It is constructed to generate an evanescent model of cache memory (data in the cache disappear according to an exponential rule), with a rate that match the overall actual number of observed misses in the worst case. It has no sense if it is not applied and interpreted outside of whole accessed dataset.

By applying the preemption operator before each instruction, we can find the value of the state-vector regarding the presence of any memory line in the cache. This is a lot of operations (even if they are simple). For load/store operations and data-cache memory, for pipeline timing regarding memory state in the cache, and for any memory address, we can wait to apply an $a^\dagger$ or a β operation. Let us suppose we have a counter of the number of instructions executed since the last update of the state l_{addr} , and then updates of the state-vector for a line at address $addr$ are given by the new operators:

$$a^{*\dagger} = P_r^{l_{addr}} a^\dagger \quad \text{and} \quad \beta^* = P_r^{l_{addr}} \beta$$

This operation must be associated with a reset of counter l_{addr} . Moreover, as B is idempotent (any non-trivial power of B is B), then

$$P_r^{l_{addr}} = (1 - p)^{l_{addr}} + [1 - (1 - p)^{l_{addr}}]B.$$

This gives a pessimistic value of the hit states. For instruction cache, we use the same principle.

3.4.1 Standard Deviation. While it is possible in this framework to be conservative and find an overestimation of the number of misses by using a pessimistic value of the error, we want to show here the case of a less conservative approach in this article. For that, we need an evaluation of the standard deviation.

This model for any given memory access provides a pseudo-probability of miss in the interval $[0, 1]$; therefore, it lays the base of a Benouilli distribution, whose variance is $V = p(1 - p)$. We can sum up the variance for all the memory accesses to provide a reasonably accurate estimation of the total variance, because the modulo operation on the addresses that gives the localization of the memory copy can be seen as a randomization of the interferences; therefore, we can make the assumption that in most of the case the covariance will be negligible.

The global result can be seen as a random walk (binomial distribution) between a case without preemption and a case of maximum interference of the preemption. The mean value is given by the attenuation factor and the variance V will give the confidence interval for this random walk, within the mathematical model.

As a standard procedure, to enhance the confidence level on physical results, it is possible to use intervals of several standard deviations. On Gaussian distributions (which is an over-approximation of the binomial distribution), the 1σ criterion (i.e., $m + 1\sigma$) would give a correct result⁷ with a confidence level of 84%. On our model, this criterion is over-evaluated as we have an upper-bound for the pseudo-probability of preemption. It means that the 6σ criterion can be though to never fail, and much less a 10σ criterion⁸ (i.e., $m + 10\sigma$). Therefore, for a first evaluation, the single σ criterion should suffice; for excellent confidence, we may want to use a 3σ criterion; and for “never fails” cases, we may want to use a 10σ criterion.

3.4.2 Illustration on a Simple Case. Let us use our previous simple example of Section 3.1.2, and introduce a possible preemption. There are 17 access instructions (and no other instruction for this simple example), so our attenuation factor is $p_0 = \frac{1}{17}$. The two first accesses (AB) are exactly the

⁷That is, pessimistic.

⁸For a Normal distribution, the confidence should be $1 - 7.6 \times 10^{-24}$, therefore could never be measured to fail as it is a very low probability, and our model already reduce the risk, because it gives a pessimistic evaluation.

same as the case without preemption. Things differ only at the second A access as the P_r operator must be applied. The current instruction is two instructions after the previous A access, hence $l_{addr} = 2$. So $P_r = (\frac{16}{17})^2 + (1 - (\frac{16}{17})^2)B = \frac{256}{289} + \frac{33}{289}B$. The state-vector for A after the preemption operator is $(0 \ \frac{256}{289} \ \frac{33}{289})$, but it *must not* be interpreted as a $\frac{33}{289}$ “chance” of having a miss. In fact, the only point at which you can find a correct interpretation of the model is at the end of the program (because this is the way the model is constructed). Moreover, after the access itself, the state-vector is $(1 \ 0 \ 0)$, as in the non-preemptive case.

For the following B access, we can say exactly the same as for the A access, and for all the following A and B accesses. As a conclusion, the rough evaluation of the number of misses triggered by preemption alone (and that adds to the misses already taken into account) is $16 \times \frac{33}{289} \simeq 1.83$.

Again, this number makes no sense without a confidence interval, which is given by the Bernoulli distribution: Each time an access is made, we have an associated variance $v = p(1 - p) = \frac{33}{289} \times \frac{256}{289}$. This leads to $\sigma = \sqrt{16 \times \frac{33 \times 256}{289^2}} \simeq 1.27$. The actual number of supplementary misses is 2, and it is below the 1σ criterion, since

$$1.83 + 1.27 = 3.10 > 2.$$

For displaying the results, we will note the evaluation of the 1σ confidence interval separated from the base evaluation. So, we will denote the result for our simple example as $4.83 + 1.27$ (as there are three misses as a base, without preemptions) for the base and the estimation of the 1σ confidence interval.

As can be seen, in this case the 1σ criterion is sufficient in this case, but a 3σ criterion would only add three extra misses compared to the real case. This estimation is large because the number of accesses are also large, compared to the number of instructions. For a program with only four accesses to A and B, the same evaluation would yield $\sigma = 0.64$, which would also give a correct answer but a less pessimistic one.

4 COMPUTATIONAL COST

The computational cost of this method is a low-order polynomial expression. We still have to provide *may/must analysis*, but for the remaining parts, ordinary accesses are between $O(A^2 \times N)$ and $O(N^2)$, where A is the associativity and N the number of memory accesses.

PROOF. For the *may analysis* K set, we have usually the order of A lines in the cache and can be up to $\frac{N}{A}$ in the worst case (i.e., all the accessed lines are potentially in the cache and can be concurred by the accessed line). Then, each operation ($a^\dagger$, b or $a^{*\dagger}$ and b^*) on a state-vector is $O(A)$, because the dimension of the vector is A . Finally, there are one operation per access, so N of them. $\square$

For path-merging operations, each operation is $O(A)$, because it is only comparing the values of the coordinates of two state-vectors. Then, the range operator is the most complex one, because it operates on a whole range of memory, so must be applied to a large subset of state-vectors. Still, the complexity is bounded in the worst case by N , the number of accessed lines in the analyzed program. Therefore, the worst case is in the order of $O(N^2 \times A)$ for each indexed access (but usually much less).

As a conclusion, by changing the abstract interpretation model by a pseudo-probabilistic one, we change the computational complexity of finding WCETs for programs with preemption from a NP-hard problem to a simple polynomial one. The drawback is that we have confidence intervals but not an absolute confidence.

Table 1. Programs Selected from the Mälardalen WCET Benchmarks

Bench	Program	Control flow	Data access	Main issues
bssort	Bubble sort 100 int	Complex with nested double loop	Mostly predictable	Regular nested loop with possible early exits
crc	CRC calculation	Cascade of conditions leading to a single loop	Complex with indexed accesses	Indexed accesses are hard to predict
dct	Inverse Cosine Transform	Simple	Regular with large amount of local variables	local variables on stack. Preemption cost dominated by instruction cache misses.
matmult	Matrix multiply 20×20	3 levels of nested loops	Simple but with large reuse of data	Large amount of data reuse challenge the prediction of cache contents when data-cache are too small
ns	Nested search	Deep nested loops with early exits	Regular with large amount of data reuse	Nested loops, data reuse and unrolled deepest loop with lots of exit points

5 EVALUATION

We use a subset of the famous Mälardalen WCET benchmark suite (Gustafsson et al. 2010; research group 2006) to evaluate the accuracy of our model. The selected programs and their characteristics are shown in Table 1.

5.1 Selected Benchmark Programs

The selected set of programs explores a reasonable amount of problems that can be found in usual real-time applications. It is worth noting that since this is a general benchmark suite that aims at evaluating all sorts of issues in programs, some programs are most fitted to test a given aspect of the WCET evaluation, and is not always fitted to test another specific aspect. The important point is that we have a sufficient selection of programs to test several aspects of cache-related WCET delays evaluation problems.

All programs were compiled with GCC-4.6 with -O2 level of optimization. Our main test target was ARM-v7 ISA without floating-point instructions, but as we concentrate on cache related low-level WCET analysis, this is not a problem *per se*, as long as the selection of programs accounts for all or at least most of the possible behaviors.

It is worth noting that we want to evaluate the results of the presented model. As it corresponds to low-level static analysis, we made the assumption that the high-level analysis was as perfect as we could hope so, which means we provided the high-level analyzer with accurate data about the program behaviors if it could not evaluate them without hints. This approach allows for a fair evaluation of the low-level analysis provided by the model, without interferences from the high-level analysis.

5.2 Evaluating the Model on Non-preemptive Behaviors

As we already noticed, the low-level analysis provided by the model without preemption is deterministic (except the case of indexed accesses to an unspecified subset of memory, which use overestimation of a kind of probability). Therefore, except for the case of crc (the sole program in

Table 2. Results for Data-cache Size of 1KB, A=4

Benchmark	Number of memory accesses	Exact worst-case number of misses	Number of misses in the model		Exact WCET (cycles)	WCET estimated in the model	
			pseudo-probabilistic range operator	Concurrence only		pseudo-probabilistic range operator	Concurrence only
bsort	20,204	15	-	15	87,245	-	87,245
crc	1,128	165	165	171	18,613	18721	18,879
dct	1,690	14	-	14	5,556	-	5,556
matmult	24,416	1,364	-	1,364	122,300	-	122,300
ns	638	19	-	19	4,698	-	4,698

N.B.: As range operator is only used in *crc*, the different implementations are only shown for this case.

Table 3. Results for Data-cache Size of 256B, A=2

Benchmark	Number of memory accesses	Exact worst-case number of misses	Number of misses in the model		Exact WCET (cycles)	WCET estimated in the model	
			pseudo-probabilistic range operator	Concurrence only		pseudo-probabilistic range operator	Concurrence only
bsort	20,204	330	-	330	92,148	-	92,148
crc	1,128	293	293	332	19,129	19,395	19,627
dct	1,690	100	-	100	6,305	-	6,305
matmult	24,416	10,595	-	10,595	236,734	-	236,734
ns	638	22	-	22	4,716	-	4,716

our set that uses indexed accesses), we expect close to exact results with our model and also close to the state-of-the-art results in WCET evaluations.

For our evaluation, we used a base of ARM-v7 superscalar processor, with two-instruction fetch and two-instruction termination units. We have one unit for each following operand: Integer unit, Load/Store unit, Branch unit. The length of the pipeline is short (five stages: two for fetch/decode, up to two for execute, one for termination). Data-cache latency is considered 2-cycles and memory latency, 20-cycles. All the characteristics can be regarded as fairly standard for nowadays low-power embedded processors (e.g., Cortex-M family of ARM processors).

As we wanted to test with accuracy the case when cache memory is very constrained (because when it is not the case, e.g., for cache of 8KB or more, most or all the program data fits in the cache and results are much accurate and less interesting), we used a configuration of our test where data cache size are very small: 1KB 4-set associative LRU for the first case, and 256B 2-set associative LRU for the second case. For the same reason, line size is kept low at 32B. Results are given in Tables 2 and 3.

As expected, we can see that for deterministic sets of accesses, the results are indistinguishable from optimal with our formalism (N.B. except the *crc* benchmark, the others does not use data dependent range-accesses so no use of range operator is done in this case, therefore, we only used accesses at known addresses and using *may analysis* to infer the concurrency set of memory lines). The difficult case is when there is a non-determined range of accesses, as in the case of the *crc* benchmark. In this case, there is exactly 41 undetermined accesses, and with the operator and *may analysis* method combined, we obtain a good accuracy (less than 0.6% overestimation both

Table 4. Results for Data-cache Size of 1KB, with Variable Number of Preemptions, for Data Cache Misses

Bench	$n_p = 1$		$n_p = 2$		$n_p = 4$		$n_p = 10$	
	exact wc	model	exact wc	model	exact wc	model	exact wc	model
bsort	25	30 + 5	32	43 + 10	47	69 + 17	89	138 + 37
crc	167	169 + 2	169	174 + 3	173	183 + 6	177	214 + 18
dct	26	30 + 5	38	43 + 9	62	66 + 15	120	118 + 26
matmult	1,387	1396 + 11	1,410	1,427 + 22	1,456	1,490 + 42	1,594	1,673 + 105
ns	25	26 + 2	28	32 + 4	35	40 + 6	55	63 + 12

Table 5. Results for Data-cache Size of 1KB, with Variable Number of Preemptions, for WCETs

Bench	$n_p = 1$		$n_p = 2$		$n_p = 4$		$n_p = 10$	
	exact wc	model	exact wc	model	exact wc	model	exact wc	model
bsort	87,397	87,500 + 85	87,512	87,721 + 110	87,780	88,163 + 150	88,471	89,336 + 550
crc	18,750	19,610 + 30	18,792	19,670 + 50	18,868	19,780 + 80	18,944	20,050 + 180
dct	6,086	6,254 + 180	6,596	6,860 + 340	7,616	7,935 + 580	9,746	10,227 + 936
matmult	122,684	122,844 + 126	123,068	123,371 + 188	123,825	124,442 + 230	126,126	127,553 + 550
ns	4,799	4,817 + 25	4,872	4,919 + 65	4,994	5,055 + 120	5,387	5,446 + 204

on number of misses and on execution time for a 1KB cache, and around 3% for number of misses and execution time for 256B cache).

Using the pseudo-probabilistic range operator, for both sizes of cache, we obtained the exact results in number of misses. Regarding execution time, we are a little bit pessimistic due to the rough estimation of the memory access time. Nonetheless, the results are still good, with an error in execution time of 1.4% for each case. We can expect to have results very close to the theoretical case by using better estimations of the cache hierarchy access times.

5.3 With Preemption

The experiments were done on the same benchmark programs as in Section 5.2. We used a fixed L1 data-cache size of 1KB, as in Table 2, so we can compare with the case without preemption. The model was evaluated for a number of preemptions that sample a significant range from 1 to 10, so that the results can be evaluated on a representative range of number of preemptions. The experimental values are given in Tables 4 and 5.

As seen in Tables, the number of evaluated cache misses is always very close and have a good accuracy regarding the number of memory accesses (see Table 2), usually below 1%. The results are generally speaking very good for the number of misses (i.e., as expected nearly always overestimated, and within one σ of the actual worst case). For WCETs, we have a good accuracy, usually below 5% and really good for low numbers of preemption (1 or 2) as errors grows with n_p . It is worth noting that a large part of the errors in WCET evaluations comes from the simple model we used for modeling the timing of cache accesses (i.e., linear). With a more accurate model of cache access time with preemption, we can imagine having the same kind of very good results as for the number of misses.

Regarding the confidence interval, we can see that the overestimation of the attenuation factor p_0 of our model means that most of the time (here in all cases except for *dct* with $n_p = 10$), we already overestimate the number of misses without a need to use the σ evaluation. As seen in Figure 5, all the results comply with the 1σ criterion. In our experience with several dozens of

Fig. 5. Evolution of the estimate on several Malardalen programs, as the number of preemptions varies. The estimation includes the 1σ confidence interval (therefore, the safer value connects the upper points of the confidence interval). The comparison is done with regard to the case without preemption.

cache configurations and programs, we found a counter example only once, with a 2σ difference required to obtain an actual overestimation of the WCET.

We can conclude the results of the method are good and really worthy of further investigations and improvements. The results are interesting, because they provide a good estimation of what the WCET will be with preemption without the need of solving difficult Operation Research (OR) problems. The results on WCETs are not yet on par with the obtained results for the number of misses, but we do not see any hard point toward achieving at least the same accuracy for both hit/miss accounting and WCETs.

5.4 Comparison with State-of-the-Art Computation of CRPD

As seen in Section 2.3, while the case for an accurate computation of Cache-Related Preemption Delays (CRPD) is thought to be an important goal to achieve in the real-time community, the difficulties and issues with regard to obtaining useful and reliable CRPDs and WCETs in the case of preemptions is such that an important part of the community would rather avoid preemptions or limit their impact altogether by using techniques like, e.g., cache-locking, which allows for a much lower variability (if at all) of the WCETs with preemptions. Therefore, to our knowledge, there is little data available publicly to measure the accuracy of WCET computing tools and models especially in the case where preemptions are allowed.

The most recent publication we are aware of is by Shah et al. (2018), who try and re-create the results found in Altmeyer and Burguière (2011), but the authors do not compare their results with actual executions. In Altmeyer and Burguière (2011), the authors evaluate the CRPDs on an ARM7 architecture and the improvements of the model of aiT⁹ with a new notion of *useful cache blocks*. They succeeded in improving the previous evaluation of aiT by around 20% on average

⁹aiT is a commercial tool for WCETs safe evaluation: <http://www.absint.com>.

on Malardalen WCET benchmark programs. This result suggests that the usual evaluations are pessimistic with a margin of several dozens of percents.

This is also consistent with what the researchers' community can hear from engineers working on mission-critical software in, e.g., avionics and aerospace, about the comparison between WCET evaluations done by available tools and measured real-world performance. Therefore, even a 20% pessimistic evaluation is usually considered an acceptable or even good result, and that is why we believe our results as shown in this article can be considered noteworthy.

6 FROM LINEAR ALGEBRA TO A REAL QUANTUM ALGORITHM

The real power of quantum computing, and its implicit parallelism, is only unleashed in a multiple CFG exploration and by using superpositions of states while preserving the quantum information until the last observation. Not losing the quantum information along the way can only be achieved by a generalized utilization of unitary operators. We are not yet there, but our research work is still a first significant step toward that goal.

For simple CFG, with a single history of execution, or for short *single-path execution*, or programs that can be reduced to that, superposition of states is not a requirement, so a quantum algorithm is quite straightforward to implement from our approach.

6.1 Single-path Execution

Operators $a^\dagger$ and β are not unitary. This means they destroy the quantum coherence information that would be available otherwise. This is not necessarily an issue for single-path execution as there is no obvious useful superposition of states: There is only a single history of execution, and the results are deterministic, as seen in Section 3.1.

To obtain a derivable base for a quantum algorithm, we must show how our formalism can be decomposed in unitary operations and observations (or measurements). This is not difficult:

- We can define the permutation matrix that fits the β operator by choosing

$$b_{LRU} = \begin{pmatrix} 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & 0 & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}, \quad (3)$$

which is a unitary operator and, therefore, can be programmed in a quantum computer.

- Then we can operate an observation (measure) on a part of the resulting vector-space by considering if it is a hit, first-in-history, or a miss. The resulting value is preserved for all the value except the two misses. In the latter case, the state after observation is a *miss*.
- The resulting combined operation is β_{LRU} .
- For the $a^\dagger$ operation, this is a global observation of the state-vector, after which the vector will be set to *first-in-history*.¹⁰

Since the formalism in single-history execution is purely linear, obtaining a quantum algorithm is relatively straightforward¹¹ from these combinations of unitary operators and observation (measurements) operations.

¹⁰ More rigorously, this operator is built as a creation operator of quantum mechanics, but the difference is not very relevant at this level.

¹¹ Building a series of quantum gates to perform the operations of a given unitary operator is dependent on the technology underneath the quantum computer on which the algorithm is performed. Any fully working quantum computer would

In the Case of Preemption. In preemption and single-history CFG (i.e., no control-flow hazard), the slow decay of the hit state also matches what could be observed in a non-perfect quantum computer where phenomena like decoherence can occur. If one fits the progression while the program executes, to the decoherence rate, then the results for WCET with preemptions is a direct result of the computing. Otherwise, if the coherence time is large compared to the computation of the WCET, then it would not be difficult to simulate a decay, as our linear operations in Section 3.4 show.

6.2 Multi-path Execution

This is the case when we think the full power of quantum computing with the superpositions of states can be utilized. For multi-path executions (programs that can be executed differently depending on external data), we will want to retain the information of the execution through multiple possible execution paths. This is future work, because it is required to retain the quantum information in all the possible execution paths or at least all the execution paths that are of interest for WCET evaluations.

While this is not straightforward at all from our simple approach, this can be achievable, at least theoretically, by modifying the operators and adding so-called redundancy qubits to store the quantum information that is lost in our current approach. Nonetheless, the number of redundancy qubits and how to modify the algorithm to that goal is a key point that will require further work. The case of multiple execution paths can be viewed as a generalized double split experiment, the same one that revealed the duality between wave and particles and marked the foundation of quantum mechanics.

7 CONCLUSION AND OUTLOOKS

In this article, we have shown a new approach to the Worst Case Execution Time (WCET) evaluation, and we tested it on several well-known benchmark programs on the LRU cache replacement policy. Our model is based on a variant of static analysis and also on a quantum mechanics analogy, with linear operators and a few non-linear operations to cope with control flow path merge cases. We took advantage of the use of a pseudo-probabilistic base to introduce the preemption as an attenuation factor. This gave us a method to take preemption into account and evaluate an associated level of confidence.

We have shown very good results on the benchmark programs, with less than 1.5% error on number of misses and a good accuracy (usually less than 5%) on WCETs but which is reduced as the number of preemption is growing. These good results are obtained in spite of a quite naive model for miss-delay accounting. We believe we can do as good an evaluation on WCET with finer models of miss/hit cache access timing within the pipeline as we did on miss ratios.

We also showed that the use of models inspired by quantum computing for this case of static analysis can yield a good opportunity for research and results that are accurate (but not absolute) without the hassles of resolving explicitly a NP-hard problem.

For future work, we want to find a complete quantum algorithm to express this problem, particularly by taking the case of multiple CFG into account.

APPENDIX

In this Appendix, we show how a simple example from Huynh et al. (2011) can be handled in our formalism.

provide a minimum set of operators that can allow any unitary operation on the qubits available in the machine to be performed. See, e.g., Nielsen and Chuang (2011) for further details.

```

int condSum(int* A, int x, int *B, int *D,
            short int* C)
{
    int i,j;
    int sum= 0;

    for (i= 0 ; i < 4 ; i++) {
        int a= A[x] ;
        for (j= 0 ; j < 16 ; j++) {
            int b;
            if (a%2 == 0)
                b= B[i*16+j];
            else b= C[i*16+j];
            sum+= D[0] +b ;
        }
    }
    return sum;
}

```

Fig. 6. The C program from Huynh et al. (2011) on the left and its associated CFG analyzed by our tool.

The source code of the program and the basic-block layout as analyzed by our in-house tool is shown in Figure 6. The code was compiled by GCC 5.4.1 using the (-O3) flag, mostly because the resulting CFG is simple to visualize this way.

The resulting binary code can be surprising but it is easy to understand if you know that the inner-most loop is unrolled inside each *if-then-else* branch (blocks 3 and 8) and the value $A[x]$ is tested for each iteration (cond 2), which is a good decision from the compiler, since nothing says it could not be changed between two iterations, e.g., in a parallel task. The loop-exit test is static and detected in *cond 5*, and the loop is executed 4 times: The counter detected by our tool is shown as being incremented from 0 to 3 by step of 1 (link between *cond 5* and *instr. block 1*), and the CFG exits to block 6 on the fourth iteration.

The detected loop is highlighted by a non-white background of the basic-blocks that constitute it. This function makes 79 memory accesses (6 store operations, to save registers on the stack, and 73 read operations, including 6 register value restorations from the stack). The whole accesses span on 19 memory-cache lines in the general case (including 2 lines for the stack).

Our tool¹² concludes that in the non-preemptive case, in a configuration with 1KB data and instruction memory caches, there are 14 misses for instruction cache and 8 for data caches, accounting each for about 75 wait-cycles. The total execution time in the worst-case is 655 cycles with these hypotheses.

The whole program fits easily in the cache, so the first preemption takes a large toll in the execution time, with 27 instruction-cache misses, 12 data-cache misses, and 852 cycles for execution, including 93 stall-cycles nearly entirely due to memory cache misses as there are few jump instructions in the code.

In Figures 7 and 8, we show, respectively, the discrepancy between the exact number of data-cache misses and those given by the model, either with the simple model or with the model

¹²Our in-house tool includes a generic processor simulator and an Abstract Interpretation module that implements our model in relation with the pipeline simulator of the processor simulation so that the pipeline simulation is completely coherent with the processor simulator.

Fig. 7. Evolution of the estimated data-misses in the example. The estimation includes also the graph with 1σ confidence interval.

Fig. 8. Deviation of the base model to the exact number of misses expressed in number of sigma.

including one standard deviation,¹³ and the difference in number of σ between the base model (see footnote) and the real number of misses in the worst case in number of σ .

In the latter case, for one preemption there is a two sigma variation (something rare but can be expected as the program is quite peculiar and very simple with little cache memory occupancy) and globally improve from that. For a high number of preemptions ($n_p \geq 7$), the evaluation becomes pessimistic at one standard deviation above the real case. This can hint that even the 3σ criterion would be sufficient for most of the cases.

¹³It is worth noting that choosing the base model or only one standard deviation from the base model would be considered unreliable. The minimum safe would be 3σ in addition to the base model, and to be safe, one should choose 10σ (although in the case of a low number of memory accesses it could be a very large overestimation).

REFERENCES

- Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. 1996. Cache behavior prediction by abstract interpretation. In *Proceedings of the Static Analysis Symposium (SAS'96)*. 52–66.
- Sebastian Altmeyer and Claire Maiza Burguière. 2011. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *J. Syst. Architect.* 57, 7 (2011), 707–719.
- Gely P. Basharin, Amy N. Langville, and Valeriy A. Naumov. 2004. The life and work of A. A. Markov. *Linear Algebra Appl.* 386, 0 (2004), 3–26. DOI : <https://doi.org/10.1016/j.laa.2003.12.041>
- J. V. Busquets-Mataix, J. J. Serrano-Martin, R. Ors, P. Gil, and A. Wellings. 1996. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceeding of the 2nd Real-Time Technology and Applications Symposium*. 204–212.
- Francisco J. Cazorla, Eduardo Quinones, Tullio Vardanega, Liliana Cucu-Grosjean, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, Luca Santinelli, Leonidas Kosmidis, Codé Lo, and Dorin Maxim. 2012. *PROARTIS: Probabilistically Analysable Real-Time Systems*. Research Report RR-7869. Retrieved from <https://hal.inria.fr/hal-00663329>.
- P. Cousot and R. Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 238–252.
- R. I. Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean. 2013. Analysis of probabilistic cache related preemption delays. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS'13)*. 168–179.
- Christian Ferdinand and Reinhard Wilhelm. 1999. Efficient and precise cache behaviour prediction for real-time systems. *Real-Time Syst.* 17, 2–3 (Nov. 1999), 131–181.
- Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET benchmarks—Past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*.
- Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. 2011. Scope-aware data cache analysis for WCET estimation. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'11)*. IEEE Computer Society, Washington, DC, 203–212.
- Leonidas Kosmidis, Jaume Abella, Eduardo Quinones, and Francisco J. Cazorla. 2013. A cache design for probabilistically analysable real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'13)*. EDA Consortium, San Jose, CA, 513–518.
- C.-G. Lee, J. Hahn, Y.-M. Seo, S.L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. 1998. *Analysis of Cache-related Preemption Delay in Fixed-priority Preemptive Scheduling*. Technical Report, vol 47. IEEE Transaction on Computer.
- A. Markov. 1906. Rasprostranenie zakona bol'shih chisel na velichiny, zavisyaschie drug ot druga. *Izvestiya Fiziko-matematicheskogo obschestva pri Kazanskom universitete*, 2-ya seriya 15, 94 (1906), 135–156.
- Dorin Maxim, Mike Houston, Luca Santinelli, Guillem Bernat, Robert I. Davis, and Liliana Cucu-Grosjean. 2012. Re-sampling for statistical timing analysis of real-time systems. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*. ACM, 111–120.
- Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. 2003. Accurate estimation of cache-related preemption delay. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'03)*. ACM, New York, NY, 201–206.
- Michael A. Nielsen and Isaac L. Chuang. 2011. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, New York, NY.
- I. Puaut and C. Pais. 2007. Scratchpad memories vs locked caches in hard real-time systems: A quantitative comparison. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'07)*. 1–6.
- E. Quinones, E. D. Berger, G. Bernat, and F. J. Cazorla. 2009. Using randomized caches in probabilistic real-time systems. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS'09)*. 129–138.
- H. Ramaprasad and F. Mueller. 2006. Bounding preemption delay within data cache reference patterns for real-time tasks. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*. 71–80.
- Mälardalen WCET Research Group. 2006. WCET Benchmarks. Retrieved from <http://www.mrtc.mdh.se/projects/wcet/benchmarks.htm>.
- Darshit Shah, Sebastian Hahn, and Jan Reineke. 2018. Experimental evaluation of cache-related preemption delay aware timing analysis. In *Proceedings of the 18th International Workshop on Worst-Case Execution Time Analysis (WCET'18) (OpenAccess Series in Informatics (OASIS))*, Florian Brandner (Ed.), Vol. 63. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 7:1–7:11.
- X. Vera, B. Lisper, and Jingling Xue. 2003. Data caches in multitasking hard real-time systems. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS'03)*. 154–165.

Received August 2018; revised March 2019; accepted May 2019