Weighted Execution Time Analysis of Applications on COTS Multi-core Architectures

Hardik Shah, Kai Huang, Alois Knoll

TUM-I1339

Weighted Execution Time Analysis of Applications on COTS Multi-core Architectures

Hardik Shah Kai Huang
Alois Knoll
Department of Informatics VI, Technical University Munich,
85748 Garching, Germany
{shah|huangk|knoll}@in.tum.de

July 26, 2013

Abstract

Commercial off-the-shelf multi-core architectures could significantly reduce costs and time-to-market of hard real-time systems. However, due to the unpredictable interference on the shared memory, the worst case execution time is either non-deterministic or overly pessimistic. Typically, the pessimism originates from the conservative assumption of maximum interference for each memory access. This paper analyzes the shared memory interference under off-the-shelf round-robin arbiter. Rather than estimating a single worst case bound for the execution time, distribution of execution times along their weights is derived. The analysis results are compared with observed execution time of the benchmark applications on a multi-core platform.

1 Introduction

Today's hard real-time applications have become computationally intensive due to the employment of radars, cameras, digital signal processing algorithms etc. Increasing the processor clock frequency to provide high computational power is not a viable solution any more as it increases power consumption beyond manageable limits. Currently, only multi-core architectures provide programming flexibility and high computational power at reasonable power consumption. Additionally, multi-core architectures are naturally suitable for multiple application execution e.g., asymmetric multi-processing.

Adhering to strict timing constraints is a key requirement for hard real time applications such as avionics and automotive. These applications, in combination, consume only 1.6% of globally produced digital ICs. Due to this small market share, employment of Commercial-off-The-Shelf (Cots) components could significantly reduce the costs of these systems. An open problem of using Cots products for hard real-time systems is that such products are usually designed to optimize average case performance. This optimization leads to difficulties in analyzing and estimating the Worst Case Execution Time (Wcet) of applications executing on Cots products.

For instance, the round robin arbiter is typically used in Cots multi-core architectures to orchestrate the accesses of shared resources, e.g., main memory, among multiple cores, due to its simple design, fairness and work conserving property [1, 2, 3]. The access latency to the shared memory under round robin arbitration depends on the arbiter pointer location and accesses from other cores at the time when the access is issued. Both, the arbiter pointer location and accesses from other cores are difficult to characterize for the analysis. Hence, being conservative, worst case latency for every shared memory access is assumed, which results in pessimistic Wcet.

In general, the worst case latency is experienced by only few accesses during execution. Instead of assuming the worst case latency for each shared memory access, our idea is to provide a range of all possible latencies and their weights (importance). With the range of possible latencies, we construct an execution distribution tree based on the application execution trace. The leaves of the tree represent all possible execution times under varying interference and their weights. Based on the required precision of given designs, specific execution time can be chosen as a "likely WCET " which is less pessimistic but still fulfills the design requirement.

We propose a novel approach of weighted execution time analysis of applications on off-the-shelf multi-core architectures with shared memory. Rather than estimating a single worst case bound for the execution time, distribution of execution times along their weights is derived. The detailed contributions of the paper are summarized as follows: a) We present a tree-based construction of the execution time distribution and the weight of each element in the distribution. b) We derive a closed-form expression for computing the weight distribution for a given application trace and formally prove the correctness of the closed form. c) We propose a conservative scheme to weigh the latencies of shared memory accesses. The weights of access latencies are derived without any knowledge of accesses from other cores. d) We conduct case studies on COTS multi-core architecture using the Mälardalen WCET benchmarks and the analysis results are compared with the observed execution time on multi-core prototype on Altera Cyclone III FPGA. The chosen execution time is up to 14% less than the absolute WCET for the given application.

The paper is organized as follows. Sec. 2 describes the closest related work. Sec. 3 provides the required background information. Sec. 4 and Sec. 5 present the core-technique of our analysis. Sec. 6 presents the test architecture and discusses the results. Finally, Sec. 8 concludes the paper.

2 Related Work

There is an overwhelming amount of research work targeting the execution time analysis of applications. There are mainly five approaches towards solving the problem. a) Static WCET analysis b) Hybrid WCET analysis c) Tailored architectures, d) Measurement under high interference, and e) Probabilistic execution time analysis. Instead of comparing our approach with individual research work, we compare our approach on a high level with other approaches.

Static Wcet analysis is an industry proven technique and several tools are available for single core architectures e.g. ait, otawa, chronos. Abstract interpretation and Implicit Path Enumeration Technique (IPET) are the popular methods of estimating the WCET of applications. The static analysis formally guarantees the upper bound on the execution time. However, due to the abstraction and utilization of Integer Linear Programming (ILP) to maximize the execution time estimation, the estimated WCET is highly pessimistic. On the other hand, undesirable events such as timing anomalies and domino effects can be analyzed [4], [5]. However, significant effort is required to model each "new" core.

Although Chattopadhyay et al. [7], Kelter et al. [6], Ding et al. [8] etc have recently presented static WCET analysis of applications on multi-core architectures, we do not compare their approaches with ours since our approach does not relate itself well with the static analysis in general. We assume that the cycle accurate RTL model is the most accurate model of the core. Hence, we record execution traces by executing an application on the RTL model and take the trace as an input for interference analysis. Clearly, like any measurement based technique, our approach cannot analyze events like timing anomalies or domino effects.

The hybrid Wcet analysis approach [9] records traces of execution by inserting instrumentation points[1] at each basic block[2]. Later, these traces are statically analyzed to construct the worst case path considering Maximum Observed Execution Time (MOET) of individual basic blocks. This approach is commercially utilized in the RapiTime [23]. The advantage of this approach is that the architecture is treated as a black box. Hence, WCET of applications on reasonably complex architectures can be measured. Moreover, since the test coverage data is already available from the traces, additional testing for certification is not required making the method cost effective. However, this approach cannot be used for multi-core architectures since the shared memory access latency depends on the interference on the shared resource which could unpredictably increase MOET of

[1]Instrumentation points simply save the performance counter value in the trace to time stamp their execution.

[2]Basic block is defined as a code segment with single entry and single exit points.

each basic block. Thus, the worst case path and the corresponding WCET are invalidated. Our approach complements this method by inserting statically analyzed shared memory interference information for each shared memory access which makes the hybrid approach multi-core capable.

Measurement under uninterrupted interference, as the name suggests, does measurements of execution time under synthetically created uninterrupted interference. Typically, this interference is created by executing a simple code in infinite loop on each co-existing core. The code intentionally accesses memory such that each instruction produces a cache miss. Thus, uninterrupted traffic towards shared memory is created. This method was presented by Fernandez et al. [10], Radojkovic et al. [11], Nowotsch et al. [12] etc. The biggest advantage of this method is that it is virtually free of charge. Additionally, the synthetic code to create the interference is trivial, change in architecture is not at all required and hybrid WCET analysis technique for single core architectures can be used for multi-cores without any modification. However, as we will show in Sec. 3.3, Sec. 5 and Sec. A, such uninterrupted interference does not guarantee either the worst case interference or the worst case execution time. On the contrary, some applications could even benefit and experience less than the average case interference in this case.

As explained above, the unpredictable interference on the shared resources is the biggest challenge of timing analysis on multi-core architectures. To avoid the interference, **tailored architectures**, built especially for predictable timing behavior, are proposed. PRET, CoMPSoC and MERASA are the examples of such architectures. Here, the predictable timing behavior is achieved by time triggered access to the shared resource - PRET [17], by intentionally delaying a shared resource access to the worst latency - CoMPSoC [16] or by providing high priority to the accesses from the safety critical task - MERASA [15]. As explained in Sec. 1, due to the significantly small market share of hard real-time systems, economic feasibility of producing these architectures is yet to be investigated.

The probabilistic execution time analysis is the closest to our approach. It provides an execution time distribution of the application instead of providing a single WCET value. The approach was first presented in Edgar et al. [18] and Bernat et al. [19] and recently investigated by PROARTIS [20] project. The project employs customized components such as random cache [21], [22] which randomly evicts a cache-line upon a cache miss. Thus, unlike popular history based eviction policies such as LRU, FIFO, Round Robin; the execution time of the program does not depend on the execution history. Hence, undesirable events such as domino effects and timing anomalies are eliminated. In the random cache, for every cache access, probability[3] of hit or miss is provided and based on that the execution time distribution is derived. Although our work is inspired by this approach, there are important differences. First, their approach is not used for multi-core architectures, yet. Second, cache is a major component of any processor core and modifications to it are significantly expensive. Third, together with the execution history, the performance advantage of using temporal locality is also eliminated. Hence, significant performance degradation is expected which is yet to be investigated. Our approach uses off-the-shelf components, hence, it is cost effective and performance preserving.

Other closely related work is from Shah et al. [14], [13]. Their approach records execution traces and inserts statically calculated worst case interference on each shared SDRAM access to estimate the WCET. Due to the employment of tailored arbiters, PBS [24] and CCSP [25], built for predictability, there approach yields reasonably tight bounds for highest priority applications. However, as explained before, the economic feasibility of customized components is yet to be investigated.

Pellizzoni et al. [26] uses memory access traces to build arrival curves of concurrently executing applications from each core. Using real-time calculus, interference bound for any access is derived. Thus, the assumption of the worst case interference for each memory access is avoided. However, even an unimportant bug fix in any concurrently executing application invalidates the derived bound and enforces re-analysis. Clearly, this approach restricts updates in all concurrently executing applications and task migration. Collision of memory accesses from concurrently executing

[3]The subtle difference between probability in their approach and weight in our approach is explained in Sec. B

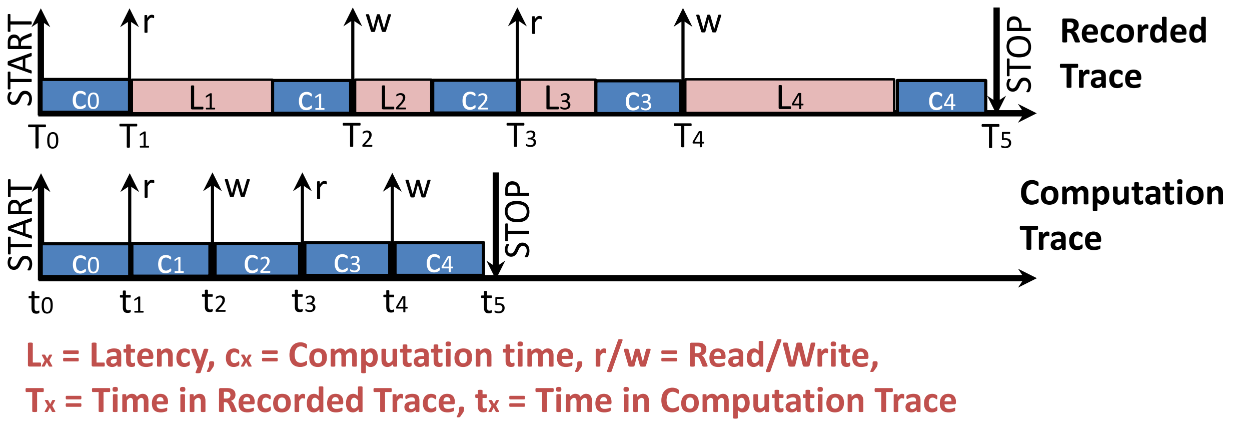
START
r
w
r
w
STOP
Recorded Trace
C0
L1
C1
L2
C2
L3
C3
L4
C4
T0
T1
T2
T3
T4
T5
START
r
w
r
w
STOP
Computation Trace
C0
C1
C2
C3
C4
t0
t1
t2
t3
t4
t5
Lx = Latency, cx = Computation time, r/w = Read/Write,
Tx = Time in Recorded Trace, tx = Time in Computation Trace

Figure 1: Computation Trace

applications are investigated by Lv et al. [27]. Here, model checking is used to determine maximum interference and corresponding WCET. Like [26], this approach also restricts not only application under test, but also the concurrently executing applications. In our approach, applications are analyzed in isolation. Hence, no restrictions in the co-existing applications are enforced.

3 Background

This section provides the necessary background information to facilitate discussion in later sections.

3.1 Computation Trace

The computation trace consists only computation time of a core. In other words, it captures the time when the core is executing only from local registers and caches. Main memory accesses occurring due to cache misses during the execution are represented by instantaneous events. At first, we remove all the latencies associated with the shared resource accesses from the total execution time. However, type of accesses and their precise occurrence time is preserved in computation trace.

Computation trace, depicted in the Fig. 1, can be easily extracted using cycle accurate simulator. For each shared memory access, its occurrence time and type (read or write) as well as its experienced latency are recorded. Later, on the host machine, these latencies are removed and each access is shifted towards left in time. We use computation trace for analysis and insert all possible latencies for each access together with their weights as explained in Sec. 4 and Sec. 5.

It must be noted that the latencies in the recorded trace depend on shared memory interference. When the same application path is executed multiple times, different interference could occur. Hence, latencies differ between multiple execution of the same application path. However, the computation trace remains unchanged[4] when the same path is executed multiple times, provided that each time we start from the same cache state. The computation trace allows us focus on interference analysis ignoring effects of other execution time altering components like caches, pipelines, branch predictors etc.

3.2 System Model

In this paper, we consider a multi-core system with a shared main memory. The shared memory is accessed via cache in a burst fashion when a cache miss occurs. For conflict resolution on the shared memory, a round robin arbiter is used. We assume application level parallelism (asymmetric multi-processing). Thus, each core executes completely independent applications. Although the applications are independent, they affect execution times of each other due to the contention on the shared memory.

[4]In this paper, we assume absence of jitter in occurrence of cache misses and leave the jitter analysis for future work.

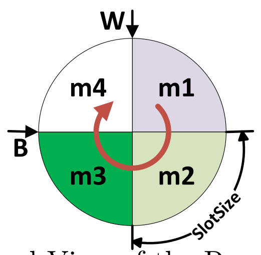

Figure 2: Graphical View of the Round Robin Arbiter

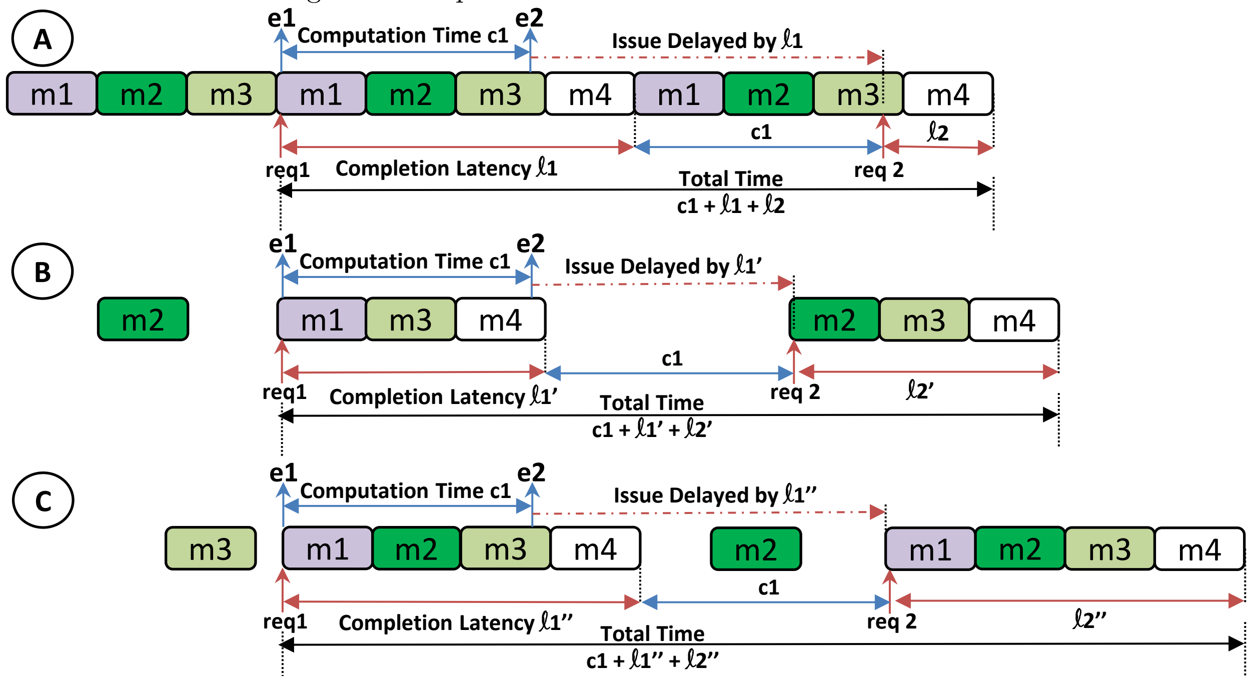

Figure 3: Latencies Under Round Robin Arbitration

Each core is equipped with an instruction cache and a data cache with unknown, but deterministic, eviction policy e.g. FIFO, LRU, pLRU etc. The deterministic eviction policy allows us to assume constant computation trace for any number of execution runs, provided that each time we start from the same cache state. On the contrary, random eviction policy randomly picks a cache-line to evict. Hence, in some execution run, if frequently visited cache-lines are evicted, there will be more number of total cache misses than if less frequently visited cache lines were evicted. Thus, the computation trace changes during multiple execution of the same path when the random cache is used.

3.3 Work Conserving Round Robin Arbiter

In this subsection we will analyze different scenarios of memory access latencies under the Round Robin (RR) arbitration scheme. Under the RR scheme, the shared resource contenders are assigned fixed number of slots, depending on their bandwidth requirements, in a virtual ring as depicted in Fig. 2. For simplicity, we consider, one slot per contender. The figure shows four contenders (master1, master2, master3 and master4) in the ring. Here, we assume that these contenders are processor cores executing independent applications. *Throughout the paper we use master and core terms interchangeably.*

The arbiter continuously searches for a master which wants to access the shared resource in a clock-wise direction. We call this master an active master. As soon as an active master is encountered, it is granted the shared resource for a predefined maximum number of clock cycles (`SlotSize`). The `SlotSize` is big enough to accommodate the burst issued for one cache-line fill. After the granted master finishes its burst access, the search process resumes from the next slot in the ring. Thus, the shared resource is always occupied as long as there is at least one active master

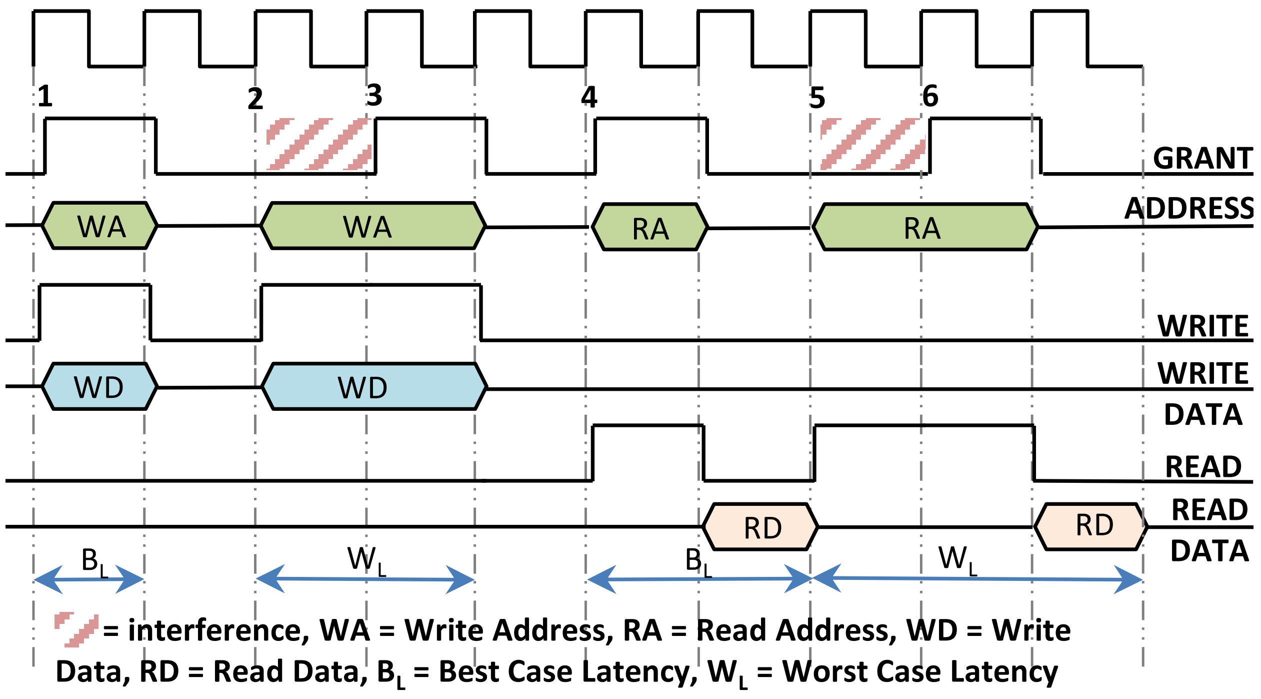
1
2
3
4
5
6
GRANT
ADDRESS
WA
WA
RA
RA
WRITE
WD
WD
WRITE DATA
READ
RD
RD
READ DATA
B_L
W_L
B_L
W_L
▨ = interference, WA = Write Address, RA = Read Address, WD = Write Data, RD = Read Data, B_L = Best Case Latency, W_L = Worst Case Latency

Figure 4: Read–Write Latencies

(hence the name, "*work conserving*").

Now let us assume that the application-under-test is executing on `m4` and `SlotSize` is same for all masters. For this architecture, an access request from `m4` experiences the worst case completion latency ($4 \times SlotSize$) if it is issued when the arbiter pointer is at **W** in Fig. 2 *and* all other masters utilize their slots. Similarly, an access request experiences the best case completion latency ($1 \times SlotSize$) if it is issued when the arbiter pointer is at **B** in the figure.

Fig. 3 depicts various interference scenarios. The computation trace of the application-under-test represents two cache miss events, `e1` and `e2` in the figure. The computation time between the two events (cache misses) is represented by `c1`.

Scenario **A** illustrates latencies in the presence of uninterrupted interference from other masters. In other words, the interfering masters continuously request an access to the shared memory. Here, the event `e1` produces an access request `req1`. When the request is issued, the arbiter has just scheduled `m1`. Since the interfering masters are continuously requesting, `m1` to `m3` utilize their slots. Thus, the request completes after the worst case latency $l1 = 4 \times SlotSize$. After `req1` is served, the master does computation for `c1` and issues an access request `req2` associated to event `e2` in the computation trace. However, during the time `c1`, `m1` to `m3` keep on accessing the shared memory and moving the arbiter pointer. Thus, when `req2` is issued the arbiter pointer is not at **W** of Fig. 2. Hence, `req2` has completion latency of $l2$ where $l2 << l1$. In this scenario, although the co-existing masters `m1` to `m3` continuously accessed the shared memory, each request from `m4` does not experience the worst case latency.

In scenario **B** and **C**, traffic generated by the interfering masters is sparse. However, in both the cases the total time is larger than the total time of scenario **A**. Similarly, other scenarios where the total time is less than that of scenario **A** can be illustrated. Scenario **C** is the absolute worst case interference where both the requests are interfered by all other masters in the system. However, in a typical multi-core execution, the completion latency of accesses is in the range $[B_L, W_L]$, where $B_L = 1 \times SlotSize$ stands for the best case completion latency and $W_L = \#Masters \times SlotSize$ stands for the worst case completion latency.

From the above discussion it is clear that the access latency under work conserving round robin arbiter depends on the arbiter pointer at the time of issue and the activity of other masters at the time of issue. Since the arbiter pointer location depends on the activity of masters, under sporadic

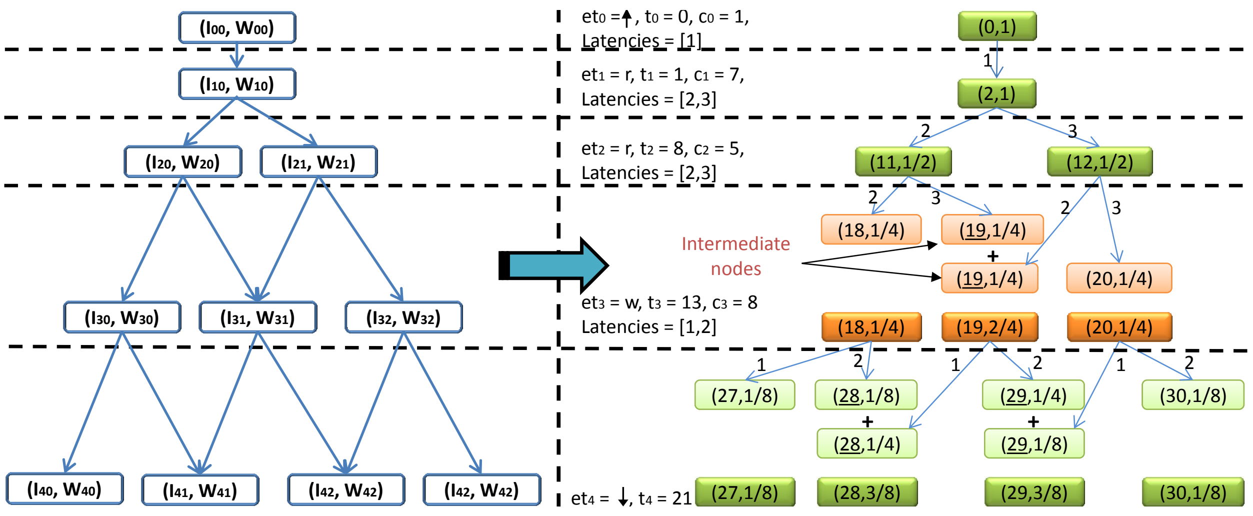

Figure 5: Execution Time Tree

interference, the completion latency experienced by any access request is in the range $[B_L, W_L] \subset N$ and all numbers in this range have equal weights (equal probability or equal importance). The assumption of having equal weights for all numbers in the latency range is valid only if the interfering cores are accessing sporadically. In Sec. 5, we analyze latencies when interference is not sporadic. For now, we consider sporadic interference to explain our approach.

4 Weighted Execution Time

In this section, we describe our tree based approach to derive execution time distribution and weight of each element in the distribution. For each shared memory access, we consider, it could experience any latency in the range $[B_L, W_L]$. Thus, each access could delay the issue of all subsequent accesses by any value in the range $[B_L, W_L]$. Considering that, we create tree where each node represents issue time of particular access. The leaves (nodes on the last level) of the tree represent execution time distribution.

4.1 Illustrative Example

In this section, we explain our technique with the help of an example. For simplicity, consider a simple dual core system with a shared memory. Both cores are equipped with a deterministic cache. The cache-line size is only one word, hence, $SlotSize = 1$ for both the cores under RR. The application-under-test is mapped to one of these cores and encapsulated by profiling macros. The profiling macros provide the start(↑) and finish(↓) events to the simulator between which the application executes, hence, cache misses occurring between only these events are analyzed. These events can be realized by setting a local register to flag an event. We assume that these start/stop events consume constant time of one processor clock cycle.

The interference on the shared memory is explained in Fig. 4. Here, the typical difference between read and write accesses are exhibited. At **point 1**, a write access is requested and it is granted immediately. Here, write data is presented together with the write access request, hence, write operation completes in one clock cycle ($B_L^w = 1$). At **point 2**, the write access request is interfered by the other core. Since there are only two cores in the system and $SlotSize = 1$, any write access completes, by the latest, in two clock cycles ($W_L^w = 2$). For a read access request (at **point 4**), the read data is available one clock cycle after the read access request is granted. Hence, the minimum time to complete a read operation is two clock cycles ($B_L^r = 2$). At **point 5**, the read access is interfered by the other core. Hence, the maximum time to complete a read access is 3 clock cycles ($W_L^r = 3$).

For the analysis, we define the parameters listed in table 1. From now onwards, by *latency* we mean *completion latency*.

Parameter	Symbol	Value
Read Latency Range	$[B_L^r, W_l^r]$	[2,3]
Cardinality of Read Latency Range	C^r	2
Write Latency Range	$[B_L^w, W_L^w]$	[1,2]
Cardinality of Write Latency Range	C^w	2
i^{th} Event Type	et_i	$r, w, \uparrow, \downarrow$
Worst Case Latency of i^{th} event	$W_L^{et_i}$	-
Best Case Latency of i^{th} event	$B_L^{et_i}$	-
Cardinality of i^{th} event Latency Range	C^{et_i}	-
Start Event	$\uparrow$	-
Read Event	r	-
Write Event	w	-
Stop Event	$\downarrow$	-
Occurrence time of i^{th} Event in Computation Trace	t_i	-
Computation time between i^{th} and $(i+1)^{th}$ Events	c_i	$t_{i+1} - t_i$
Computation trace length	N	

Table 1: Parameters

Let us assume that a computation trace is given, $e = \{\uparrow, r, r, w, \downarrow\}$ with the associated time $t = \{0, 1, 8, 13, 21\}$ and computation time $c = \{1, 7, 5, 8\}$. Here, the length of the trace, $N = 5$. As shown in the Fig. 5, we can build an execution time tree for the given trace. The i^{th} level of the tree represents issue time profile of the i^{th} event and contains j nodes. Each of j nodes represents issue time of the i^{th} event, I_{ij}, and the weight of the issue time, W_{ij}.

The root node starts with e_0, $\uparrow$ event, which occurs at $t_0 = 0$. Since there are no previous events which could delay the issue of $\uparrow$, it will be issued immediately with weight 1. Thus, the root node is represented by a duple $(I_{00} = 0, W_{00} = 1)$ where I_{00} is the issue time of the root node and W_{00} is the weight of I_{00}.

For e_1, a read access, the issue time is delayed by the completion latency of e_0. The completion latency of the $\uparrow$ and $\downarrow$ is assumed to be constant - one clock cycle. Thus, issue time of the read access $I_{10} = I_{00} + c_0 + 1 = 2$. Moreover, the completion time of e_0 is constant, hence, there is only one node for e_1 event which has weight of 1.

For e_2, issue time depends on the completion time of e_1. As depicted in Fig. 4, a read access event (e_1) can be completed in either 2 or 3 clock cycles with equal weights under sporadic interference. Hence, the issue of e_2 is delayed by either 2 or 3 clock cycles. We create C^r $(= 2)$ number of nodes for the e_2 to represent the issue times, $I_{20} = I_{10} + c_1 + 2 = 11$ and $I_{21} = I_{10} + c_1 + 3 = 12$. Since, these two nodes inherit weight from their parent node and they themselves have equal weight, $W_{20} = W_{10} \div C^r = 1/2$ and $W_{21} = W_{10} \div C^r = 1/2$.

Similarly, nodes for e_3 can be created. Here, two nodes have the same issue time (underlined). Hence, these nodes are merged to one node and their weights are summed up. Note, that the e_3 is write type, hence, it has completion latencies of 1 and 2.

The issue times for the finish event, $\downarrow$, are considered as execution times of the application the event encapsulates. Thus, for our computation trace e, the possible execution times are {27, 28, 29, 30} with the associated weights of {1/8, 3/8, 3/8, 1/8}.

4.2 Formal Analysis

From the tree it is clear that the at each level, the issue time through nodes are sequentially increasing integers. This is due to the fact that $[B_L^r, W_L^r]$ and $[B_L^w, W_L^w]$ are sequentially increasing integer ranges. Hence, at any level i, the issue times of j^{th} node and x^{th} node is bounded by the following relation,

$$I_{(i)(j+x)} = I_{ij} + x, j < x \tag{1}$$

As shown in Fig. 6, the Best Case Execution Time (BCET) and the Worst Case Execution Time (WCET) can be calculated by traversing through the left and right sides of the tree, respectively. For $c_(-1) = 0$,

$$BCET = I_{(N-1)(0)} = \sum_{i=0}^{N-2} (B_L^{et_i} + c_{i-1}) \tag{2}$$

$$WCET = I_{(N-1)(\lambda-1)} = \sum_{i=0}^{N-2} (W_L^{et_i} + c_{i-1}) \tag{3}$$

Here, λ is the total number of nodes in the last level.

$$\lambda = 1 + WCET - BCET \tag{4}$$

Putting $W_L^{et_i} = B_L^{et_i} + C^{et_i} - 1$ in equation 3,

$$\lambda = 1 + \sum_{i=0}^{N-2} (C^{et_i} - 1) \tag{5}$$

According to [28], the variability of execution time in percentage is defined as,

$$variability = (1 - \frac{BCET}{WCET}) \times 100 \tag{6}$$

From equations (2), (3) and (6), it is clear that the variability of execution time depends on,

1. Length of the trace (number of cache misses). The longer the trace, the higher the variability
2. Difference between $W_L^{et_i}$ and $B_L^{et_i}$. If $W_L^{et_i} >> B_L^{et_i}$ then the variability is high.
3. Computation time. If $\sum c_i >> \sum W_L^{et_i}$ then variability is low. These applications are computation intensive and their BCET and WCET are dominated by the computation time, instead of memory access latencies.

Fig. 6 depicts cut off and cut off Execution Time (cET). The cut off of any application is decided based on the consequences of missed deadline. For example, in cars, cut off of break application must be much lower than the cut off of climate control application. Missing a deadline in climate control application could compromise comfort of the passengers while missing a deadline in break application could endanger lives of passengers and people in the proximity. Hence, higher assurance (lower cut off) from the break application is expected.

To compute cET, we traverse the last level of the tree from WCET to BCET (right to left). While traversing the tree, we accumulate weights of each node and stop traversing as soon as the cut off is reached. In the figure, we chose cut off = 1/2. The cET means that one out of two times the application is executed under varying interference, the execution time could be equal or greater than 29.

4.3 Computation of Weight using Convolution

In this subsection we prove that the weights of issues times at level $(i+1)$ is convolution of weights of issues times at level i and impulse function O.

For simplicity, we consider $c_i = 0$ first. The latency ranges $[B_L^r, W_L^r]$ and $[B_L^w, W_L^w]$ are modeled as an arithmetic sequence, i.e., $O = \{o_k\}, \forall k\ o_{k+1} - o_k = d, d \in N$. Now the tree construction presented in the previous section can be considered as constructing an n-ary tree with root node of issue time

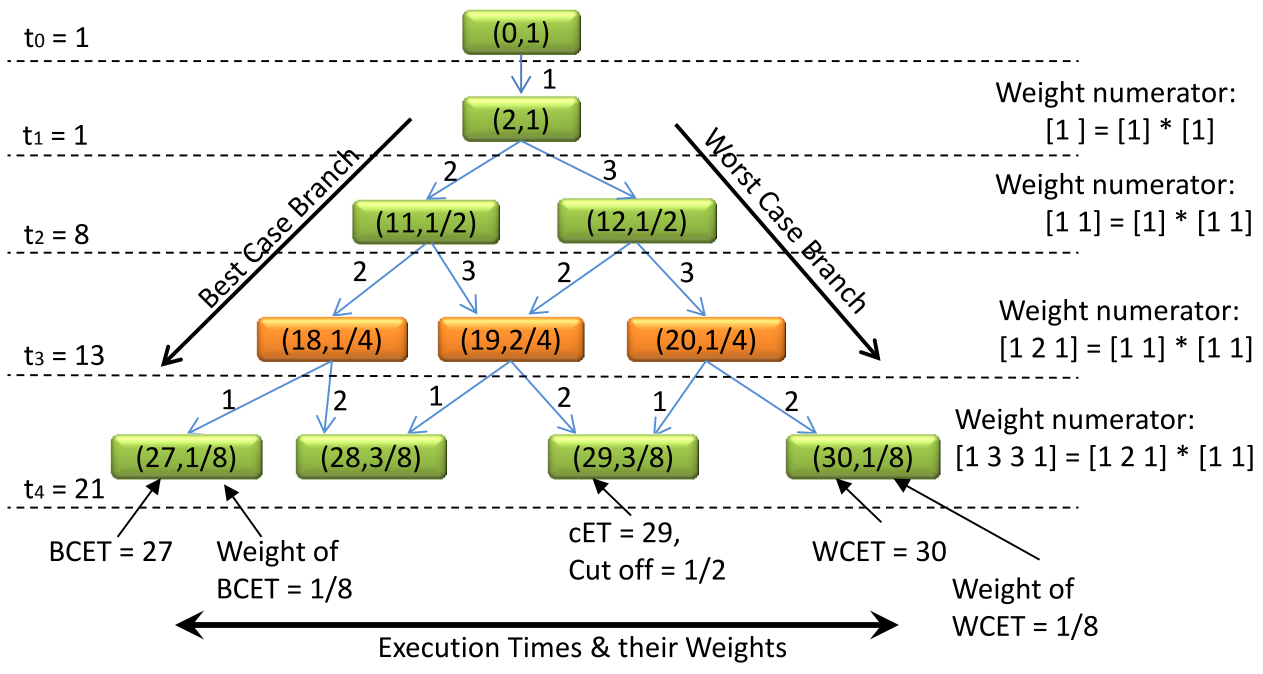

Figure 6: Execution Time Tree with Convolution

$I_0 = 0$ and weight $W_0 = 1$. We will show how to extend the results for $c_i > 0$ and $[B^r_L, W^r_L] \neq [B^w_L, W^w_L]$ afterward.

Let $\{I_{i,j,k}\}_{k=0}^{|O|-1}$ denotes the sequence of issue times of the children of the j^{th} node at the i^{th} level of the tree. Here, k denotes the index of *unmerged* children of the j^{th} node. Let $\{I_{i,j}\}$ denotes the sequence of issue times of nodes at level i. Since each node in the constructed tree has a unique issue time, $I_{i,j}$ also refers to node $n_{i,j}$ in this section. Before showing the main result, we first prove a few lemmas.

Lem. 1 *For any i^{th} level of the constructed tree,*

$$\{I_{i,j+1,k}\}_{k=0}^{|O|-1} = d + \{I_{i,j,k}\}_{k=0}^{|O|-1} \tag{7}$$

where $d = o_{k+1} - o_k, \forall k$.

Proof: We prove by induction. Lets first check the children of the root node. The children of the root node are O. Therefore, by the definition it is an arithmetic sequence. Now assume at level i, $\{I_{i,j}\}$ is an arithmetic sequence. For node $n_{i,j}$, it has $|O|$ children, the issue time of these children are:

$$\{I_{i,j,k}\}_{k=0}^{|O|-1} = I_{i,j} + \{o_k\}_{k=0}^{|O|-1} = \{I_{i,j} + o_k\}_{k=0}^{|O|-1} \tag{8}$$

For node $n_{i,j+1}$, the issue time of its children is:

$$\begin{aligned}\{I_{i,j+1,k}\}_{k=0}^{|O|-1} &= I_{i,j+1} + \{o_k\}_{k=0}^{|O|-1} = d + I_{i,j} + \{o_k\}_{k=0}^{|O|-1} \\ &= d + \{I_{i,j} + o_k\}_{k=0}^{|O|-1} = d + \{I_{i,j,k}\}_{k=0}^{|O|-1}\end{aligned} \tag{9}$$

The lemma therefore holds. □

Lem. 2 *For any level i of the constructed tree, the issue time $I_{i,j}$ of the nodes is an arithmetic sequence.*

Proof: From Lemma 1, the issue time of children of node $n_{i,j+1}$ can be considered as a right shift of those for node $n_{i,j}$. Therefore, we get $I_{i,j,k} = I_{i,j+1,k-1}, \forall j, k$. As by construction, the issue time for each node is unique. Hence, only one node is generated for children of different nodes that have the same latencies. Therefore, we get $\{I_{i+1,j}\}$ is arithmetic sequence with distance d. □

Lem. 3 *The number of nodes for level $i+1$ is $|\{n_{i,j}\}|+|O|-1$.*

Proof: As shown in Lemma 2, the issue time of children for node n_{i+1} can be seen as right shift by d of n_i. For the first node, $|O|$ children will be created, for all the rest $|\{n_{i,j}\}|-1$ nodes, one new issue time (node) will be created for each node, i.e., $|\{n_{i,j}\}|-1$ new nodes. Therefore, the length of level $i+1$ is $|\{n_{i,j}\}|+|O|-1$. □

To illustrate above results, we show an example in Fig. 7. Suppose there are three nodes at the i^{th} level and three different latencies in O. Each node $n_{i,j}$ has three children. Some of these children share a same issue time. For instance, the third children of node $n_{i,1}$, the second children of node $n_{i,2}$, and the first children of node $n_{i,3}$ have the same issue time, which can be merged into one single node during the construction of the $i+1$ level of the tree.

$$\begin{array}{c|ccccc} & n_{i+1,0} & n_{i+1,1} & n_{i+1,2} & n_{i+1,3} & n_{i+1,4} \\ n_{i,0} & o_0+I_{i,0} & o_1+I_{i,0} & o_2+I_{i,0} & & \\ n_{i,1} & & o_0+I_{i,1} & o_1+I_{i,1} & o_2+I_{i,1} & \\ n_{i,2} & & & o_0+I_{i,2} & o_1+I_{i,2} & o_2+I_{i,2} \end{array}$$

Figure 7: Example for Issue Time Computation

Thm. 1 *The issue time of level $i+1$ of the tree can be computed from the issue time sequence of level i by a min-plus convolution*

$$I_{i+1,j} = \min_k \{I_{i,k} + o_{j-k}\}, \tag{10}$$

by setting the undefined values to $+\infty$.

Proof: This can be directly derived from Lemmas 1–3, as $I_{i,k}+o_{j-k} = I_{i,k-1}+o_{j-k+1}$. □

Theorem 1 tells how the issue time of nodes are computed. More importantly, it shows how the tree can be constructed, as the constructed nodes have unique latencies. With this knowledge, we state the main result of this section:

Thm. 2 *The Weights of the nodes at level $i+1$ is the convolution of level i with the impulse function O, i.e.,*

$$W_{i+1,j} = \sum_k W_{i,k} \cdot W_{o_{j-k}} \tag{11}$$

by setting the undefined values to 0.

Proof: Functions (10) and (11) are isomorphic as they are both convolution operation. The min-plus convolution in (10) defines the structure of the tree, i.e., the parent-child relation between nodes. The function 11) reuses the constructed structure, only replacing the node merging by summing up the weights of nodes with the same issue time, as shown in Figure 8. Therefore, the theorem holds. □

$$\begin{array}{c|ccccc} & n_{i+1,0} & n_{i+1,1} & n_{i+1,2} & n_{i+1,3} & n_{i+1,4} \\ n_{i,0} & W_{o_0}W_{i,0} & W_{o_1}W_{i,0} & W_{o_2}W_{i,0} & & \\ n_{i,1} & & W_{o_0}W_{i,1} & W_{o_1}W_{i,1} & W_{o_2}W_{i,1} & \\ n_{i,2} & & & W_{o_0}W_{i,2} & W_{o_2}W_{i,1} & W_{o_2}W_{i,2} \end{array}$$

Figure 8: Example for Weight Computation

Cor. 1 *Theorems 1 and 2 hold for $c_i > 0$*

Proof: According to the multilinearity of convolution, i.e., $c(f_1 * f_2) = (cf_1) * f_2 = f_1 * (cf_2)$, the corollary holds. □

Cor. 2 *Theorems 1 and 2 hold for* $[B_L^r, W_L^r] \neq [B_L^w, W_L^w]$

Proof: As the construction of the next level of the tree is independent from the previous construction, the O can be different for each construction. Therefore, different O can be applied during the construction of the tree. □

Note that sequence of weights for latencies does not need to be an arithmetic sequence. In the next section, we will show how to compute the weights for different latencies.

5 Analysis Under Deterministic Interference

The example given in Section 4 uses equal weights for all latencies in the range $[B_L, W_L]$. The equal weights may not always be true, as the interfering cores could have stuck-at faults on their request lines or could be doing big DMA transfers, etc. In all these cases, the interfering cores send many access requests in a relatively short period of time which results in the scenario **A** of Fig. 3. This section presents a technique to conservatively weigh the latencies of the shared memory accesses. Here, we first determine the maximum latency experienced by the application when interfering cores are continuously accessing the shared memory. Then weight allocated to this latency is increased at the cost of weights of lower latencies.

5.1 Deterministic Interference Basics

We assume that an interfering core either do uninterrupted accesses or does not do any access at all (turned off). Hence, the rotation of the arbiter pointer becomes deterministic as depicted in Fig. 9 (Note that if at least one core does arbitrary accesses than the rotation of the arbiter pointer becomes non-deterministic and the analysis of the previous section applies). We call this type of interference "α interference". Here, $\alpha = 3$ if the uninterrupted interference is produced by three masters, $\alpha = 2$ if uninterrupted interference is produced by two masters and so on.

Now, let us reconsider scenario **A** of Fig. 3, $\alpha = 3$. The latency experienced by $req2$ can be easily calculated using the following equation.

$$l_2 = 4 \times SlotSize - \{c_1 \bmod (3 \times SlotSize + 1)\} \tag{12}$$

Denote this deterministic latency for i^{th} request by DL^i. Equation (12) can be generalized as follows,

$$DL_\alpha^i = (\alpha + 1) \times SlotSize - \{c_{(i-1)} \bmod (\alpha \times SlotSize + 1)\} \tag{13}$$

Let $\Theta_\alpha^{(i-1)} = \{c_{(i-1)} \bmod (\alpha \times SlotSize + 1)\}$, we have:

$$DL_\alpha^i = (\alpha + 1) \times SlotSize - \Theta_\alpha^{(i-1)} \tag{14}$$

Consider average of all DL_α^i values is $\overline{DL_\alpha}$ and the average of all Θ_α^i values is $\overline{\Theta_\alpha}$ (note that $\overline{DL_\alpha}$ and $\overline{\Theta_\alpha}$ represent average values over entire execution path). Hence, equation (14) can be re-written as,

$$\overline{DL_\alpha} = (\alpha + 1) \times SlotSize - \overline{\Theta_\alpha} \tag{15}$$

Equation (15) implies that as $\overline{\Theta_\alpha} \to \alpha \times SlotSize$, $\overline{DL_\alpha} \to 1 \times SlotSize = B_L < A_L$. Thus, certain applications could, on an average, achieve less than the average case latency $(A_L = (B_L + W_L) \div 2)$ for their accesses due to the α interference. We consider these applications as being benefited from the α interference since round robin is considered to be a fair algorithm. Under fairness property, accesses should experience, on an average, average case latency.

The intuition behind this observation is depicted in the Fig. 9. If an application has a $\overline{\Theta_\alpha}$ in the favorable region, the $\overline{DL_\alpha}$ is less than the A_L. Here, the issue of request occurs close to the next scheduling opportunity. Similarly, if an application has a $\overline{\Theta_\alpha}$ in the unfavorable region, the $\overline{DL_\alpha}$ is more than A_L. We consider these applications as being penalized from the α interference.

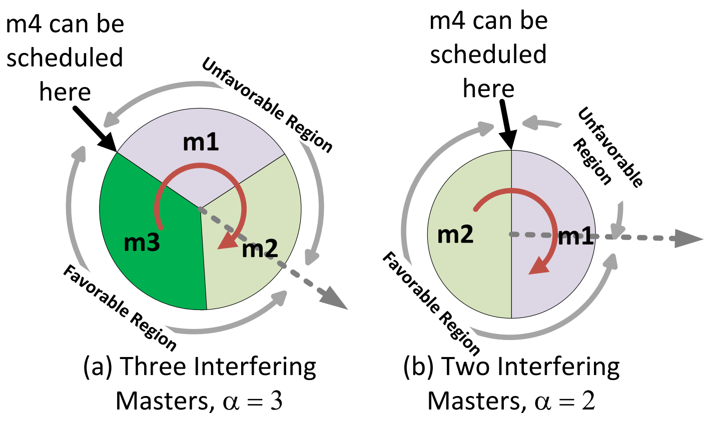

Figure 9: Rotation of Arbiter Pointer under Uninterrupted Interference.

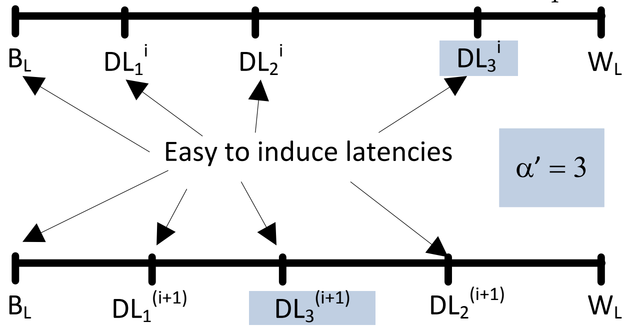

Figure 10: Easy to induce latencies for i^{th} and $(i+1)^{th}$ accesses

As an example, consider an application with $\overline{\Theta_\alpha} = 0$ for a particular α interference. It means, under α interference, this application on an average requests just after its scheduling opportunity. Hence, it has to wait for $\alpha \times SlotSize$ to be scheduled. After being scheduled, it needs another $SlotSize$ to complete. Thus, it experiences completion latency of $(\alpha+1) \times SlotSize$ for majority of its requests. In general, application that does accesses close to their scheduling point in the unfavorable region is vulnerable to high latencies due to interference.

The $\overline{DL_\alpha}$ is an important factor to classify applications as being vulnerable to high latencies or being not vulnerable to high latencies due to the high interference. In the following section, we will use it to compute the weights of latencies.

5.2 Allocating Conservative Weights

With the help of $\overline{DL_\alpha}$, we can derive which latency should carry higher weight. To do that, we apply "reverse engineering" approach. Using interfering cores, we want to induce an average latency, $\bar{x} \in [B_L, W_L]$ for the application-under-test. Note that the application-under-test and $\bar{x}$ are given, and $\bar{x}$ is the average latency over all accesses during application execution. Our task is to design interfering applications such that when these applications run together with the application-under-test the average latency experienced by all accesses of the application-under-test is $\bar{x}$.

This task is very challenging except for given $\bar{x} = B_L$ and $\bar{x} = \overline{DL_\alpha}$. If given $\bar{x} = B_L$, we turn off all interfering cores ($\alpha = 0$). Thus, the application-under-test experiences B_L for each access. Hence, average latency, $\bar{x} = B_L$, holds. If given $\bar{x} = \overline{DL_\alpha}$, we turn on only α number of interfering cores and execute an application on them that will generate back-to-back cache misses. Design of an application that will generate back-to-back cache misses is explained in Sec. 6 and in [10], [11], [12].

To induce latencies other than B_L and $\overline{DL_\alpha}$, the interfering cores must analyze computation trace of the application-under-test thoroughly and do coordinated accesses with the accesses from application-under-test. Moreover, phase change of even 1 clock cycle in application-under-test will not produce the required $\bar{x}$ latency. Note that executing the same application on interfering cores will not produce worst case interference (Fig. 16 in the appendix). Hence, it is very difficult to induce any other latency except B_L and $\overline{DL_\alpha}$.

Let us consider $\overline{DL} = max(\overline{DL_\alpha}), \alpha = 1,2,3....$ Thus, $\overline{DL}$ is the largest possible average latency

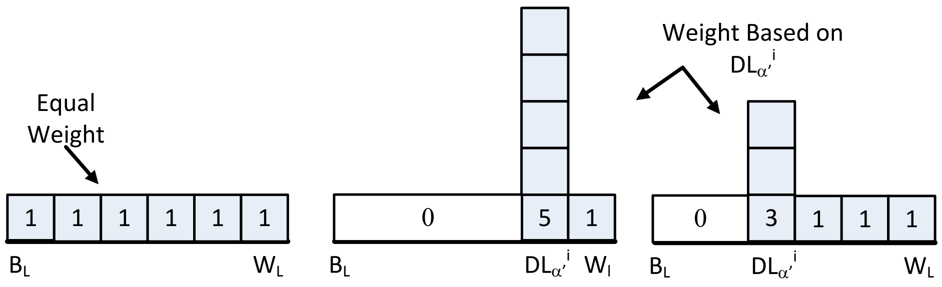

Figure 11: Convolution Vector for Conservative Weight Allocation

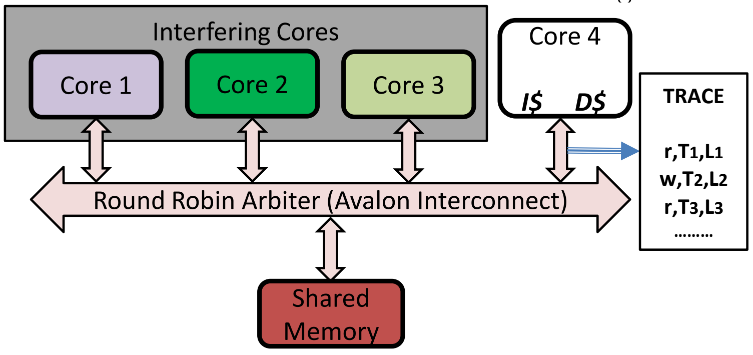

Figure 12: Test Setup

which can be easily induced in application-under-test and the associated α is the number of interfering masters needed to induce $\overline{DL}$. We denote the α associated to the $\overline{DL}$ as α'. We know that $\overline{DL}$ is the average of elements $DL^i_{\alpha'}$. Fig. 10 depicts these parameters for i^{th} and $(i+1)^{th}$ access requests of application-under-test. As shown in the figure, it may be possible that for some accesses, $DL^i_{\alpha'} < DL^i_{\alpha}$. However, it is these $DL^i_{\alpha'}$ latencies which leads to the maximum of all easy-to-induce latencies. Hence, we give highest weight to the $DL^i_{\alpha'}$ of all accesses.

To stay conservative, only higher latencies are allocated more weight at the cost of weights of the lower latencies and not the other way around. Our conservative weighing scheme is based on the following:

1. Since $DL^i_{\alpha'}$ is the easy-to-induce latency and $\overline{DL}$ is the maximum of all easy-to-induce average latencies, $DL^i_{\alpha'}$ is allocated the highest weight.
2. Since $DL^i_{\alpha'}$ is easy-to-induce, latencies lower than $DL^i_{\alpha'}$ are not interesting any more. Hence, their entire weights are added to the weight of $DL^i_{\alpha'}$.
3. Weights of latencies higher than $DL^i_{\alpha'}$ are unchanged from the equal allocation. This captures the fact that if the interference is not the "α interference", some sporadic interference could induce more than $DL^i_{\alpha'}$ latencies.

Fig. 11 depicts the resulting convolution vector. Note that as $DL^i_{\alpha'}$ approaches W_L, its weight increases which in turn increases weight density towards the worst case latencies. This captures the fact that under α' interference, if the application-under-test does many accesses just after its scheduling point (Fig. 9), it is highly vulnerable to the worst case latencies.

6 Case Studies

This section provides case studies. The experimental results are derived from Altera Cyclone III FPGA.

6.1 Test Setup

Fig. 12 depicts our test architecture. The architecture contains four 32 bit NIOS II Floating point cores with instruction and data caches. Each cache has cache lines of 32 bytes and the total size of the cache is 1 KB. An on-chip SRAM serves as a main memory. The Altera Avalon interconnect by default provides round robin arbitration for the shared resources [1]. The interconnect is 32 bits wide. Hence, it takes minimum 1×8 clock cycles to write back a cache line of 32 bytes. However due to the interference on the main memory, the write back may take up to $4 \times 8 = 32$ clock cycles. Thus, $[B_L^w, W_L^w] = [8, 32]$. As depicted in Fig. 3, $[B_L^r, W_L^r] = [9, 33]$.

Altera provides encrypted cycle accurate simulation models of processors and memories. We used these models to record trace (Sec. 3.1) of applications from the Mälardalen WCET benchmark. The benchmark applications run on the Core 4. In our setup, instruction and data cache share a single master port to connect to the shared main memory (via the RR arbiter). Measurement for the recorded trace is probed from this single master port. Thus, the recorded trace captures consolidated instruction and data cache misses. We processed the recorded trace and removed latencies from it to create computation trace.

6.2 Experiment I

In this experiment the computation traces of applications from Mälardalen WCET benchmark and equations (2), (3) and (5) were used to derive execution times. The conservative weights of execution times are calculated using convolution vector as explained in Sec. 5. Using execution times and their weights BCET, WCET and cET are extracted. We considered 10^{-8} as a cut off to calculate cET. The Maximum Observed Execution Time (MOET) is considered after executing the application-under-test for multiple times under varying interference. Obviously, after each iteration, caches are flushed. The varying interference is created as follows.

Each interfering core either executes one of the applications of the Mälardalen benchmark or a synthetic application in an infinite loop. In the case of benchmark application, the cache miss pattern is sporadic while in the case of the synthetic application, back-to-back cache misses are generated. The synthetic application accesses an integer array of 512 elements (2 KB - double the size of data cache) in the stride of 8 integers (32 B - cache-line size) ensuring a new cache-line is accessed each time. The access type is write type, hence, for every access, dirty line must be written back to the main memory and the requested line is fetched from the main memory. Thus, we get two back-to-back accesses (write and read) to the main memory. This is exactly the same method used by [10], [11], [12] to create uninterrupted interference.

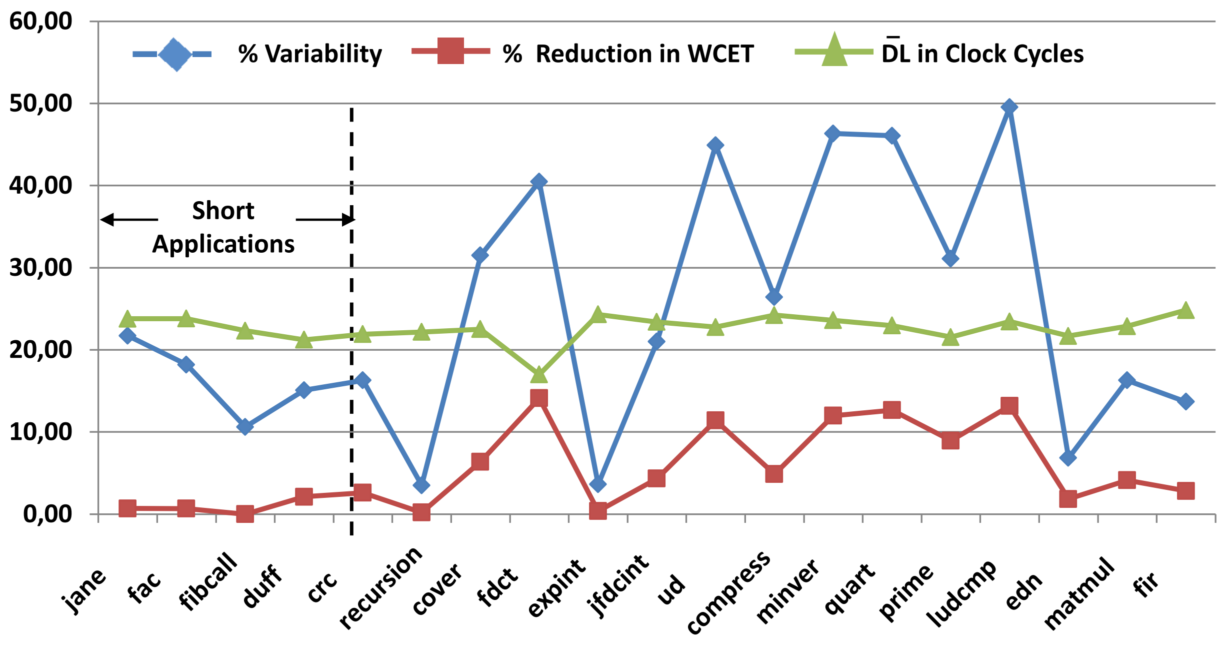

Figure 13: Results

Table 2 and Fig. 13 illustrate results of this experiment. The applications are arranged in the ascending order of their BCET. The "Reduction in WCET" depicts how much less execution time

]

Bench-mark	Execution Times			cET	Reduction in WCET	MOET	$\overline{DL}$
	BCET	WCET	Variability				
jane	692	884	21.72%	878	0.68%	831	23.78
fac	971	1187	18.19%	1179	0.67%	1127	23.80
fibcall	1012	1132	10.60%	1132	0.00%	1108	22.33
duff	4194	4938	15.07%	4834	2.11%	4617	21.22
crc	5061	6045	16.28%	5888	2.60%	5626	21.91
recursion	6644	6884	3.49%	6871	0.19%	6808	22.18
cover	8356	12196	31.48%	11418	6.38%	10676	22.51
fdct	13417	22537	40.47%	19354	14.12%	16844	16.99
expint	16005	16605	3.61%	16543	0.37%	16401	24.31
jfdcint	20696	26192	20.98%	25056	4.33%	24008	23.40
ud	21196	38476	44.91%	34088	11.40%	31825	22.77
compress	21214	28822	26.40%	27420	4.86%	26296	24.22
minver	100147	186619	46.33%	164253	11.98%	156133	23.59
quart	128077	237469	46.06%	207410	12.65%	195444	22.95
prime	138729	201369	31.10%	183331	8.95%	173956	21.55
ludcmp	249765	494973	49.54%	429782	13.17%	405503	23.44
edn	313973	336989	6.83%	330811	1.83%	326928	21.68
matmul	1138798	1360318	16.28%	1304203	4.12%	1273093	22.86
fir	1221979	1415467	13.67%	1375517	2.82%	1340169	24.83

Table 2: Results for Exe. Times in Clock Cycles

must be considered if we take cET instead of absolute WCET into account for certification. The table clearly shows that the reduction in WCET is highly application dependent.

For short applications (`jane, fac, fibcall, duff, crc`), the reduction in WCET is negligible. These applications are short and does not have many cache misses. As explained in Sec. 4, each cache miss is represented by one level in the tree (Fig. 5). Weights of execution times decrease level by level. Due to the less number of levels, none (`fibcall`) or only few (`crc`) execution time weights are decreased far below 10^{-8} in order to be termed as "negligible". This enforces consideration of almost all possible execution times to calculate cET. Hence, negligible reduction in the absolute WCET is achieved for short applications.

Some of the applications (`recursion, expint, edn, matmul, fir`) are long, however, their execution time variability (equation (6)) is less. Execution times of these applications are dominated by computation instead of memory access latencies. Although these applications have many cache misses, considering applications' total execution time the numbers of cache misses are small. Hence, variable latencies of cache misses cannot influence the final execution time significantly.

Fig. 13 depicts relation between variability in execution time, reduction in the WCET and the $\overline{DL}$ factor. Apart from short applications, reduction in WCET follows pattern of variability. However, the reduction in the WCET is not exactly proportional to the variability in execution time. For example, although `ud, minver, quart` and `ludcmp` have higher variability than the `fdct`, the `fdct` application has the highest reduction in WCET. This high reduction in WCET is attributed to the low $\overline{DL}$ factor of the `fdct` application. In other words, the `fdct` application does majority of accesses in the favorable region of Fig. 9, hence, it is less vulnerable to high latencies.

6.3 Experiment II

In this experiment, we investigate the relationship between the reduction in WCET with the variability of the execution time and $\overline{DL}$ factor. We hypothetically added another four cores in our test architecture to increase the $[B_L, W_L]$ range and analyzed the weighted execution time of `matmul` application from the Mälardalen benchmark suit. The `matmul` application is chosen due to

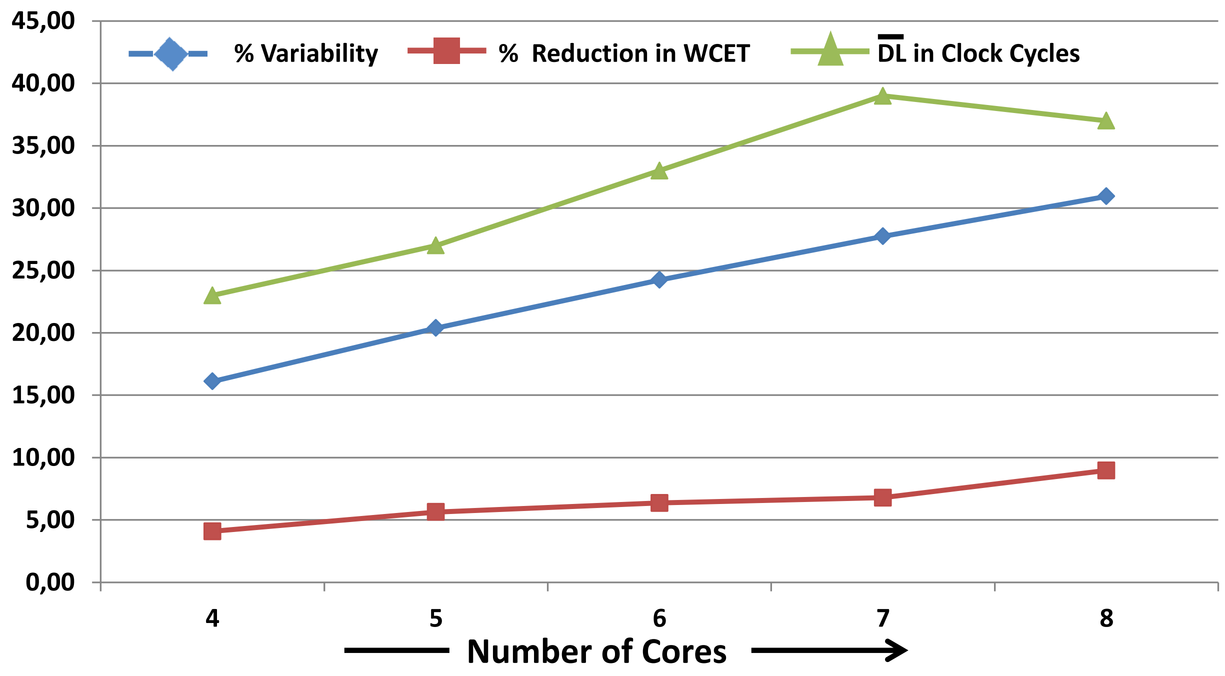

Figure 14: Variability vs Reduction in WCET for `matmul` Application

its reasonable length and low execution time variability. Fig. 14 shows that we could achieve higher reduction in WCET with the increase in execution time variability. Note that $\overline{DL}$ is the highest for $\alpha = 6$. Hence, under α interference, the `matmul` application produces larger execution time if 6 out of 7 co-existing masters are turned on than all 7 co-existing masters are turned on.

7 Discussion

Our approach derives cET using factors such as `SlotSize`, Number of masters, c_i, t_i, number of cache miss, $\overline{DL}$ factor and cut off. Note that L_i was measured during application execution, however, it was removed to build computation trace. `SlotSize` and Number of masters are architecture dependent and known. c_i, t_i and number of cache miss are application dependent. From these parameters $\overline{DL}$ factor is derived. The cut off depends on the severity of consequences if deadline is missed. Thus, none of the parameters used to derive cET depends on the co-existing applications. This shows that we did the cET analysis of applications executing on COTS architectures in isolation.

In future, we would extend our analysis to support *refresh* operation of SDRAM to be able to use an SDRAM as a main memory in the test architecture. Apart from the *refresh* operation, as informed earlier, *jitter* in occurrence of cache misses in the computation trace is left for future work. We did not consider shared L2 caches for our analysis since we believe that they tremendously increase execution time pessimism when applications are analyzed in isolation. Consider that a highly active co-existing core always fills up the shared L2 cache with its data. In this case, each L1 miss of the application-under-test also produces an L2 miss. Hence, each L1 miss must be considered as an L2 miss when analyzed in isolation. Our belief is supported by the latest report on using multi-cores in airborne systems [29].

8 Conclusion

This paper has proposed a novel approach of weighted execution time analysis of applications on off-the-shelf multi-core architectures with shared memory. Instead of providing an absolute worst case execution time, the execution time distribution and the weight of each execution time value are derived. The derivation process is independent of activities of co-existing cores making it truly analysis in isolation.

The approach is tested on the real world applications from the Mälardalen benchmark suit on a multi-core architecture. Depending on the required precision, a specific execution time from the derived profile is chosen as a "likely WCET". The chosen execution time is up to 14% less than the absolute WCET.

References

[1] SOPC Builder Design Optimizations. www.altera.com/literature.

[2] Multi-layer AHB Technical Overview. www.infocenter.arm.com/.

[3] LEON4 32-bi Processor Core. www.gaisler.com/.

[4] G. Gebhard. Timing Anomalies Reloaded. In *WCET 2010*, Dagstuhl, Germany.

[5] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. 28(7), 2009.

[6] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-aware multicore wcet analysis through tdma offset bounds. In *Real-Time Systems (ECRTS)*, 2011.

[7] Chattopadhyay, S. and Kee, C.L. and Roychoudhury, A. and Kelter, T. and Marwedel, P. and Falk, H. A Unified WCET Analysis Framework for Multi-core Platforms. In *RTAS*, 2012.

[8] Ding, H., Liang, Y., and Mitra, T. Shared Cache Aware Task Mapping for WCRT Minimization. In *ASP-DAC*, 2013.

[9] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner. Using measurements as a complement to static worst-case execution time analysis. In Intelligent Systems at the Service of Mankind, vol. 2, UBooks Verlag, Dec. 2005.

[10] 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 *Embedded Software (EMSOFT)*, pages 175–184, Tampere, Finland, 2012. ACM.

[11] 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.*, 8(4):34:1–34:25, Jan. 2012.

[12] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 132 –143, may 2012.

[13] H. Shah, A. Knoll and B. Akesson. Bounding Resource Interference: Detailed Analysis vs. Latency-Rate Analysis. In *Proc. DATE*, 2013.

[14] H. Shah, A. Raabe and A. Knoll. Bounding WCET of Applications Using SDRAM with Priority Based Budget Scheduling in MPSoCs. In *Proc. DATE*, 2012.

[15] T. Ungerer, *et al.* Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30:66–75, September 2010.

[16] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14:2:1–2:24, January 2009.

[17] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. CASES '08, NY, USA.

[18] Edgar, S. and Burns, A. Statistical Analysis of WCET for Scheduling. RTSS 2001.

[19] Bernat, G. and Colin, A. and Petters, S.M. WCET Analysis of Probabilistic Hard Real-Time Systems RTSS 2002.

[20] Cazorla, F. J., E. QuiËIJnones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston. PROARTIS: Probabilistically Analysable Real-Time Systems. ACM Transactions on Embedded Computing Systems, 2012.

[21] QuiÃśones, E., E. Berger, G. Bernat, and F. J. Cazorla. Using Randomized Caches in Probabilistic Real-Time Systems. ECRTS 2009.

[22] Kosmidis, L., J. Abella, E. Quinones, and F. Cazorla. A Cache Design for Probabilistic Real-Time Systems. DATE 2013.

[23] http://rapitasystems.com/

[24] M. Steine, M. Bekooij, and M. Wiggers. A priority-based budget scheduler with conservative dataflow model. In *Proc. DSD*, 2009.

[25] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. In *Proc. RTCSA*, 2008.

[26] Pellizzoni, R. and Schranzhofer, A. and Jian-Jia Chen and Caccamo, M. and Thiele, L. Worst case delay analysis for memory interference in multicore systems. In *Proc. DATE*, 2010.

[27] Mingsong Lv and Wang Yi and Nan Guan and Ge Yu Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software. In *Proc. RTSS*, 2010.

[28] Kirner, R. and Puschner, P. Time-predictable computing. In *Springer-Verlag*, 2010.

[29] MULCORS - Use of Multicore Processors in airborne systems. Research Project Report EASA.2011/6. www.easa.europa.eu

A Interference Scenarios

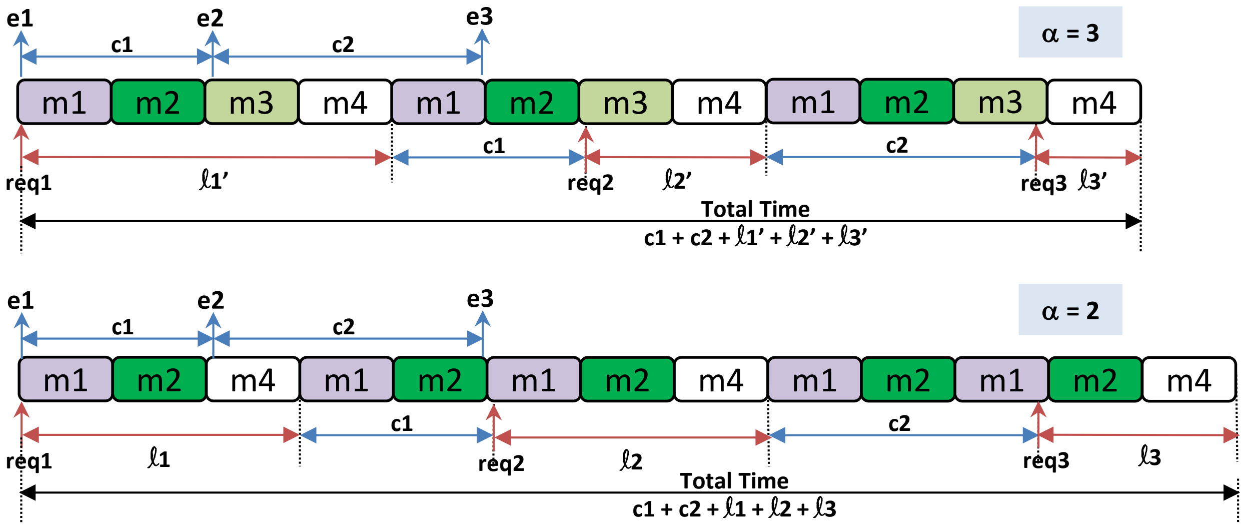
e1
e2
e3
c1
c2
α = 3
m1 m2 m3 m4 m1 m2 m3 m4 m1 m2 m3 m4
req1
ℓ1'
req2
ℓ2'
req3
ℓ3'
Total Time
c1 + c2 + ℓ1' + ℓ2' + ℓ3'
α = 2
m1 m2 m4 m1 m2 m1 m2 m4 m1 m2 m1 m2 m4
ℓ1
ℓ2
ℓ3
c1 + c2 + ℓ1 + ℓ2 + ℓ3

Figure 15: Total Execution Time under $\alpha = 3$ and $\alpha = 2$

Fig. 15 depicts particular application execution under uninterrupted (α) interference. Here, two cases are depicted, i) Uninterrupted interference is produced by 3 cores ii) Uninterrupted interference is produced by 2 cores. Note that the latencies experienced by the application executing on `m4` is more in case the interference is produced by 2 cores than the interference is produced by 3 cores. Hence, for this particular application, measurement of execution time under $\alpha = 3$ interference and considering that as a WCET is an optimistic estimation.

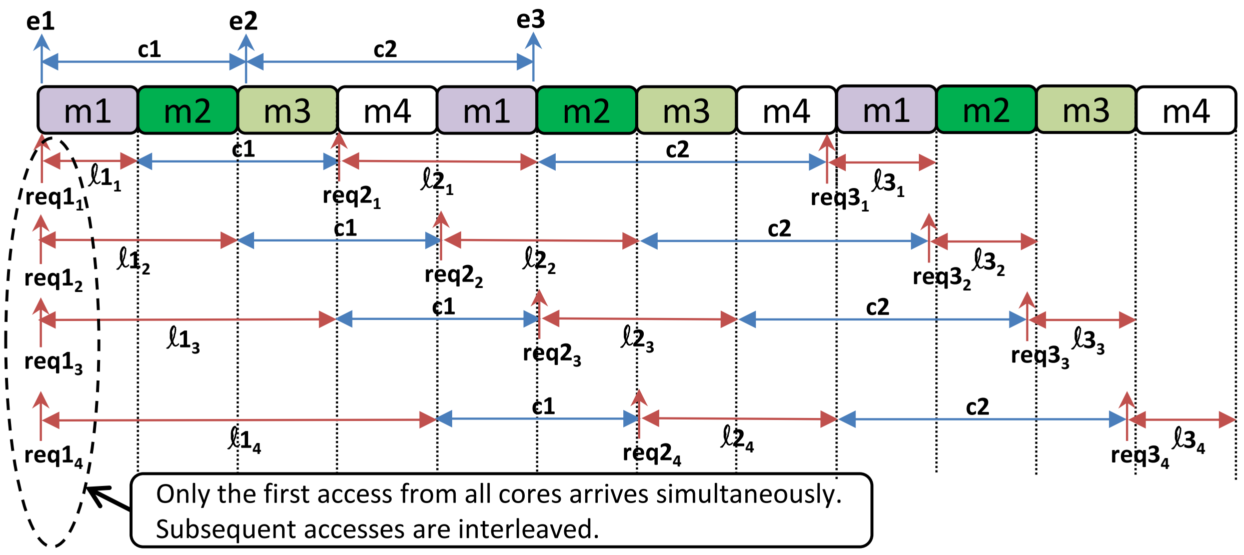
e1
e2
e3
c1
c2
m1 m2 m3 m4 m1 m2 m3 m4 m1 m2 m3 m4
req1₁ ℓ1₁ req2₁ ℓ2₁ req3₁ ℓ3₁
req1₂ ℓ1₂ req2₂ ℓ2₂ req3₂ ℓ3₂
req1₃ ℓ1₃ req2₃ ℓ2₃ req3₃ ℓ3₃
req1₄ ℓ1₄ req2₄ ℓ2₄ req3₄ ℓ3₄
Only the first access from all cores arrives simultaneously.
Subsequent accesses are interleaved.

Figure 16: Executing the same application on all cores.

Fig. 16 depicts a scenario when all cores execute precisely the same application. The subscripts in the parameters denotes the associated masters. It is clear that the apart from the first access of `m4` ($l1_4 = 4 \times SlotSize$), none of the accesses experiences the worst case latency. Here, only the first access from all masters arrive concurrently. All subsequent accesses are interleaved due to the round robin arbitration.

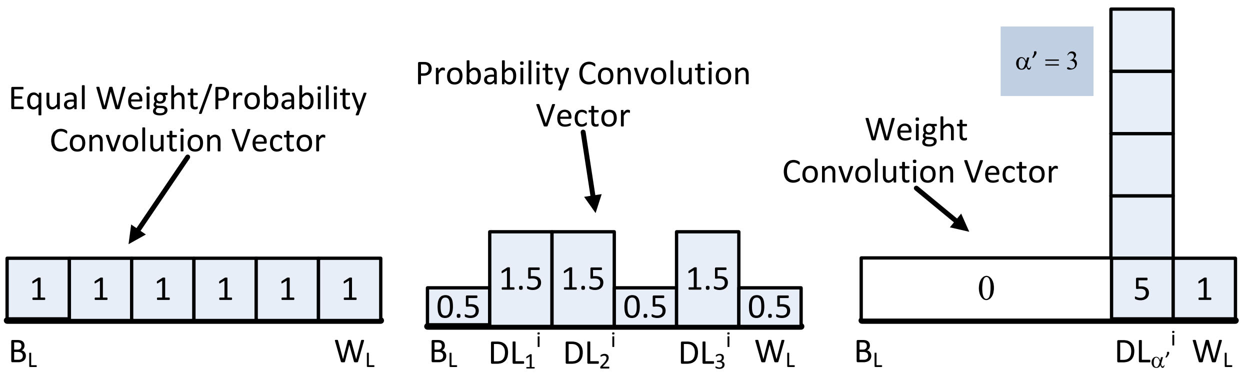

Figure 17: Difference between Probability and Weight

B Difference between Weight and Probability

There is a subtle difference between assigning probabilities to latencies and assigning conservative weights to latencies. Fig. 17 depicts the difference in the convolution vectors. Under α interference, for i^{th} access, all DL_α^i latencies should be assigned higher probability since all these latencies are "easy-to-induce". Moreover, these higher probability is assigned at the cost of probabilities of all other latencies. As depicted in the figure, this approach increases probability density towards B_L end of the range. Hence, if the "application-under-test" executes under $\alpha' = 3$ interference, the increased probability density towards B_L produces optimistic cET.

In our approach, in order to stay conservative, we deliberately increase weight of $DL_{\alpha'}^i$ latency such that weight density could increase only towards W_L end. Note that in our approach weight of W_L never decreases.