Bounding SDRAM Interference: Detailed Analysis vs. Latency-Rate Analysis

Hardik Shah[1], Alois Knoll[2], and Benny Akesson[3]
[1]fortiss GmbH, Germany, [2]Technical University Munich, Germany, [3]CISTER-ISEP Research Centre, Portugal

***Abstract*—The transition towards multi-processor systems with shared resources is challenging for real-time systems, since resource interference between concurrent applications must be bounded using timing analysis. There are two common approaches to this problem: 1) Detailed analysis that models the particular resource and arbiter cycle-accurately to achieve tight bounds. 2) Using temporal abstractions, such as latency-rate ($\mathcal{LR}$) servers, to enable unified analysis for different resources and arbiters using well-known timing analysis frameworks. However, the use of abstraction typically implies reducing the tightness of analysis that may result in over-dimensioned systems, although this pessimism has not been properly investigated.**

This paper compares the two approaches in terms of worst-case execution time (WCET) of applications sharing an SDRAM memory under Credit-Controlled Static-Priority (CCSP) arbitration. The three main contributions are: 1) A detailed interference analysis of the SDRAM memory and CCSP arbiter. 2) Based on the detailed analysis, two optimizations are proposed to the $\mathcal{LR}$ analysis that increase the tightness of its interference bounds. 3) An experimental comparison of the two approaches that quantifies their impact on the WCET of applications from the CHStone benchmark.

I. Introduction

The challenge of designing embedded real-time systems is increasing, as stringent power and performance requirements cause a transition to complex multi-processor systems with shared resources. One important problem is bounding interference between applications in shared resources using timing analysis, which is essential to determine the worst-case execution time (WCET) of real-time applications. Common approaches to this problem can be classified in two categories: 1) Detailed analysis that models resources and arbiters in a cycle-accurate or cycle-approximate manner [1], [2] to achieve tight bounds on interference. 2) Analysis based on abstractions that capture the timing behavior of the shared resource at a higher level [3]–[5] to enable unified analysis for different resources and arbiters using well-known timing analysis frameworks, such as network calculus or data-flow analysis. The use of abstraction typically implies pessimistic analysis, increasing the difficulty of satisfying deadlines and potentially resulting in over-dimensioned systems. However, this pessimism has not been properly investigated and quantified, which makes it difficult to determine if and when the use of abstraction is appropriate.

This paper addresses this problem by comparing detailed analysis and the latency-rate ($\mathcal{LR}$) server abstraction [6] in terms of WCET of concurrently executing applications for the case of an SDRAM shared under Credit-Controlled Static-Priority (CCSP) arbitration [7]. The three main contributions of this paper are: 1) A detailed analysis of shared SDRAM interference under CCSP arbitration. 2) Two optimizations to $\mathcal{LR}$ analysis derived from the detailed analysis to tighten the interference bounds. 3) The cost of using the $\mathcal{LR}$ abstraction is quantified through experimental evaluation of the two approaches and comparison of their impact on WCET of applications from the CHStone benchmark [8]. Conservativeness of both analyses is also demonstrated by comparing results to the observed execution time on an FPGA board.

The rest of the paper is organized as follows. Sec. II discusses related work. Sec. III provides necessary background before a detailed interference analysis of a shared SDRAM resource under CCSP arbitration is presented in Sec. IV. Two optimizations are then proposed to the $\mathcal{LR}$ analysis in Sec. V, based on observations from the detailed analysis. The two approaches are experimentally evaluated in Sec. VI before conclusions are drawn in Sec. VII.

II. Related Work

There are several works on shared resource interference analysis using detailed analysis and $\mathcal{LR}$ analysis. This section discusses related work in the two categories separately.

Detailed interference analysis uses cycle-accurate or cycle-approximate models of a particular resource and arbiter. The number of works in this area is large, since each model is customized to a specific resource and arbiter combination to achieve tight bounds. We hence discuss work related to shared memory interference analysis only. The works in [9], [10] use models of concurrent applications for detailed interference analysis on shared memories for WCET estimation. In contrast, [1], [2] perform independent analyses of tasks sharing memories, assuming worst-case interference under round-robin arbitration and Priority Based Budget Scheduler [11] (PBS), respectively. This makes the analyses less tight than [9], [10], but they are independent of the concurrent applications.

The $\mathcal{LR}$ server abstraction [6] is a simple linear lower bound on the service provided by a resource. The model was originally developed for analysis of networks, but has gained popularity in the context of real-time embedded systems in recent years. Example uses of the model involve modeling buses [3], networks-on-chips [4], and SRAM and SDRAM memories [5]. The abstraction has two key benefits: 1) It enables resource interference to be bounded for the many arbiters belonging to the class [6], such as Weighted Round-Robin, Time-Division Multiplexing, PBS, and several varieties of Fair Queuing, thereby addressing the diversity of arbitration in complex systems in

978-3-9815370-0-0/DATE13/©2013 EDAA

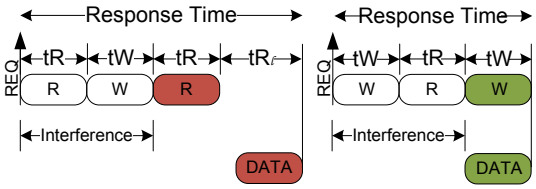


Fig. 1: Response times: (a) read request (b) write request.

a unified manner. 2) It supports formal performance analysis using approaches based on well-known frameworks, such as network calculus or data-flow analysis.

Despite the many works on detailed analysis and $\mathcal{LR}$ analysis, the two approaches have not been compared in terms of the resulting WCET of real-time applications. This paper addresses this issue by comparing the two analyses for a shared SDRAM under CCSP arbitration and identifying and quantifying their sources of pessimism.

III. Background

This section provides a high-level overview of the considered system, the $\mathcal{LR}$ abstraction, and the CCSP arbiter, to facilitate the discussions in later sections.

A. System Model

We consider multi-processor systems with shared SDRAMs that are free from timing anomalies [12]. The memory controller can be any real-time controller with a close-page policy, such as [1], [2], [5]. The benefit of this policy is that it reduces dependencies between requests and maximizes the guaranteed bandwidth. However, a small dependency still exists between consecutive requests, as the SDRAM data path is bi-directional and requires a few cycles to switch from read to write and vice versa. For conservative analysis, we must hence consider that the interference on the shared SDRAM is produced by alternating read/write requests. This is illustrated for both read and write requests in Fig. 1. Fig. 1(a) shows that a read request suffers interference from a sequence of requests from other masters. Once a read request is scheduled, it requires maximum tR clock cycles to complete its execution in the memory controller and another tR_l clock cycles to receive all data from the memory. Since the data of write requests is delivered along with the request, it just requires a maximum of tW cycles to finish, as indicated in Fig. 1(b). The parameters tR_l, tR, and tW depend on the type of SDRAM and its frequency.

SDRAMs store data as a charge on its internal array of capacitors. These capacitors must be periodically refreshed every *tREFI* cycles for a duration of *tRFC* cycles for data retention. During this time, SDRAM cannot be used, stalling any processor that accesses the SDRAM.

We assume a trace of SDRAM requests representing the application under analysis is available, where the i^{th} memory request is represented as $(\tau_i, RequestType_i)$. Where τ_i is processing time before the request is issued and $RequestType_i$ is the type of the request (read/write).

B. Latency-Rate Servers

We now introduce the concept of latency-rate ($\mathcal{LR}$) [6] servers as a shared-resource abstraction and its applicability to SDRAMs. In essence, a $\mathcal{LR}$ server guarantees a

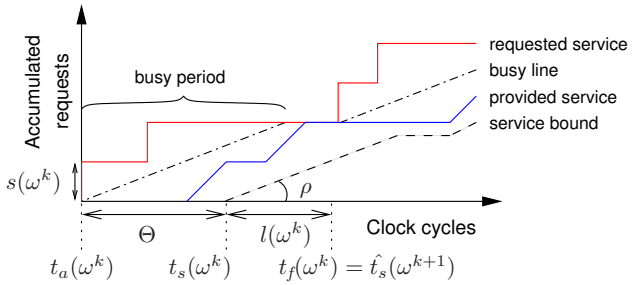


Fig. 2: A $\mathcal{LR}$ server and its associated concepts.

master a minimum allocated rate (bandwidth), ρ, after a maximum service latency (interference), Θ, as shown in Fig. 2. This linear service guarantee bounds the amount of data that can be transferred during any interval independently of the behavior of other masters. The values of Θ and ρ depend on the arbiter and its configuration.

The $\mathcal{LR}$ service guarantee is conditional and only applies if the master produces enough requests to keep the server busy. This is captured by the concept of *busy periods*, which intuitively are periods in which a master requests at least as much service as it has been allocated (ρ) on average. This is illustrated in Fig. 2, where the master is busy when the requested service curve is above the dash-dotted reference line with slope ρ that we informally refer to as the *busy line*. The figure also shows how the service bound is shifted when the master is not busy.

We proceed by showing how scheduling times and finishing times of requests are bounded using the $\mathcal{LR}$ server guarantee. From [13], the worst-case scheduling time, t_s, of the k^{th} request from a master, m, is expressed according to the first term (the max-expression) in Equation (1), where $t_a(\omega^k)$ is the arrival time of the request and $t_f(\omega^{k-1})$ is the worst-case finishing time of the previous request from master m. The worst-case finishing time is then bounded by adding the time it takes to finish a scheduled request of size $s(\omega^k)$ at the allocated rate, ρ, of the master, which is called the *completion latency* and is defined as $l(\omega^k) = s(\omega^k)/\rho$. This is expressed in Equation (1) and is visualized for request ω^k in Fig. 2.

$$t_f(\omega^k) = \max(t_a(\omega^k) + \Theta, t_f(\omega^{k-1})) + s(\omega^k)/\rho \quad (1)$$

Both Θ and $l(\omega^k)$ are expressed in abstract *service cycles*, where each service cycle correspond to the execution time of a request on the particular resource. To be useful in a concrete timing analysis, the service cycles must be converted to clock cycles. Here, we follow the approach in [14] that considers SDRAM memories with variable execution times of requests. Intuitively, this means always considering an interfering refresh and then alternating between interfering write (tW) and read requests (tR), as previously mentioned in Sec. III-A. An extra interfering request is also added to the service latency to consider blocking in the case where a request arrives just one clock cycle after a scheduling decision has been made. For the completion latency, the average time of a read and a write, plus a small penalty of *tRFC* cycles to compensate for a refresh once every *tREFI* cycles, are considered. This 'refresh tax' ensures that refresh is correctly accounted for if a master has a busy period longer than *tREFI* [14].

C. CCSP Arbitration

This paper uses a CCSP arbiter [7], which comprises a rate regulator and a static-priority scheduler. The regulator provides *accounting* and *enforcement* and determines which requests are *eligible* for scheduling at a particular time, considering the allocated service of the masters. The service allocated to a master m consists of two parameters: allocated burstiness (σ_m) and allocated rate (ρ_m). For a valid allocation, two conditions must hold: 1) $\sigma_m \geq 1$, for the interference bound of the arbiter to be valid, and 2) $\sum_{\forall m} \rho_m \leq 1$, to prevent over-loading the resource.

The two allocation parameters are used by the CCSP accounting mechanism, shown in Equation (2), to compute the number of credits, $c_m(t)$, of a master m at time t. The equation is evaluated after each scheduling decision, so credits are updated per service cycle as opposed to per clock cycle. The intuition behind the mechanism is that a master starts with σ_m credits. The credits are incremented by ρ_m at every scheduling decision and one credit is removed when the master is scheduled ($\gamma(t) = m$). Lastly, to prevent masters from accumulating credits during long periods of idleness and later starve lower-priority masters, credits are saturated at the initial value, σ_m, if the master is not scheduled and does not have any backlogged requests ($b_m = 0$). It has been shown in [7] that the service latency in number of service cycles, Θ_m, of a master m under CCSP arbitration is computed according to Equation (3), where M_m^+ is the set of higher priority masters.

$$c_m(0) = \sigma_m$$

$$c_m(t+1) = \begin{cases} c_m(t) + \rho_m - 1 & \gamma(t) = m \\ c_m(t) + \rho_m & \gamma(t) \neq m \wedge b_m > 0 \\ \min(c_m(t) + \rho_m, \sigma_m) & \gamma(t) \neq m \wedge b_m = 0 \end{cases} \quad (2)$$

$$\Theta_m = \frac{\sum_{\forall m_j \in M_m^+} \sigma_{m_j}}{1 - \sum_{\forall m_j \in M_m^+} \rho_{m_j}} \quad (3)$$

IV. Detailed Analysis

This section explains the detailed analysis method for bounding the WCET of an application sharing an SDRAM under CCSP arbitration. Here, N denotes total number of masters, m the master under investigation, and rw the type (read/write) of interfering requests. Priorities are assigned in ascending order, masters $N \rightarrow (m+1)$ have higher and $(m-1) \rightarrow 1$ have lower priority than m.

Equation (4) defines the replenishment period, P_x, of master x, which is the number of clock cycles ($\approx$ nearest integer) required to accumulate one credit in the arbiter. The equation considers that $1/\rho_x$ scheduling decisions are required by the arbiter to replenish a credit given the allocated rate of the master and the average time it takes to serve a request in the worst-case sequence of alternating reads and writes. The variable η_x is used to store the clock cycle when master x receives its next credit.

$$P_x = \left\lfloor \frac{1}{\rho_x} \times \frac{(tR + tW)}{2} \right\rceil, \qquad \forall x \in [N, 1] \quad (4)$$

We now present four intuitions about the worst-case response time (WCRT) of a memory request: I) Interfering requests and the request from m itself form an alternating sequence of read and write, which maximizes SDRAM latencies (Sec. III-A). II) All masters use their credits *only* to interfere with m. When m is not backlogged or not eligible ($c_m < 1$), other masters do not request and accumulate as many credits as possible. As soon as m becomes eligible and backlogged, all higher priority masters ($N \rightarrow (m+1)$) request simultaneously, maximally delaying scheduling of the request from m. III) One of the lower priority masters ($(m-1) \rightarrow 1$) requests one clock

Algorithm 1 Detailed Analysis

```
1: TopLevel()
2: tRef ← tREFI                                          # initial refresh
3: c_[N→1] ← σ_[N→1]                                     # initial credits
4: for request i = 1 → TotalAccesses do
5:     LatRead ← WCRT(t, m, RequestType_i, read)
6:     LatWrite ← WCRT(t, m, RequestType_i, write)
7:     Lat ← max(LatRead, LatWrite)
8:     tRef ← tRef + Lat + τ_i
9:     if tRef ≥ tREFI then
10:        Lat ← Lat + tRFC
11:        tRef ← (tRef − tREFI) + tRFC
12:        η_[1→N] ← η_[1→N] + tRFC
13:    end if
14:    t ← t + Lat + τ_i
15: end for

16: WCRT(Time t, Master m, RequestType at, Interference rw)
17: T ← t
18: while c_m = 0 OR T < t do
19:    T ← η_m
20:    UpdateCredit(T, [1 → N], s = 1)
21: end while
22: if m ≠ 1 then
23:    T ← T + Delay(rw)
24:    rw ← ¬rw
25:    UpdateCredit(T, [N → m], s = 0)
26: end if
27: HighPrioCredit ← HighPrioCredit + c_[N→m+1]
28: while HighPrioCredit > 0 do
29:    for master x = N → m + 1 do
30:       for credit j = c_x → 0 do
31:          c_x ← c_x − 1
32:          HighPrioCredit ← HighPrioCredit − 1
33:          T ← T + Delay(rw)
34:          rw ← ¬rw
35:          UpdateCredit(T, [(x − 1) → m], s = 0)
36:       end for
37:    end for
38:    UpdateCredit(T, [N → m + 1], s = 1)
39:    HighPrioCredit ← HighPrioCredit + c_[N→m+1]
40: end while
41: T ← T + Delay(at)
42: if at = read then
43:    T ← T + tR_l
44: end if
45: c_m ← c_m − 1
46: lat = (T − t) ↩

47: UpdateCredit(Time T, Master x, Saturate s )
48: if (c_x ≥ σ_x) and (s = 1) then
49:    η_x ← T + P_x                                     # in saturation
50: else
51:    if T ≥ η_x then
52:       add ← 1 + (T − η_x) / P_x
53:       c_x ← c_x + add
54:       rem ← (T − η_x) mod P_x
55:       η_x ← T + P_x − rem
56:    end if
57:    if (c_x > σ_x) and (s = 1) then
58:       c_x ← σ_x                                      # in saturation
59:    end if
60: end if↩
```

cycle before m becomes eligible and backlogged. Hence, this lower priority master blocks all higher priority masters and m. IV) One refresh interferes at every $tREFI$.

Algorithm 1 estimates the WCET of the provided trace based on above mentioned intuitions. The algorithm has three functions. 1) Lines 1-15: *TopLevel* takes care of refresh interference and initial credits (σ) of all masters. 2) Lines 16-46: *WCRT* calculates the worst-case response time of a single request. 3) Lines 47-60: *UpdateCredit* tracks credits of all masters and calculates their η variables according to Equations (2) and (4).

The *TopLevel* function initializes a refresh counter ($tRef$) to $tREFI$ to consider an interfering refresh that may occur together with the first request. It then calls the *WCRT* function for all requests in the provided trace one by one. At this stage, the length of the alternating interfering sequence (Intuition I) for any request is unknown (it depends on the current credits of other masters). Hence, for every request, the *WCRT* function is called twice considering an alternating sequence starting with a read request and a write request, respectively. The maximum of them is the WCRT of that request, which is then added to the refresh counter together with processing time (τ_i) (Line 8). As soon as *tRef* exceeds $tREFI$, refresh interference $tRFC$, is added to the latency (Line 10, Intuition IV) and η as well, since masters are not replenished during refresh. At the end, the variable t contains the WCET of the given trace.

The *WCRT* function, at first, checks if m has at least one credit to be eligible for scheduling. If not, then time is forwarded to η_m when m receives its next credit (Lines 17-21). Moreover, we assume that during this forwarded time, $T - t$, other masters do not request and accumulate as many credits as possible (Intuition II). However, their credits are saturated at maximum (σ) since they are assumed not to be backlogged (Equation (2)). We call the *UpdateCredit* function with $s = 1$ if the master is assumed to be "not backlogged" and with $s = 0$ if the master is assumed to be "blocked" by other masters. According to Intuition III, after being eligible, at Line 22, m must assume interference from one lower priority master (provided that m itself is not the lowest priority master, $m \neq 1$). *Delay()* simply returns the value of tR or tW depending on if the request is a read or a write. For each interfering request, we assume the next interfering request to be of opposite type ($rw \leftarrow \neg rw$) to form an alternating sequence of read/write requests (Intuition I). We also assume that all high-priority masters request simultaneously when m becomes eligible and backlogged (Intuition II). Thus, the low-priority request (Line 22) blocks all high-priority masters and m. Hence, their credits are replenished in Line 25 using the *UpdateCredit* method, which allows credits to exceed the σ boundary ($s = 0$).

Lines 27-40 consider interference from all higher priority masters. After each high-priority request, the credits of the scheduled master x is reduced by one (Line 31) and the credits of all "blocked" masters, $(x - 1) \rightarrow m$, are replenished (Line 35). It may be possible that during these high-priority requests, some of the higher priority masters become eligible again and interfere once more. Hence, credits of all higher priority masters are re-calculated (Lines 38-39) and their interference is iteratively considered.

After considering interference from one lower priority master and all higher priority masters, the request from m itself is scheduled (Line 41). For a read request, an additional latency of tR_l (Fig. 1) is added on Line 43.

V. Latency-Rate Optimizations

The $\mathcal{LR}$ server abstraction assumes that service is provided in a fluid and continuous manner, which is not the case for shared resources in multi-processor systems. This section learns from the detailed analysis in Sec. IV and proposes two improvements to the $\mathcal{LR}$ server analysis to tighten the WCRT of requests by capitalizing on the fact that service is provided in discrete chunks that are served in a non-preemptive manner. In the case of our shared SDRAM, this corresponds to that only complete memory requests (e.g. cache lines) are scheduled and are then served non-preemptively until completion.

A. Reduced CCSP Service Latency Bound

The first improvement is specific to the CCSP arbiter and considers an improved bound on the service latency (interference), Θ. The bound in Equation (3) does not consider that a master must have one full credit to pay for a complete resource access to be scheduled. Instead, it lets higher priority masters sum up their fractional credits as well, over-estimating the worst-case interference.

This can be addressed by using the same interference bound used in the detailed analysis in lines 28-40 of Algorithm 1. This bound can be captured by an iterative equation, as shown in Equation (5). A conservative bound for a master m is computed by iteratively evaluating the equation over the variable k, Θ_m^0, Θ_m^1, Θ_m^2, etc., until the equation converges and two consecutive iterations produce the same result. This is guaranteed to happen eventually, since the resource may not be overloaded, i.e. $\sum_{\forall m_j \in M_m^+} \rho_{m_j} < 1$.

$$\begin{aligned} \Theta_m^0 &= 0,\ \Theta_m^1 = \sum_{\forall m_j \in M_m^+} \sigma_{m_j} \\ \Theta_m^k &= \Theta_m^{k-1} + \sum_{\forall m_j \in M_m^+} \left\lfloor (\Theta_m^{k-1} - \Theta_m^{k-2}) \times \rho_{m_j} \right\rfloor \end{aligned} \tag{5}$$

The intuition behind the bound is that all higher priority masters start with σ_{m_j} credits, which is the worst case [7]. The iteration then considers the interference from these credits, and computes the number of fully replenished credits of each master during this time. The interference resulting from these replenished credits are then considered in the following iteration until the equation converges.

B. Non-Preemptive Service

The second optimization applies to all resources where requests are served in a non-preemptive manner and is independent of which $\mathcal{LR}$ arbiter is being used. The key insight is that a scheduled request in a non-preemptive resource is *temporarily* served at the full capacity of the resource $\rho' = 1$ until it finishes and not at the allocated rate, ρ. It is hence guaranteed to be finished after one

service cycle, as opposed to after the completion latency of $1/\rho$ service cycles, suggested by the $\mathcal{LR}$ abstraction. This is captured by the detailed analysis, which adds a single read (tR) or write (tW) request on behalf of the considered master after waiting for the worst-case interference (Line 41 in Algorithm 1).

The irregular guarantee provided to a busy master by a non-preemptive server and the difference compared to the original $\mathcal{LR}$ bound is shown in Fig. 3. However, the irregular bound is not a $\mathcal{LR}$ guarantee, as it cannot be represented using only the two parameters Θ and ρ. It is tempting to capture this by using the maximum rate of the server, $\rho' = 1$. However, this would make the server believe that the master is *always* served at the full rate of the resource through the dependency on the previous finishing time in Equation (1), resulting in non-conservative bounds. Similarly to [13], we instead reduce the service latency to $\Theta' = \Theta - (1/\rho - 1)$ service cycles, as shown in Fig. 3. An implication of this is that the intuition behind the scheduling time is lost, as requests are no longer guaranteed to be scheduled at t_s, although the scheduling time can still be computed using the original Θ if needed. More importantly, the bound on finishing time is still conservative and tighter than before, as shown in Fig. 3.

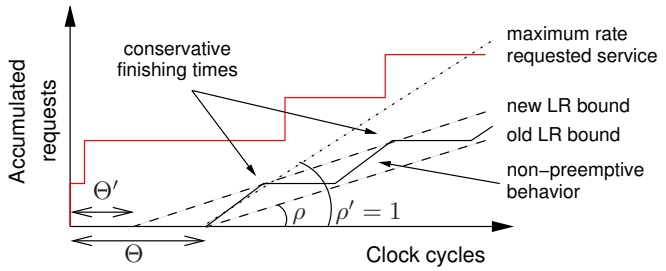


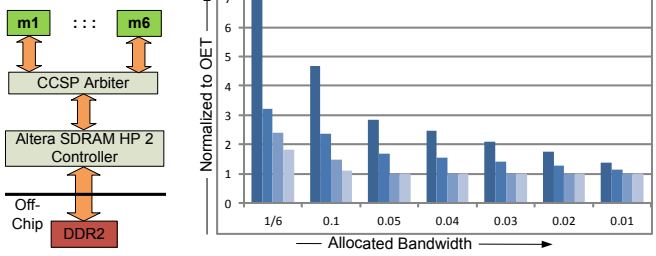
Fig. 3: Improved $\mathcal{LR}$ bound for non-preemptive resources.

VI. Experiments

This section experimentally evaluates the two analyses presented in this paper. First, we explain the experimental setup and then proceed with two experiments. The first experiment quantifies the impact of both the analyses on WCET for two applications. The second experiment illustrates how the obtained results depend on the arbiter configuration by varying the allocated rate.

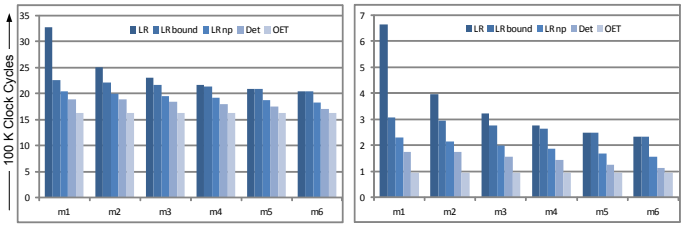
A. Experimental Setup

The experiments consider six masters, implemented as hardware traffic generators to replay memory traces. The masters are connected to a Micron DDR2-667 memory through an SDRAM controller running at 125 MHz on an Altera Cyclone III FPGA, as shown in Fig. 4a. The traces are executed in a blocking manner. A memory request thus stalls the traffic generator and delays all subsequent processing and memory requests by the memory latency.



(a) Setup (b) Results with reduced allocations.

Fig. 4



(a) JPEG (b) Motion Compensation

Fig. 5: Results with equal allocations in 100K Clock Cycles

Techniques from [2] were used to achieve predictable behavior from the Altera SDRAM controller and to extract the worst-case memory parameters in clock cycles for a fixed request size of 32 B. For this combination of memory and memory controller, the worst-case timing parameters are: read access $tR = 12$, read latency $tR_l = 46$, write access $tW = 14$, refresh interval $tREFI = 975$, and refresh time $tRFC = 41$ clock cycles. This setup hence delivers on the system model presented in Sec. III-A.

The SDRAM traces were obtained by executing applications from the CHStone [8] benchmark on the Simplescalar instruction set simulator [15] with 16 KB L1 D-cache, 16 KB L1 I-cache, 128 KB shared L2 cache and 32 B cache line configuration. We filtered out the L2 cache misses to obtain a trace of the access going to SDRAM. The L2 cache is partitioned to avoid effects of concurrent applications on cache behavior. The trace is then fed to all six masters emulating "same application, same path" being executed on six separate processors. We then analyzed the traces to estimate WCET of that execution path on all masters using both detailed analysis and $\mathcal{LR}$ analysis.

In this paper, we study results of JPEG and motion compensation applications, which are the least and most memory intensive applications in the CHStone benchmark, respectively. In Figures 4b and 5, "LR" denotes regular $\mathcal{LR}$ analysis, "LR bound" $\mathcal{LR}$ analysis with improved service latency bound for CCSP, "LR np" non-preemptive $\mathcal{LR}$ analysis, "Det" detailed analysis, and "OET" Observed Execution Time on FPGA. It should be noted that "LR np" applies the non-preemptive optimization on "LR bound" and hence contains both optimizations proposed in Sec. V.

B. Equal Allocations

In the first experiment, we allocate equal rates to all six masters, $\rho = 1/6$. Fig. 5 depicts the results of the JPEG and motion compensation applications. Although both analyses appear conservative with respect to the OET for all masters, results for JPEG are better than motion compensation for both analyses, and the detailed analysis produces better results than the others. The difference between the detailed analysis and the OET is explained by that the actual execution has less than worst-case interference and less read/write switches in the memory.

The regular $\mathcal{LR}$ analysis produces 1.7 and 3.8 times larger WCET than the detailed analysis for the JPEG and motion compensation applications for the lowest priority master, respectively. Having equal allocated rate for all masters in this test, the differences between the two applications are attributed to their different memory intensities. This leads to the obvious conclusion that a tight memory interference analysis is more important for memory intensive applications.

For the highest priority master, the regular $\mathcal{LR}$ analysis produces 1.2 and 2.0 times larger bounds than the detailed analysis. However, the tightness of $\mathcal{LR}$ analysis increases with the proposed optimizations. The discrete bound on service latency is especially helpful for low-priority masters, where it reduces the difference with respect to the detailed analysis for $m1$ from a factor 3.8 to 1.8 for the motion application. In contrast to the discrete bound, considering non-preemption in the $\mathcal{LR}$ analysis benefits masters of all priorities equally, as it results in a constant reduction in interference that is inversely proportional to the allocated rate. With both optimizations applied, the difference with respect to the detailed analysis is reduced to a factor 1.08 and 1.40, respectively, for the two applications. This difference is due to the refresh management of the $\mathcal{LR}$ analysis, where an interfering refresh is considered for every new busy period as it is included in the service latency of all masters. This drawback is extremely pronounced in this experiment, as the blocking nature of the system combined with the particular allocated rate, causes every single request to start a new busy period.

C. Reduced Allocations

The previous experiment revealed the importance of memory intensive accesses for both analysis methods. This experiment further demonstrates this by reducing the allocated rate of the lowest priority master, who suffered the largest pessimism, in the motion compensation application. Results in Fig. 4b show that the WCET of all techniques approach the OET as the allocated rate is reduced. The reason for this is that reducing the allocated rate enlarges the replenishment period (Equation (4)), increasing the WCRT of requests and the WCET of the application. This causes results from all methods to approach the OET, since over-estimating the number of refreshes or read/write switches becomes insignificant. However, the detailed analysis approaches the OET faster and comes very close already at $\rho = 0.1$, while $\mathcal{LR}$ with the proposed optimizations comes close to OET at $\rho = 0.05$. At this point, the reduction in allocated rate enables the master to remain busy for the entire execution of the trace, causing refresh interference to be estimated tightly. In contrast, the default $\mathcal{LR}$ analysis does not come close to the OET even for $\rho = 0.01$, due to its highly pessimistic bound on service latency. From this experiment, we conclude that both analyses produce tighter results for lower allocated rates, but that the detailed analysis approaches the OET faster than $\mathcal{LR}$ analysis. The allocated rate hence presents a trade-off between high bounds and tight bounds. Furthermore, the tightness of the WCET estimated by $\mathcal{LR}$ analysis depends on the ability of the master to stay busy, as this results in more accurate refresh interference analysis.

VII. Conclusion

This paper compares a detailed analysis and latency-rate ($\mathcal{LR}$) analysis of resource interference to assess the impact of using temporal abstractions in timing analysis of real-time embedded systems. A detailed analysis is presented for an SDRAM shared under Credit-Controlled Static-Priority (CCSP) arbitration. Based on the detailed analysis, two improvements are proposed for the existing $\mathcal{LR}$ analysis to reduce the estimated resource interference: 1) a tighter latency bound for the CCSP arbiter and 2) a latency optimization for non-preemptive resources.

The two analysis approaches are experimentally compared to each other and to an observed execution time on FPGA to quantify their impact on worst-case execution times (WCET) of applications from the CHStone benchmark. The experiments reveal that both methods are conservative with respect to the observed execution and that the WCET estimated by $\mathcal{LR}$ analysis is up to 3.8 times larger than that of detailed analysis. However, the proposed optimizations reduce this factor to 1.4. As the allocated rate is reduced, both analyses produce increasingly tight results and the gap between them disappears.

Acknowledgment

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by the EU ARTEMIS JU funding, within the RECOMP project, ref. ARTEMIS/0202/2009, JU grant nr. 100202. Also, partially funded by German BMBF projects ECU (13N11936) and Car2X (13N11933).

References

[1] M. Paolieri *et al.*, "An Analyzable Memory Controller for Hard Real-Time CMPs," *Embedded Systems Letters, IEEE*, 2009.
[2] H. Shah *et al.*, "Bounding WCET of Applications Using SDRAM with Priority Based Budget Scheduling in MPSoCs," in *Proc. DATE*, 2012.
[3] J. Vink *et al.*, "Performance analysis of SoC architectures based on latency-rate servers," *Proc. DATE*, 2008.
[4] A. Hansson *et al.*, "Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis," *IET CDT*, 2009.
[5] B. Akesson and K. Goossens, "Architectures and modeling of predictable memory controllers for improved system integration," in *Proc. DATE*, 2011.
[6] D. Stiliadis and A. Varma, "Latency-rate servers: a general model for analysis of traffic scheduling algorithms," *IEEE/ACM Trans. Netw.*, 1998.
[7] B. Akesson *et al.*, "Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration," in *Proc. RTCSA*, 2008.
[8] "Chstone, a suite of benchmark programs for c-based high-level synthesis." [Online]. Available: http://www.ertl.jp/chstone/
[9] M. Lv *et al.*, M. Lv, W. Yi, N. Guan, and G. Yu, "Combining abstract interpretation with model checking for timing analysis of multicore software," in *Proc. RTSS*, 2010.
[10] R. Pellizzoni *et al.*, "Worst case delay analysis for memory interference in multicore systems," in *Proc. DATE*, 2010.
[11] M. Steine *et al.*, "A priority-based budget scheduler with conservative dataflow model," in *Proc. DSD*, 2009.
[12] R. Wilhelm *et al.*, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 7, 2009.
[13] M. H. Wiggers *et al.*, "Modelling run-time arbitration by latency-rate servers in dataflow graphs," in *Proc. SCOPES*, 2008.
[14] B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems*. Springer, 2011.
[15] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, 1997.