A Novel Analytical Technique for Timing Analysis of FMTV 2016 Verification Challenge Benchmark

Junchul Choi, Donghyun Kang, and Soonhoi Ha
Department of Computer Science and Engineering, Seoul National University, Seoul, Korea,
Email: {hinomk2, kangdongh, sha}@iris.snu.ac.kr

***Abstract*—In this paper, we present solutions to FMTV 2016 verification challenges, combining the response time analysis and schedule time bound analysis. The worst case response time of a task is computed by the conventional response time analysis while the end-to-end latency of a cause-effect chain is conservatively estimated by considering the schedule time bounds of associated runnables. Three separate challenges are discussed in order. The proposed technique is first explained to address the first challenge that ignores the memory latency. For the second challenge, we estimate the memory access latency by computing the maximum possible arbitration delay with arrival curve analysis. Finally, we propose a heuristic algorithm that determines the mapping of data labels to optimize the end-to-end latency.**

I. Challenge Model and Terminologies

We first review the Amalthea performance model [1] of the benchmark, making some assumptions for unclear explanation in the provided problem specification [2][3].

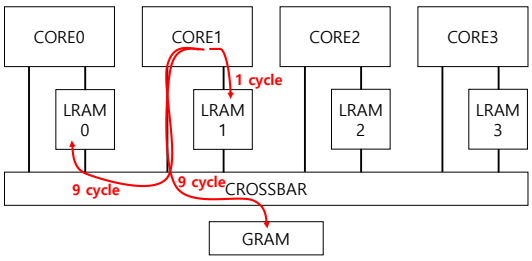

Fig. 1. Microcontroller architecture used in the challenge

The provided Amalthea model contains a hardware model of a simplified microcontroller architecture with four symmetric cores as shown in Fig. 1. Each core A_i^C has its own local memory A_i^L. A crossbar network is used for the interconnection among cores and a global memory A^G.

A task τ_i is a basic mapping unit onto a core and task-to-core mapping is given. The core τ_i is mapped to is denoted by m_i. A task is invoked either periodically or sporadically. I_P and I_S denote a set of periodic tasks and a set of sporadic tasks, respectively. The minimum and the maximum initiation interval are specified for each task τ_i and are denoted as p_i^l and p_i^u. If τ_i is a periodic task ($\tau_i \in I_P$), p_i^l is equal to p_i^u which means that the initiation interval becomes the period. All tasks are simultaneously initiated at the system activation time. The basic timing requirement for task τ_i is to finish execution before its deadline denoted by d_{τ_i}. Since implicit deadline model is assumed, deadline d_{τ_i} is equal to p_i^l.

A task τ_i consists of a set of runnables $\{r_{i,j} \mid 1 \leq j \leq |\tau_i|\}$ where runnable $r_{i,j}$ is an unit of execution and $|\tau_i|$ means the number of runnables in the task. Runnables in a task are executed sequentially on the mapped core in the ascending index order. The lower and the upper bound of execution time of $r_{i,j}$, denoted $c_{i,j}^l$ and $c_{i,j}^u$, are specified assuming that code is executed directly from core-exclusive flashes without contention. Note that memory access delay is not included in the execution times. The runnables are assumed to read all required data at the beginning of their execution and write back the results after execution is completed. We assume that when a runnable attempts to access a memory, no preemption is allowed until the resource request is processed. 1250 runnables are specified in the provided model.

A distinct priority is assigned to each task for the fixed-priority scheduling. We assign each task a unique index in the descending priority order; task τ_i has a higher priority than τ_j if $i < j$. A task τ_i is scheduled by either preemptive or cooperative fixed priority scheduling policy. S_P and S_C denote a set of preemptive tasks and a set of cooperative tasks, respectively. A task $\tau_i \in S_P$ can preempt lower priority tasks at any time, whereas a task $\tau_i \in S_C$ can preempt lower priority cooperative tasks at the boundary of runnable executions [4]. There are 21 tasks and preemptive tasks have higher priorities than cooperative tasks in the provided model.

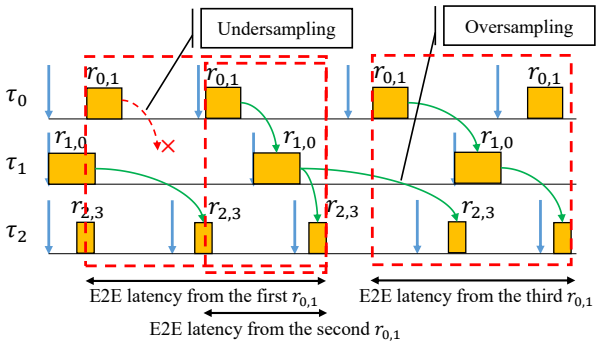

Fig. 2. End-to-end latency of an example cause-effect chain

A cause-effect chain CEC_i defines a chain of runnables that are connected by read/write dependency with labels. Note that there are no cyclic dependencies between tasks within a cause-effect chain. Due to the potential different task periods, data may get lost (undersampling) or get duplicated (oversampling).

We assume an end-to-end latency of a cause-effect chain as the maximum time duration between the first input that may be undersampled and the first output generated from the corresponding or later input. This semantic is as same as the *reaction time constraint* of the AUTOSAR [5]. Fig. 2 shows end-to-end latencies from three stimulus runnable instances in an example cause effect chain $\{r_{0,1}, r_{1,0}, r_{2,3}\}$. Since we are concerned about reaction time, the second $r_{1,0}$ instance is regarded as the reaction of the first $r_{0,1}$ instance. The third $r_{2,3}$ instance is the first response to the second $r_{1,0}$ instance so that final reaction of the first $r_{0,1}$ instance is generated by the third $r_{2,3}$ instance. Three cause-effect chains are specified in the provided model.

Data is specified by a set of labels: each size is less than the memory transfer size 32bits. Memory arbitration model is assumed differently in each challenge as follows:

- **Challenge 1: calculate tight end-to-end latencies ignoring memory accesses and arbitration**
  All read/write accesses to labels take zero time so that only runnable execution times affect the end-to-end latencies.
- **Challenge 2: calculate tight end-to-end latencies including memory accesses and arbitration**
  All labels are assumed to be stored in the global memory. Read and write accesses have symmetric memory access times. When accessing the global memory, crossbar transfer takes 8 cycles and access to global memory takes 1 cycle. When there is a contention at global memory, the accesses are assumed to be arbitrated according to the FIFO policy.
- **Challenge 3: optimize end-to-end latencies by mapping the labels among the local and global memories**
  We can map a label in a local memory whose access latency is 1 cycle. We assume that local memory size is limited. Local memories are also arbitrated according to the FIFO policy.

For all challenges, we aim to conservatively estimate the upper bound of response time of each task τ_i, denoted as L_{τ_i}, and end-to-end latency of cause-effect chain CEC_i, denoted as L_{CEC_i}, as tightly as possible.

II. Proposed Solution Technique for Challenge 1

Since memory access delay is ignored in challenge 1, we compute the worst-case response time of a task τ_i, considering the execution times only.

A. End-to-end latency of a preemptive task

If a higher priority task is released during the execution of a preemptive task $\tau_c \in S_P$, it is preempted by all runnables in the higher priority task. Thus we can formulate the upper bound of the latency between the release time of a runnable $r_{c,i}$ to the finish time of a runnable $r_{c,j}$, denoted $UBL^f(r_{c,i}, r_{c,j})$ where $1 \le i \le j \le |\tau_c|$, as follows using the response time analysis:

$$UBL^f(r_{c,i}, r_{c,j}) = \sum_{k=i}^{j} c^u_{c,k} + \sum_{\tau_h \in hp(\tau_c)} \left(\left\lceil \frac{UBL^f(r_{c,i}, r_{c,j})}{p^l_h} \right\rceil \cdot \sum_{k=1}^{|\tau_h|} c^u_{h,k} \right) \quad (1)$$

where $hp(\tau_c) = \{\tau_h | m_h = m_c, c > h\}$ is a set of higher priority tasks. Then the estimated end-to-end latency of a preemptive task τ_c becomes $L_{\tau_c} = UBL^f(r_{c,1}, r_{c,|\tau_i|})$.

B. End-to-end latency of a cooperative task

For a cooperative task τ_c, the release of τ_c can be blocked by at most one runnable execution of a lower priority task mapped on the same core. Higher priority cooperative tasks released after the start time of a runnable $r_{c,j}$ have no effect on the finish time. We formulate the upper bound of the latency between release time of a runnable $r_{c,i}$ to start time of a runnable $r_{c,j}$, denoted $UBL^s(r_{c,i}, r_{c,j})$ where $1 \le i \le j \le |\tau_c|$ as follows:

$$UBL^s(r_{c,i}, r_{c,j}) = \left(i = 1? \max_{r_{l,k} \in \cup lp(\tau_i)} c^u_{l,k} : 0 \right) + \sum_{k=i}^{j-1} c^u_{c,k} + \sum_{\tau_h \in hp(\tau_c)} \left(\left\lceil \frac{UBL^s(r_{c,i}, r_{c,j}) + 1}{p^l_h} \right\rceil \cdot \sum_{k=1}^{|\tau_h|} c^u_{h,k} \right) \quad (2)$$

where $lp(\tau_c) = \{\tau_l | m_l = m_c, c < l\}$ is a set of lower priority tasks mapped on the same core. The first, second, and third terms indicate the maximum blocking from a lower priority task, the sum of maximum execution times of runnables, and the maximum preemptions from higher priority tasks, respectively. Blocking delay is zero when $i \neq 1$ since any lower priority task cannot start after the first runnable starts. Note that $UBL^s(r_{c,i}, r_{c,j}) + 1$ is used in the third term to include the higher priority tasks released between the finish of the $(j-1)$-th runnable and the start of the (j)-th runnable. Then $UBL^f(r_{c,i}, r_{c,j})$ can be estimated as follows:

$$UBL^f(r_{c,i}, r_{c,j}) = \left(i = 1? \max_{r_{l,k} \in \cup lp(\tau_i)} c^u_{l,k} : 0 \right) + \sum_{k=i}^{j} c^u_{c,k} + \sum_{\tau_h \in \in hp(\tau_c) \cap S_P} \left(\left\lceil \frac{UBL^f(r_{c,i}, r_{c,j})}{p^l_h} \right\rceil \cdot \sum_{k=1}^{|\tau_h|} c^u_{h,k} \right) + \sum_{\tau_h \in hp(\tau_c) \cap S_C} \left(\left\lceil \frac{UBL^s(r_{c,i}, r_{c,j})}{p^l_h} \right\rceil \cdot \sum_{k=1}^{|\tau_h|} c^u_{h,k} \right) \quad (3)$$

All requests of higher priority preemptive tasks within $UBL^f(r_{c,i}, r_{c,j})$ are accounted in the third term while the requests of higher priority cooperative tasks after $r_{c,j}$ starts are excluded. Then the estimated worst-case response time of a cooperative task τ_c becomes $L_{\tau_c} = UBL^f(r_{c,1}, r_{c,|\tau_i|})$.

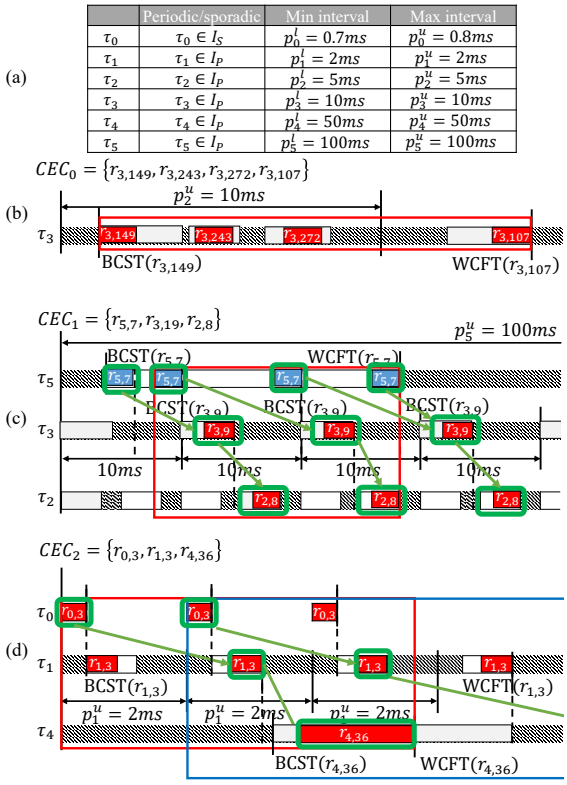

Fig. 3. End-to-end lantency computation of three example cause-effect chains. A white box indicates the schedule time bound of a runnable while a red or a blue box indicates an execution time of the runnable.

C. End-to-end latency of a cause-effect chain

In this section, we compute the end-to-end latency of a cause-effect chain. Before explaining the latency computation, we define two variables $BCST(r_{c,i})$ and $WCFT(r_{c,i})$ which mean a lower bound of start time of $r_{c,i}$ and an upper bound of finish time of $r_{c,i}$ respectively. $WCFT(r_{c,i})$ is formulated as $WCFT(r_{c,i}) = UBL^f(r_{c,1}, r_{c,i})$. Since a cooperative task is not blocked by a low priority task in the best case, $BCST(r_{c,i})$ for either a preemptive task or a cooperative task can be formulated in the same way:

$$BCST(r_{c,i}) = \sum_{k=1}^{i-1} c_{c,k}^l + \sum_{\tau_h \in hp(\tau_c)} \left(\left\lceil \frac{\max(0, BCST(r_{c,i}) - \delta_h + 1)}{p_h^u} \right\rceil \cdot \sum_{k=1}^{|\tau_h|} c_{h,k}^l \right) \quad (4)$$

where $\delta_h = p_h^u - \sum_{k=1}^{|\tau_h|} c_{h,k}^l$. For each higher priority task, the maximum initiation interval and the minimum execution times are considered to compute the minimum interference.

A cause-effect chain is defined by a sequence of runnables that have read/write dependency over a label between each pair of runnables. Fig. 3 shows three example cause-effect chains and the activation patterns of five tasks are summarized in Fig. 3 (a). A cause-effect chain CEC_0 in Fig. 3 (b) consists of four runnables in the same task τ_3. In this case, we have to analyze how many task instances are involved in the chain. If the (i+1)-th runnable of the chain has a smaller index than the i-th runnable, labels written by the i-th runnable will be read by the (i+1)-th runnable in the next task instance. Hence the number of the instances involved in the chain is computed by counting how many times runnable indices decrease in the task sequence. In Fig. 3 (b), two task instances are involved in the chain since index decrease appears only once in the chain ($r_{3,272} \rightarrow r_{3,107}$). If one task instance covers the cause-effect chain, the end-to-end latency can be computed as $UBL^f(r_{c,b}, r_{c,e})$ where $r_{c,b}$ and $r_{c,e}$ are the first and the last τ_c runnables in the chain. Otherwise, the worst-case end-to-end latency becomes the distance from the BCST of the first runnable to the WCFT of the last runnable plus the task period multiplied by the count of index decreases in the chain, which gives $p_3^u + WCFT(r_{3,107})$ - $BCST(r_{3,149})$ for the example of Fig. 3 (b).

A cause-effect chain CEC_1 in Fig. 3 (c) consists of three runnables with different activation patterns. In this case, we consider the schedule time bound of the first runnable ($BCST(r_{5,7})$, $WCFT(r_{5,7})$) and examine all possible BCSTs of the second runnable $r_{3,19}$ that may appear after the first runnable. In the example of Fig. 3 (c), there are three possible BCSTs of $r_{3,19}$. If we consider a pair of runnables only, the worst-case scenario is that the second runnable starts just before the first runnable finishes and the label written by the first runnable is read by the second runnable at the latest in the next task instance. Based on this observation we define a set of starting points of the first runnable as shown in blue color in the figure. The set includes the schedule of the first runnable whose finish time coincides with a possible BCST of the second runnable as well as the earliest and the latest schedule within the schedule bound.

For the subsequent pair of runnables, for instance the second and the third runnables in the example of Fig. 3 (c), we need to consider the schedule time bound of the successor and the WCFT of the predecessor. If the WCFT of the predecessor lies in the schedule time bound of the successor, the label written by the predecessor should be read by the successor runnable at the latest in the next task instance. For each candidate starting point of the first runnable in the chain, the figure shows the longest cause-effect chain by green arrows where red and blue boxes mean the executions of runnables. Among all candidate starting points, we find one that gives the worst-case chain latency that is represented by a red bounding box in the figure, which corresponds to the second candidate starting point.

In this example, we consider a single runnable involved in each task. In case more than one runnable of the same task is included in the chain, we group them as a sub-chain. Then, a cause-effect chain consists of a sequence of sub-chains where each sub-chain consists of a set of runnables in the same task. If the worst-case latency of the sub-chain spans more than one task instance like the case of Fig. 3 (b), we need to consider only one starting point for the sub-chain for the second case.

The third case shown in Fig. 3 (d) is the case that the cause-

effect chain starts with a sporadic task: the first runnable in CEC_2 belongs to a sporadic task τ_0. Since the sporadic task may start anytime, we find the worst-case scenario in which the finish time of $r_{0,3}$ is aligned with the best case start time of the first $r_{1,3}$ instance. Then the end-to-end latency from $r_{0,3}$ to $r_{1,3}$ is bounded by $UBL^f(r_{0,3}, r_{0,3}) + p_1^u + WCFT(r_{1,3}) - BCST(r_{1,3})$. Note that we need to check only one starting point, which makes the finish time of the sub-chain be aligned with the best case start time of next sub-chain, unlike the case of periodic tasks in Fig. 3 (c). We repeat this computation for all task instances of the first periodic task in the chain within the hyper-period of tasks. In Fig. 3 (d), τ_1 is the first periodic task. If we repeat computation for all τ_1 instances, the maximum latency occurs with the third τ_1 instance since labels written by the third $r_{1,3}$ instance is missed by the first $r_{4,36}$ instance.

Algorithm 1 Algorithm to compute the end-to-end latency of a cause-effect chain

```
1: E2E ← 0, d_len ← 0
2: if the first sub-chain is in a sporadic task then
3:     d_len ←end-to-end latency of the first sub-chain
4:     while all sporadic sub-chains before the first periodic sub-chain do
5:         d_len ← d_len + (one period) + (WCFT of the last runnable) −
   (BCST of the first runnable)
6:     end while
7: end if
8: for all instances of the first periodic sub-chain within hyperperiod do
9:     find all candidate starting points of the first runnable
10:    for all candidate starting points do
11:        start ← (candidate starting point)
12:        end ←corresponding end point
13:        for all sub-chains after the first periodic sub-chain do
14:            if sub-chain is in a sporadic task then
15:                end ← end + (one period) + (WCFT of the last
   runnable) − (BCST of the first runnable)
16:            else
17:                end ←minimum WCFT among runnable instances whose
   BCST is no smaller than end
18:            end if
19:        end for
20:        E2E ← max(E2E, end − start)
21:    end for
22: end for
23: return E2E + d_len
```

Now we summarize the proposed technique for the estimation of the end-to-end latency of a cause-effect chain with Algorithm 1. At first, if the chain starts with sporadic tasks, we compute the end-to-end latency d_len of those sporadic sub-chains (lines 2-7). Then for the first periodic sub-chain, we examine all instances of the first periodic sub-chain within the hyperperiod of the chain. (lines 8-23). For each instance, we find all candidate starting points and compute the latency from the starting point to the end time of the chain (lines 9-21). If the chain starts with a sporadic task or a sub-chain that spans more than one task instance, we need to consider only one starting point which is the BCST of the runnable. Otherwise, we find all candidate starting points as Fig. 3 (c). From each starting point, we find the end point of the chain (lines 13-19).

III. Proposed Solution Technique for Challenge 2

In the second challenge, we consider the worst-case memory access delay in the latency computation. Since memory accesses are arbitrated according to the FIFO policy and a core is assumed to be blocked during memory access, one memory access may be delayed by at most three accesses (one per each core). Hence a naive way to find a conservative upper bound is to assume that each access experiences blocking by three queued accesses. To find a tighter bound of memory access delay, however, we analyze the maximum number of memory accesses issued by tasks in each core within any time window of size Δt by adopting the event stream model [6]. Then we can bound the number of memory accesses that are issued from remote cores. For example, if there are total 10 accesses during the worst-case response time of a task τ_i, L_{τ_i}, and all accesses are assumed to be blocked by three accesses, the total memory access delay will be $10 \cdot (8+3+1)$ cycles. If we know that some cores cannot issue more than 10 accesses within any time window of size L_{τ_i}, we can tighten the upper bound of memory access delay.

Since we aim to find the maximum number of accesses within a time window, we consider the lower bound of execution time and the lower bound of initiation interval in this section. For brevity, we define a variable $\mathcal{C}_i$ as the sum of best case execution times of all runnables in τ_i plus memory access delay without contention.

For a given time window of size Δt, we have to compute the maximum memory access requests from each core. To tackle this problem, several approaches that find an upper bound of the number of shared resource accesses within a time window have been proposed ([7], [8]). In this paper, we propose an improved technique by accounting for the scheduling pattern of tasks. For each core, we have to find out the task execution scenario that produces the maximum memory access requests within the time window. Since the number of task execution scenarios is enormously large, we consider the partitioning of the time window to tasks in the core. The partitioned time means the net execution time of a task. Note that a task may have multiple task instances in the time window that may not be continuous due to preemption or periodic appearance. Since the total execution time within a time window cannot exceed Δt, we check all combinations of task net execution times. For instance, suppose that there are two tasks in a core and $\Delta t = 3$. Then we check all possible combinations of execution time partitions: (0,3), (1,2), (2,1), and (3,0) where (a,b) means the net execution times of two tasks. If we compute the minimum and the maximum bound of net execution time that a task may take within a time window Δt, we can eliminate the infeasible partitions. If the first task cannot take 3 time units in any time window of size 3, (3,0) becomes impossible. With a given net execution time of a task, we find the upper bound of memory access requests.

At first, we define two functions $t_i^{min}(\Delta t)$ and $t_i^{max}(\Delta t)$ that represent the minimum and the maximum execution time amount a task τ_i may take within a time window Δt,

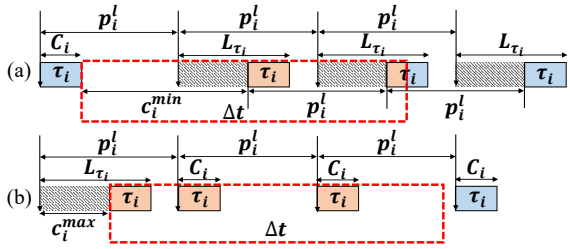

Fig. 4. The minimum execution time scenario (a) and the maximum execution time scenario (b) in a time window Δt

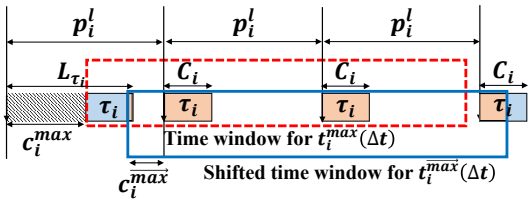

Fig. 5. The shifted time window by $\mathcal{C}_i - 1$ from the time window for $t_i^{max}(\Delta t)$

respectively. Fig. 4 illustrates two scheduling patterns of task τ_i that correspond to $t_i^{min}(\Delta t)$ and $t_i^{max}(\Delta t)$, respectively. In the figure, a dashed rectangle indicates the time window Δt, and the task is invoked with the minimum initiation interval p_i^l. The start time of a task may be delayed by $L_{\tau_i} - \mathcal{C}_i$ in the worst-case by preemption or memory arbitration delay, which is represented as the grey area in the execution profile.

For a task τ_i to take the minimum *net* execution time in the time window, the worst-case interval between two consecutive job instances should be considered. The worst-case interval is observed when an instance finishes its execution as soon as possible with response time of $\mathcal{C}_i$ and the start times of all subsequent instances are maximally delayed by $L_{\tau_i} - \mathcal{C}_i$ as shown in Fig. 4 (a). Then the minimum *net* execution time is found when the time window starts immediately after the finish time of the first instance. In summary, we can derive the function $t_i^{min}(\Delta t)$ as follows:

$$t_i^{min}(\Delta t) = \mathcal{C}_i \cdot \left\lfloor \max(0, \Delta t - c_i^{min})/p_i^l \right\rfloor + \min(\mathcal{C}_i, \max(0, \Delta t - c_i^{min}) \bmod p_i^l) \quad (5)$$

where $c_i^{min} = p_i^l + L_{\tau_i} - 2 \cdot \mathcal{C}_i$. The first term and the second term indicate fully included executions and partially included execution, respectively.

On the contrary, we should consider the shortest interval between two τ_i instances in order to compute the maximum execution time in time window Δt: an instance starts as late as possible to finish at its end-to-end latency L_{τ_i} and subsequent instances start immediately at their request time. The execution time amount is maximized in time window Δt when the time window starts at the start time of the first task instance, as illustrated in Fig. 4 (b). The maximum amount of execution time $t_i^{max}(\Delta t)$ is derived as follows:

$$t_i^{max}(\Delta t) = \min\left(\Delta t, \mathcal{C}_i \cdot \left\lfloor (\Delta t + c_i^{max})/p_i^l \right\rfloor + \min(\mathcal{C}_i, (\Delta t + c_i^{max}) \bmod p_i^l)\right) \quad (6)$$

where $c_i^{max} = L_{\tau_i} - \mathcal{C}_i$. The first term and the second term indicate fully included executions and partially included execution, respectively.

Now we compute how many instances may exist in a time window Δt. Fig. 5 shows the same task schedule scenario of Fig. 4 (b) and the dashed rectangle indicates the time window to achieve the maximum execution time $t_i^{max}(\Delta t)$ in a time window Δt. In order to cover the task instances as many as possible in the time window, we shift the time window to the right direction. If the shift amount is greater than or equal to $\mathcal{C}_i$, the first task instance becomes outside of the time window, making the number of instances decreases. Hence the shift amount should be less than $\mathcal{C}_i$. On the other hand, we need to shift the time window as much as possible to include the instances at the right side of the time window. In summary, to make the maximum number of task instances that may lie in the time window, the time window should be shifted by $\mathcal{C}_i - 1$. In the figure, the shifted rectangle contains one more instance of the task than the dashed rectangle.

Note that when the number of task instances laid in the time window is maximized, the *net* execution time may be smaller than that for the case when the net execution time in the time window is maximized. Hence we need to compare two cases to find the maximum possible resource demand; (1) the case the number of task instances is maximized and (2) the case the net execution time is maximized. Then we need to compute the maximum *net* execution time for the first case and the maximum number of task instances for the second case. The number of instances for the second case can be computed as $n_i(\Delta t) = \left\lceil \frac{t_i^{max}(\Delta t)}{\mathcal{C}_i} \right\rceil$. We denote the maximum *net* execution time for the first case $\overrightarrow{t_i^{max}}(\Delta t)$, where arrow indicates that the time window is shifted by $\mathcal{C}_i - 1$ to maximize the number of instances. Then $\overrightarrow{t_i^{max}}(\Delta t)$ can be formulated as follows:

$$\overrightarrow{t_i^{max}}(\Delta t) = \mathcal{C}_i \cdot \left\lfloor \max(0, \Delta t - \overrightarrow{c_i^{max}})/p_i^l \right\rfloor + \min(\mathcal{C}_i, \max(0, \Delta t - \overrightarrow{c_i^{max}}) \bmod p_i^l) \quad (7)$$

where $\overrightarrow{c_i^{max}} = p_i^l - L_{\tau_i}$. We denote the maximum number of τ_i instances laid in the time window Δt as $\overrightarrow{n_i}(\Delta t)$ and $\overrightarrow{n_i}(\Delta t)$ can be computed from $\overrightarrow{t_i^{max}}(\Delta t)$ to be $\left\lceil \frac{(\overrightarrow{t_i^{max}}(\Delta t) - 1)}{\mathcal{C}_i} \right\rceil + 1$.

Finally, we formulate memory access bound function $D_{A_i^C, A^G}(\Delta t)$ which finds the maximum number of accesses from a core A_i^C to global memory A^G within any time window of size Δt. When we distribute the time amount Δt to tasks mapped onto A_i^C, we should consider the constraint that a task τ_k can be assigned the bounded *net* execution time t_k between $t_k^{min}(\Delta t)$ and $t_k^{max}(\Delta t)$. And, for a given Δt, we consider two cases where the number of instances of a task τ_k is $n_k(\Delta t)$ or $\overrightarrow{n_k}(\Delta t)$ as the access bound function of each individual task, $D_{\tau_k, A^G}(t_k, \Delta t)$. In summary, the memory access bound

function is formulated as follows:

$$D_{A_i^C,A^G}(\Delta t) = \max \left\{ \sum_{m_k=A_i^C} D_{\tau_k,A^G}(t_k,\Delta t) \;\middle|\; \begin{array}{c} \sum_{m_k=A_i^C} t_k = \Delta t, \\ \forall_{m_k=A_i^C} t_k \geq t_k^{min}(\Delta t) \end{array} \right\} \tag{8}$$

$$D_{\tau_k,A^G}(t_k,\Delta t) = \max \begin{pmatrix} \eta_{k,A^G}^{e(n_k(\Delta t))}(\min(t_k, t_k^{max}(\Delta t))) \\ \eta_{k,A^G}^{e(\overrightarrow{n_k}(\Delta t))}(\min(t_k, \overrightarrow{t_k^{max}}(\Delta t))) \end{pmatrix} \tag{9}$$

where $\eta_{k,A^G}^n(t)$ is the maximum number of resource accesses that may be issued from n instances of a task τ_k to a memory A^G when the net execution time of τ_i does not exceed t time units. $\eta_{k,A^G}^n(t)$ can be computed by moving the time window of size t on the n task instances that are executed one after another and finding the maximum number of resource accesses among all time windows.

$D_{A_i^C,A^G}(\Delta t)$ can be obtained by the max-plus convolution of individual demand bound functions of (9) in polynomial time since the max-plus convolution has associative property and commutative property. $D_{A_i^C,A^G}(\Delta t)$ is used to bound the arbitration delay during the latency computed by equations (1), (2), and (3). For a preemptive task, it may be blocked by one memory access from a lower priority task. For a cooperative task, we consider the maximum blocking by one lower priority runnable with its worst case memory access delay (8+3+1 per one access). This blocking delay is independently computed and included in the worst-case latency. After computing the lower priority blocking delay, we consider memory accesses that are issued from the target task and higher priority tasks. To bound the interference from a core, we compute the number of memory accesses from the core during the latency of interest.

IV. Proposed Solution Technique for Challenge 3

In this section, we propose a greedy algorithm that determines a label-to-memory mapping to optimize the end-to-end latencies. If a label is mapped to a local memory A_i^L, we can save the crossbar transfer delay (8 cycles) which is larger than the worst-case arbitration delay (4 cycles).

Algorithm 2 presents a pseudo code of the proposed greedy algorithm to determine label-to-memory mapping. Initially labels are mapped to a global memory A^G (line 8). At first, we compute each fitness value of a mapping of $L[i]$ to A_j^L, $F[i][j]$ (line 9). The fitness value is higher if $L[i]$ is more frequently accessed from A_j^C. Then we determine a mapping of each label (lines 11-18). We select the most beneficial mapping according to the fitness values (line 12). Since we assume a limited local memory size, the label $L[l]$ can be mapped to A_m^L in case A_m^L has enough memory size (lines 13-16). The progress is repeated until there is no mapping that optimize the memory access delay (line 11).

Unlike the latency computation in challenge 2, the memory accesses from A_i^C to A_i^L do not involve transfer delay so that only arbitration delay at the memory should be considered. The technique to compute memory arbitration delay bound explained in challenge 2 can be easily extended to compute the memory access bound function of each memory separately.

Algorithm 2 Greedy algorithm to determine label-to-memory mapping

Input: a set of labels L, an array of label sizes S_L and local memory size s
Output: an array of label mapping M

1: $S_M \leftarrow$ one dimensional array of size 4
2: $F \leftarrow$ two dimensional array of size $|L| \times 4$
3: ▷ $S_L[i]$ is a label size of $L[i]$
4: ▷ $M[i]$ is a memory a label $L[i]$ is mapped to
5: ▷ $S_M[i]$ indicates available memory size of A_i^L
6: ▷ $F[i][j]$ is a fitness value of a mapping of $L[i]$ to A_j^L
7: **for** $0 \leq i < |L|, 0 \leq j < 4$ **do**
8: $M[i] \leftarrow A^G$, $S_M[j] \leftarrow s$
9: $F[i][j] \leftarrow \sum_{m_k=A_j^C} \frac{\#\text{accesses of } \tau_k \text{ to } L[i]}{p_k^l}$
10: **end for**
11: **while** $\exists_{i,j} F[i][j] > 0$ **do**
12: find indices l and m that $F[l][m] = \max_{i,j} F[i][j]$
13: **if** $S_M[m] \geq S_L[l]$ **then**
14: $S_M[m] \leftarrow S_M[m] - S_L[l]$
15: $M[l] = A_m^L$
16: **end if**
17: $F[l][m] \leftarrow 0$
18: **end while**

V. Challenge Results

The estimated end-to-end latencies of all tasks and cause-effect chains from the proposed technique are summarized in Table. I. In the table, WCRT and E2E L. mean the worst-case response time and the end-to-end latency, respectively. (C1), (C2), and (C3) columns show the estimated results for challenge 1, challenge 2, and challenge 3, respectively. We assume unlimited local memory size in the experiment since no constraints are given.

Even without memory access delay, 6 out of 21 tasks in the challenge model are unschedulable according to our analysis results since core utilizations are too high: utilizations are 97%, 133.5%, 106.8%, and 117.9% for each core. There is even a task that has the worst-case execution time larger than its deadline (Task_10ms). End-to-end latencies of cause-effect chains cannot be analyzed due to the runnables in unschedulable tasks. We claim that the worst-case execution time should be decreased to make the system schedulable.

Results show that the portion of the memory access delay in the worst-case response time is not significant. Task_50ms becomes unschedulable when memory access delay is not ignored. Because of the memory access delay, its worst-case response time becomes over 8,000,000 and one more preemption of Task_20ms whose worst-case execution time is 2,093,688 occurs, making the response time larger than the deadline. Almost all read/write accesses go to local memory after label-to-memory mapping is done so that the memory access delay decreases accordingly. Task_20ms is barely schedulable after label-to-memory mapping.

We conducted additional experiment to find maximum execution times of tasks satisfying all task deadlines. For each core, we scale down all worst-case execution times of mapped

TABLE I
END-TO-END LATENCIES OF TASKS AND CAUSE-EFFECT CHAINS SPECIFIED IN THE PROVIDED SYSTEM MODEL (UNIT: CYCLE)

Task		WCRT (C1)	WCRT (C2)	WCRT (C3)
CORE0	ISR_10 (τ_0)	6,068	6,308	6,112
	ISR_5 (τ_1)	57,704	58,256	57,785
	ISR_6 (τ_2)	63,894	64,698	63,996
	ISR_4 (τ_3)	137,054	138,278	137,206
	ISR_8 (τ_4)	261,725	263,843	261,973
	ISR_7 (τ_5)	530,598	534,453	531,061
	ISR_11 (τ_6)	853,378	859,207	854,081
	ISR_9 (τ_7)	unschedulable	unschedulable	unschedulable
CORE1	Task_1ms (τ_{11})	152,870	156,345	153,588
	Angle_Sync (τ_{12})	unschedulable	unschedulable	unschedulable
CORE2	Task_2ms (τ_{13})	80,817	82,425	81,188
	Task_5ms (τ_{14})	267,180	270,252	267,900
	Task_20ms (τ_{16})	3,709,404	3,760,278	3,719,254
	Task_50ms (τ_{17})	7,973,611	unschedulable	7,992,287
	Task_100ms (τ_{18})	unschedulable	unschedulable	unschedulable
	Task_200ms (τ_{19})	unschedulable	unschedulable	unschedulable
	Task_1000ms (τ_{20})	unschedulable	unschedulable	unschedulable
CORE3	ISR_1 (τ_8)	7,011	7,383	7,066
	ISR_2 (τ_9)	10,560	11,160	10,635
	ISR_3 (τ_{10})	15,347	16,247	15,448
	Task_10ms (τ_{15})	unschedulable	unschedulable	unschedulable

Cause-effect chain	E2E L. (C1)	E2E L. (C2)	E2E L. (C3)
EffectChain_1	unschedulable	unschedulable	unschedulable
EffectChain_2	unschedulable	unschedulable	unschedulable
EffectChain_3	17,817,190	unschedulable	17,835,552

TABLE II
SCALED WORST-CASE EXECUTION TIMES FOR SCHEDULABLE SYSTEM AND END-TO-END LATENCIES (UNIT: CYCLE)

Task		WCET	WCRT	Deadline
CORE0 (96%)	ISR_10 (τ_0)	5,825	5,867	140,000
	ISR_5 (τ_1)	49,570	55,472	180,000
	ISR_6 (τ_2)	5,942	61,434	220,000
	ISR_4 (τ_3)	70,233	131,706	300,000
	ISR_8 (τ_4)	58,345	251,485	340,000
	ISR_7 (τ_5)	62,375	509,791	980,000
	ISR_11 (τ_6)	58,729	814,042	1,000,000
	ISR_9 (τ_7)	71,133	896,985	1,200,000
CORE1 (71%)	Task_1ms (τ_{11})	108,537	109,164	200,000
	Angle_Sync (τ_{12})	540,360	1,197,321	1,332,000
CORE2 (93%)	Task_2ms (τ_{13})	75,159	75,511	400,000
	Task_5ms (τ_{14})	173,317	249,170	1,000,000
	Task_20ms (τ_{16})	1,947,129	3,383,696	4,000,000
	Task_50ms (τ_{17})	573,714	7,358,075	10,000,000
	Task_100ms (τ_{18})	1,751,743	19,908,947	20,000,000
	Task_200ms (τ_{19})	25,758	19,933,318	40,000,000
	Task_1000ms (τ_{20})	25,511	19,958,468	200,000,000
CORE3 (83%)	ISR_1 (τ_8)	5,819	5,869	1,900,000
	ISR_2 (τ_9)	2,945	8,834	1,900,000
	ISR_3 (τ_{10})	3,973	12,832	1,900,000
	Task_10ms (τ_{15})	1,944,313	1,986,787	2,000,000

Cause-effect chain	E2E Latency
EffectChain_1	2,269,514
EffectChain_2	2,628,493
EffectChain_3	13,888,054

tasks by the same percentage and find the maximum percentages that make all estimated end-to-end latencies of tasks below deadlines. Table II summarizes the scaled worst-case execution times and the end-to-end latencies for challenge 3. Note that the percentage decrease for each core is proportional to the utilization of the core.

VI. CONCLUSION

We present a solution technique to FMTV 2016 verification challenges, combining the response time analysis and schedule time bound analysis. The main contribution is that we consider schedule time bounds of runnables to tightly compute end-to-end latencies of cause-effect chains. Memory access bound functions are described to find the maximum possible arbitration delay with arrival curve analysis. A simple greedy algorithm is proposed to determine label-to-memory mapping. It took about one month to understand the challenge model and to solve the problem, applying the technique we have developed beforehand.

ACKNOWLEDGMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning(NRF-2013R1A2A2A01067907) and MSIP(Ministry of Science, ICT&Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2015-H8501-15-1005) supervised by the IITP(Institute for Information&communications Technology Promotion). The ICT at Seoul National University provides research facilities for this study.

REFERENCES

[1] *AMALTHEA: An Open Platform for Embedded Multicore Systems*. [Online]. Available: http://www.amalthea-project.org/

[2] *2016 Formal Methods for Timing Verification (FMTV) challenge, co-located with the 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. [Online]. Available: https://waters2016.inria.fr/challenge/

[3] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmark for free," in *sixth International Workshop on Analysis Tools and Methodologies for Embedded Real-time Systems (WATERS)*, 2015.

[4] *AUTOSAR Specification of RTE Software*. [Online]. Available: https://www.autosar.org/fileadmin/files/releases/2-0/software-architecture/rte/standard/AUTOSAR_SWS_RTE.pdf

[5] *AUTOSAR Specification of Timing Extensions*. [Online]. Available: https://www.autosar.org/fileadmin/files/releases/4-1/methodology-templates/templates/standard/AUTOSAR_TPS_TimingExtensions.pdf

[6] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the symta/s approach," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 2, pp. 148–166, Mar 2005.

[7] M. Negrean, S. Schliecker, and R. Ernst, "Response-time analysis of arbitrarily activated tasks in multiprocessor systems with shared resources," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, April 2009, pp. 524–529.

[8] S. Schliecker and R. Ernst, "Real-time performance analysis of multiprocessor systems with shared memory," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, pp. 22:1–22:27, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/1880050.1880058