Response-Time Analysis of Parallel Fork-Join Workloads with Real-Time Constraints

Philip Axer, Sophie Quinton,
Moritz Neukirchner, Rolf Ernst
Institut für Datentechnik
TU Braunschweig, Germany

Björn Döbel, Hermann Härtig
Operating Systems Group
TU Dresden, Germany

***Abstract*—The advent of multi- and many-core processors comes with new challenges and opportunities for the designer of embedded real-time applications. By using parallel programming techniques (e.g. OpenMP) software engineers can leverage from the available hardware parallelism and speed up the algorithms. The inherent redundancy of multi-core architectures can also be used to implement fault-tolerance by executing code redundantly on multiple cores in parallel. Parallel programming and redundant execution are typical examples for fork-join tasks in which the program is partially parallelized. However, complex synchronization of parallel segments across multiple cores can cause unanticipated effects. This is especially problematic in hard real-time applications where data must be available in bounded time (e.g. stereo vision for pedestrian detection). The contribution of this work is a novel worst-case response time analysis which accounts for synchronization of fork-join tasks with arbitrary deadlines. We apply the analysis to the Romain framework which extends the L4 microkernel by redundant multithreading targeted towards fault-tolerant embedded systems. By using formal analysis, we show that parallelizing workloads can lead to drastic performance impairments compared to traditional sequential execution if not done carefully.**

***Keywords*-real time systems, performance analysis, parallel programming, embedded software, fault tolerance, redundancy**

I. Introduction

Recently, multi- and many-core processors emerged on the embedded market and it is predicted that this technology will replace traditional single-core architectures due to their performance advantage. For instance, a commercially available many-core platform targeted for the embedded market is the Tilera TILE-Gx processor family which offers up to 100 cores aimed at video and network processing.

Contrary to desktop computing, embedded software is in many cases subject to real-time constraints. In that sense, a hard real-time application must not only produce correct data, but it must be produced within a constrained time (deadline). Real-time analysis (e.g. [16]) helps the engineer to validate the timing of traditional sequential applications. However, analysis of software which runs on and is tailored to massively parallel hardware is challenging. In this paper we explain how to analyze parallel fork-join workload as often found on today's as well as next generation hardware platforms.

A simple fork-join task is depicted in Figure 1. It differs from traditional (e.g. independent) tasks by their unique precedence relation semantics. The task starts the first stage with a sequential segment and in the second stage it forks into multiple parallel segments. After *all* parallel segments have finished a join-construct synchronizes the execution and the next sequential segment starts. Execution continues this way until the task terminates.

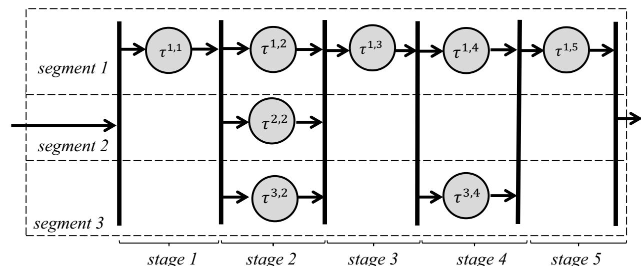

Figure 1. Fork-join model as used in OpenMP or Romain replication framework [8]. A fork-join task can be subdivided into vertical stages and horizontal segments.

There are several ways how embedded programmers already use fork-join constructs especially with focus on new parallel hardware architectures. Tasks can be parallelized and mapped to multiple cores to speed up code paths. This is usually done by using parallel programming techniques (e.g. OpenMP [18]). Here the programmer splits a task into parallelized and sequential segments. Sequential segments are executed on a single resource, whereas parallel segments are distributed (forked) across multiple cores.

Also safety-critical applications such as active steering and autonomous driving can benefit from the vast number of cores by using them for hot redundancy in order to increase reliability on future potentially unreliable hardware [4]. Redundant copies of the same application are executed on several cores and intermediate results are compared on the fly. Self-checking redundant multithreading (Romain framework) was implemented in the L4 Microkernel [8]. Copies of the application run in isolation until interaction with the environment occurres (i.e. a system call). In this case, the operating system waits until all copies of the process have data available, but data is only forwarded after majority voting is performed. In fact, this behavior is precisely modeled by a fork-join task as shown above.

We believe, that there will be no immediate technology jump from single-core to massively parallel many-core archi-

1068-3070/13 $26.00 © 2013 IEEE
DOI 10.1109/ECRTS.2013.31

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on September 28,2023 at 20:06:34 UTC from IEEE Xplore. Restrictions apply.

tectures but rather a gradual update. Thus, traditional sequential software and parallel software will coexist on the same cores and will partially interfere with each other. Obviously, this interference caused by local scheduling effects must be considered when validating real-time constraints.

Contributions: The main contribution of this paper is a worst-case response-time analysis for fork-join task graphs mapped to multi-core architectures using partitioned fixed-priority scheduling. Contrary to related work, we consider task interference with higher priority workload such as sequential tasks as well as higher priority fork-join tasks. We show the applicability and apply the presented approach to the Romain redundancy framework [8].

The paper is structured as follows: First, we discuss related work. In Section III we discuss our system model and further assumptions. In Section IV, we study the impact of fork-join tasks on independent tasks. The main contribution is the response time analysis of fork-join tasks presented in Section V. In Section VI we apply the methodology to real-world examples extracted from the Romain framework. Finally we conclude the work in Section VII.

II. Related Work

Task-parallel programming models facilitate splitting application logic into sequential and parallel parts. Toolkits, such as OpenMP [18] and Intel's Thread Building Blocks [13] support the programmer by automating most of the parallelization and synchronization work. However, such runtimes make real-time analysis harder, which is a point we address in this paper.

Classical response-time analyses are available from real-time research for a large variety of different scheduling policies. They can be directly applied to single processor systems. For example, when computing the worst-case response time of a task, which is the largest time from its release until completion, one can rely on the *busy window* technique [16, 25]. Schliecker et al. provided an extension of the busy window approach in [23] to consider the effects of shared resource conflicts as often seen in multicore architectures (e.g. memory controller, shared busses). Lakshmanan et al. presented the partitioned multiprocessor scheduling in [15] and proved utilization bounds for partitioned deadline-monotonic scheduling (PDMS) in which task mapping is fixed. Holenderski et al. [12] addressed multi-resource scheduling in which parallel tasks can access local and global resources which can be preemptible as well as non-preemptible. In that scope a generalized shared resource protocol (Parallel-SRP) was presented. Baruah et al. presented a generalized parallel task model [2] which also supports fork-join tasks and study the schedulability under EDF and conclude that EDF has a speedup bound of 2. Fork-join task models in particular were considered in [21], the presented model is compatible with our work. The authors decomposed fork-join task constraints into a set of sequential deadline constraints under implicit deadline assumption. Based on this, schedulability bounds for global EDF scheduling were given. Lakshmanan et al. introduced a stretch transformation in [14] in order to fracture workload as little as possible.

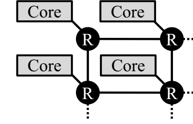

Figure 2. Standard multi-core architecture. Processing elements are connected to an on-chip network. Tasks are mapped to cores and scheduled by an operating system.

There exist a large variety of scheduling and mapping techniques to handle fork-join tasks such as [17]. Here deadlines were assigned to a fork-join task and an EDF-like scheduler was used to schedule all subtasks in a fork-join task independently. In [10] an algorithm was given to map tasks to cores under a partitioned multiprocessor scheme. Fork-join decomposition and priority assignment were put together in [9] in scope of RT-OpenMP.

Most of related work in the field of parallel task graphs only considers global EDF or variations thereof under implicit deadline assumptions (i.e. [6, 15, 2, 21, 9]). To the best of our knowledge no response time analysis is yet available for fork-join workloads under partitioned fixed-priority scheduling.

III. Assumptions, System and Task Model

We consider a multi-core system consisting of multiple processing elements connected by a communication fabric such as a network on chip (NoC). An example for such an architecture is depicted in Figure 2. We assume that tasks are managed by an operating system and are statically mapped to individual cores. This is a valid assumption, since partitioned scheduling schemes are widely used (e.g. AUTOSAR [1]) and well understood [5].

Additional communication overhead such as NoC overhead, cache coherency traffic or shared resource accesses will not be explicitly modelled in this paper (i.e. we assume that any overhead is accounted in the execution times or application graph as described below).

A. Task Model

Before describing the properties of a fork-join task, it must be highlighted that we support two types of tasks in our system.

- *independent* tasks and
- *fork-join* tasks

An independent task τ executes for at most time C once it is activated, is mapped to one core denoted by M and has a static priority p. We assume preemptive fixed-priority scheduling, so a high priority task can preempt a low priority task at any time.

A fork-join task Γ is an extension of an independent task and consists of multiple stages with further data dependency. A fork-join task as shown in Figure 1 is a directed acyclic graph consisting of a set of independent tasks and edges between tasks which describe precedence dependencies.

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on September 28,2023 at 20:06:34 UTC from IEEE Xplore. Restrictions apply.

In any fork-join graph, we can identify *segments* and *stages* as annotated in Figure 1. In that sense, a segment can only be started once all segments released in the previous stage have finished their execution. Each segment is modeled by an independent task, thus it has a worst-case execution time, priority and unique mapping. Furthermore, we assume that events are queued at the first stage of a fork-join task, so that only one event at a time (fifo) is processed. A queued event is admitted, once the previous event exits the last stage. Nested forks in which only some segments have common synchronization points are not supported.

In the previous section we motivated the use-case for sequential and parallel stages, however this concept can be generalized. It can be observed that a sequential stage is equivalent to a parallel stage consisting of only one segment. Thus, we do not need to differentiate between sequential and parallel stages, since the number of segments in a stage is not restricted.

Now we establish some short-hand notations to reference individual segments and their respective properties. A fork-join task Γ is represented by a set of regular tasks which we call subtasks $\tau^{\sigma,s}$. Here, $\tau^{\sigma,s}$ is the σ-th segment in the s-th stage as can be seen in Figure 1. Similarly, a fork-join task is parametrized by a set of execution times and priorities $C^{\sigma,s}, p^{\sigma,s}$ one for each subtask $\tau^{\sigma,s}$. For now, we assume that all subtasks in one segment are mapped to the same core. That is all tasks in the same row in Figure 1 are mapped to one core. Thus σ implicitly encodes the mapping for fork-join tasks, and we can say $\tau^{\sigma,s}$ is mapped to core σ. In that sense σ can be a segment as well as a core.

B. Event Models

The dataflow into a task (i.e. fork-join or independent) is modeled with the help of *event models*. Event models abstract the activation of tasks from an actual trace by representing only the worst-case behavior.

Following this concept the arrival curve $\eta(\Delta t)$ describes the maximum number of events which can arrive during any given time window Δt at the input and be queued for processing. Thus, a given event model η is always a conservative approximation for any actual event trace that is smaller than η.

An alternative representation is the notion of a minimum distance between n subsequent events $\delta(n)$. As shown in [24], both representations η and δ are pseudoinverse and can be converted to each other.

$$\delta(n)=\min_{\Delta t\geq 0, \Delta t\in\mathbb{R}}\{\Delta t|\eta(\Delta t)\geq n\} \tag{1}$$

$$\eta(\Delta t)=\max_{n\geq 1, n\in\mathbb{N}}\{n|\delta(n)\leq\Delta t\} \tag{2}$$

Standard event models [20] such as periodic with jitter, sporadic and others can be expressed in this way. The δ function for a bursty input with a given period $\mathcal{P}$, jitter $\mathcal{J}$ and minimal distance d^{min} between any two events is defined as:

$$\delta(n)=max((n-1)d^{min},(n-1)\mathcal{P}-\mathcal{J}) \tag{3}$$

After describing the model which is used throughout the paper, we can now define the response time which we want to derive.

Definition 1 (Response Time)**.** *The response time of an event of task τ is the time from the arrival of the event until it has fully been processed.*

Definition 2 (Fork-Join Response Time)**.** *The response time of an event of fork-join task Γ is the time interval defined by the time when the event arrives at a fork-join task until it leaves the last stage.*

The worst-case (fork-join) response time R is an upper bound to any response time which can be observed.

To obtain the worst-case (fork-join) response time, we will use the busy-window approach. Before we dive into fork-join specific concepts, we establish general notions as used in any busy-window analysis. However, contrary to established definitions we first give a generalized definition of the busy window concepts. This is necessary since traditional definitions (i.e. those given in [16]) do not apply in our case as we will see later.

Definition 3 (Busy Window)**.** *A busy window w of a task is the time interval in which all response times of the task depend on the execution of at least one previous activation in the same busy window, except for the very first activation of the task.*

Or informally rephrased: the busy window is the time interval in which events released earlier have a "timing effect" on events released later. Naturally, only activations inside the largest busy-window w need to be evaluated [16].

Definition 4 (Multiple-Event Busy Time)**.** *The worst-case multiple-event busy time $B(n)$ of a task is the largest time interval from the arrival of the first activation until n consecutive activations of the task have been fully processed, assuming all events arrive in the same busy window.*

Definition 5 (Multiple-Event Queuing Delay)**.** *The worst-case multiple event queuing delay $Q(n)$ of a task is the largest time interval from the arrival of the first activation until the n-th event receives service for the first time under the assumption that all events arrive in the same busy window.*

We now show how the largest busy window can be obtained from the multiple-event busy time and queuing delay.

Theorem 1. *The busy window can be retrieved by:*

$$w=\min_{n\geq 1, n\in\mathbb{N}}\{B(n)|Q(n+1)<\delta(n+1)\} \tag{4}$$

Proof: The proof is by contradiction. We assume there exists an even larger busy window $\hat{w}$ which includes even more activations than those in w. We can conclude that according to eq. 4 busy window $\hat{w}$ contains at least one event $\hat{n}$ for which $Q(\hat{n})<\delta(\hat{n})$.

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on September 28,2023 at 20:06:34 UTC from IEEE Xplore. Restrictions apply.

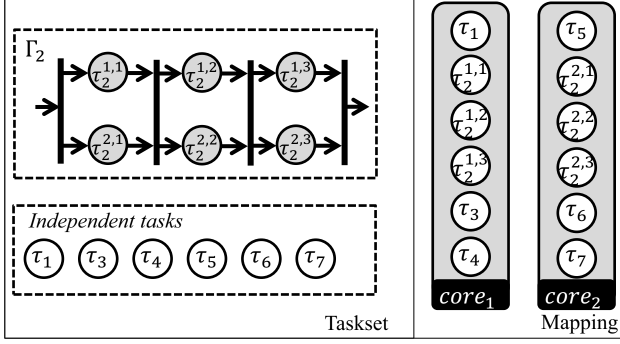

Figure 3. Example task set and mapping as used for illustrative purposes. Two cores, one fork-join task Γ_2 and some individual tasks.

However, this implies that the event $\hat{n}$ could get service prior to its actually occurrence. This means that the queuing delay of the actual occurrence of $\hat{n}$ cannot be affected by previous events. If the queuing is not affected by previous activations, then the response time can also be not affected. This violates the definition of a busy window in which the response time of $\hat{n}$ must be influenced by at least one event released earlier. ■

Theorem 2. *An upper bound for the response time of the n-th activation in a busy window can be obtained by:*

$$R(n) = B(n) - \delta(n) \quad (5)$$

Proof: $B(n)$ is by definition an upper bound on the processing time for n events, and $\delta(n)$ is by definition a lower bound for the arrival time of n events. Thus we can conclude that the difference is an upper bound. ■

The worst-case response time is the largest among all:

$$R^{+} = \max_{1 \le n \le \eta(w)} R(n) \quad (6)$$

IV. Response-Time Analysis of Independent Tasks

We now derive the worst-case response time for any independent task τ_i which is not part of a fork-join task. Contrary to systems which solely consist of independent tasks, an independent task τ_i can be preempted not only by independent tasks but also by fork-join tasks Γ_j. The following analysis includes these timing effects. To illustrate the effects caused by a fork-join tasks on an individual task we use the example task set as shown in Figure 3. In the example we have six independent tasks as well as one fork-join task mapped to two cores (Core 1, Core 2). The fork-join task consists of three stages and two segments. We now demonstrate how to obtain the response time of task τ_3 running on Core 1. We chose this task as an example, since it has a lower priority than all subtasks of Γ_2 as well as τ_1.

Given that we know Q and B, the worst-case response time can be derived. We now establish formulas to retrieve those functions for independent tasks. The multiple event busy time as well as the multiple event queuing delay can be computed by the following recurrence relations:

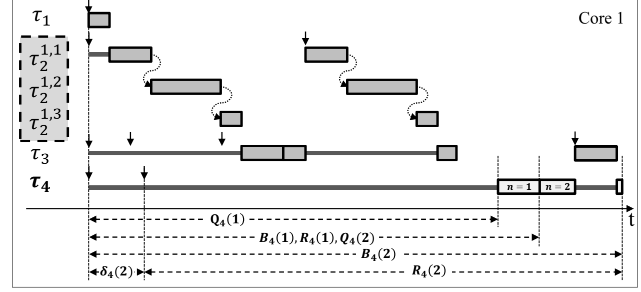

Figure 4. Worst-case schedule for independent task τ_4 running on core 1. Queuing delays as well as multiple-event busy times are indicated for the first two activations.

$$B_i(n) = n \cdot C_i + I_{i,IND}(B_i(n)) + I_{i,FJ}(B_i(n)) \quad (7)$$

$$Q_i(n) = (n-1) \cdot C_i + I_{i,IND}(Q_i(n)) + I_{i,FJ}(Q_i(n)) \quad (8)$$

These formulas are analogous to related work but include additional fork-join interference. Here $n \cdot C_i$ is the processing time required to execute n activations of task τ_i. The interference $I_{i,IND}(\Delta t)$ is an upper bound for workload caused by higher priority independent tasks in any time window of length Δt. Similarly, $I_{i,FJ}(\Delta t)$ denotes the interference caused by fork-join (sub-) tasks of higher priority mapped to the same core.

The higher priority interference of independent tasks can be classically computed [16] by considering all tasks that are of higher or equal priority (denoted by hp_{ind}).

$$I_{i,IND(\Delta t)} = \sum_{\forall \tau_j \in hp_{ind}(i)} \eta_j(\Delta t) \cdot C_j \quad (9)$$

To compute the interference caused by fork-join tasks, it is necessary to derive the event model at the input of subtasks.

Theorem 3. *The number of events that arrive in some busy window of length Δt at the input of a subtask $\tau_i^{\sigma,s}$ can be conservatively approximated by the input event model η_i of the corresponding fork-join task Γ_i assuming no events are queued at the input of the fork-join task:*

Proof: The proof is by contradiction. Assume a sub event model η' larger than η_i. This implies it is possible to observe more events at a subtask inside the fork-join task than events arrive before the fork-join task. Since only one event at a time can enter the fork-join task, and the number of events must be preserved, the hypothesis must be rejected. ■

This is intuitively shown in Figure 4, for each activation of Γ_2 each subtask is activated once in a cascading fashion. Analysis-wise, it is equivalent to assume that each stage is activated together with the entire fork-join task, rather than considering the end of the predecessor stage. This is because the actual analysis is agnostic of the actual release time of an interfering job as long as there is a case where the job contributes to the interference. With respect to the worst-case response time, stages are executed in a back-to-back fashion. If the first stage contributes to the inference,

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on September 28,2023 at 20:06:34 UTC from IEEE Xplore. Restrictions apply.

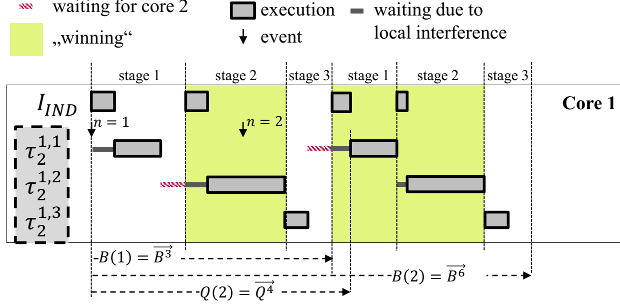

Figure 5. Effect of multiple events (i.e. n=2). Each event cascades through all three stages. Effects of waiting induced by core 2 is indicated as a red bar (but core 2 is not explicitly shown). Higher priority interference is denoted as I_{IND}. Queuing delay as well as the multiple event busy times are shown for the first two events. Note that $B(1)$ as well as $B(2)$ end at the end of the stage, since the completion times are delayed by core 2 (red bar).

then all other stages will contribute. Thus, for the sake of simplicity we assume that each stage is activated together with the entire fork-join task (i.e. the first stage). The effect, when activations of a fork-join task are already queued due to inter-core blocking of subtasks is not considered in this work. Such effects can be incorporated by the approach shown in [22].

Hence the interference by higher priority fork-join tasks is given by

$$I_{i,FJ}(\Delta t) = \sum_{\forall \tau_j^{\sigma,s} \in hp_{fj}(i)} \eta_j^{\sigma,s}(\Delta t) \cdot C_j^{\sigma,s} \tag{10}$$

$$= \sum_{\forall \tau_j^{\sigma,s} \in hp_{fj}(i)} \eta_j(\Delta t) \cdot C_j^{\sigma,s} \tag{11}$$

Here, hp_{fj} is the set of all higher priority fork-join subtasks which are mapped to the same core as the task under analysis, τ_i.

V. Response-Time of Fork-Join Tasks

Similar to the previously presented analysis of independent tasks, we also use the busy-window approach to derive the response time for fork-join tasks. However, previously introduced formulas cannot directly be applied to fork-join constructs. Mainly, because the behavior of the fork-join task depends on a complex interaction between multiple cores. That is, some segments in one stage finish earlier than others, inducing a "waiting time" in which one or more cores are potentially idle. This is the reason why we cannot directly apply the busy-window equations and also why we introduced the more generic busy-window definitions.

This section is organized in the following way: first, we discuss the response-time algorithm and in a second step we prove its conservatism. Generally, we use a greedy algorithm which iterates over stages. That is, first the finishing time of stage 1 is computed, than the finishing time of stage 2 and so on. The finishing time of the last stage is the response time.

Before we go through the analysis, we must highlight the relation between events and stages. An example is shown in Figure 5. The Gantt diagram shows the behavior on core 1, where I_{IND} denotes the interference. Hatched (red) bars denote where core 1 has to wait for core 2 to finish the previous stage, although the activity on core 2 is not explicitly shown in this figure. The first event $n=1$ as indicated by the arrow, arrives right at the start of the busy window. Obviously, this event ripples through all three stages. The second event ($n=2$) again cascades through all three stages, thus the first two events together execute a chain of six stages in total. We can conclude, that the behavior of two events is equivalent to the behavior of one event consisting of six stages. Without loss of generality, we model the behavior of multiple events by repeating the sequence of stages assuming the initial graph consists of s_{max} stages:

$$\tau^{\sigma,s} \equiv \tau^{\sigma,\hat{s}} \text{ with} \\ \hat{s} = s \pmod{s_{\max}} \tag{12}$$

Here, the $\equiv$ operator refers to all task parameters such as priority and execution time.

Now, similar to the multiple event busy time for independent tasks, we can define a stage-completion time for fork-join tasks.

Definition 6 (Stage-Completion Time)**.** *The stage-completion time $\overrightarrow{B}^s$ of fork-join task Γ is the largest time interval from the start of the busy window until all segments in the s-th stage have executed.*

Similarly, we can define the window from the start of a stage to the end of that stage by using the completion times.

Definition 7 (Stage-Completion Window)**.** *The stage-completion window $\overleftrightarrow{B}^s$ is defined as the relative time window associated with the completion of the s-th stage:*

$$\overleftrightarrow{B}^s = \overrightarrow{B}^s - \overrightarrow{B}^{s-1} \tag{13}$$

The stage-completion time can be used to formulate the multiple event busy time by evaluating $n \cdot s_{max}$ stages.

$$B(n) = \overrightarrow{B}^{n \cdot s_{max}} \tag{14}$$

Now we investigate how to obtain the stage-completion times of the example fork-join task Γ_2 as used in the previous section. Figure 6 shows the corresponding Gantt diagram. Naturally, all segments in the first stage of Γ_2 can start executing independently. In the worst-case scenario both segments are preempted by the worst-case interference (i.e. τ_1 and τ_5).

As it turns out, in this particular example the segment which executes on core 2 takes longer and gives the stage-completion time B^1. The light blue event on core 1 is a *non-critical* event, as it has no influence on B^1 whether a non-critical event actually arrives in B^1 or not. Respectively, we say that activations on core 2 were *critical* events since they *contributed* to the worst-case for that stage.

We say core 2 has "won" the first stage because it contributes to the worst-case behavior of that stage and we say that core 1 has "lost" that stage because it did not contribute.

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on September 28,2023 at 20:06:34 UTC from IEEE Xplore. Restrictions apply.

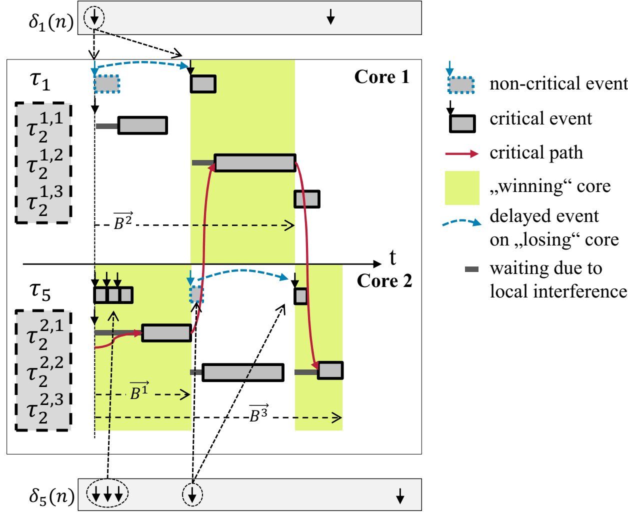

Figure 6. Illustrative example of the worst-case scheduling behavior of the first Γ_2 event. Stages are executed one after each other, once the previous stage has finished. The input event model δ is given for tasks τ_1 and τ_5 above and below the Gantt diagram. Stage-completion times are shown for the first event of Γ_2, $n{=}1$.

Obviously, in every stage there is only one segment which "wins". If two segments show equally large stage-completion times a winner can be chosen, all events of the "losing" stage are guaranteed to be considered in the following stages, as explained later. In that sense, events will be accounted in a different order. By choosing a different winner the stage completion time of following stages cannot be increased.

Thus a *critical path* spans across segments over time (cf. red arrow Figure 6). Note that there might exist *multiple* critical paths of the same length. However since they are all of the same total length it is sufficient to identify one.

Definition 8 (Critical Path). *The critical path function $\rho^{\sigma,s}$ is* 1 *if σ contributes to the completion time in the s-th stage and* 0 *if it does not.*

The critical path can be computed by evaluating the which segment maximizes the stage completion.

Definition 9 (Stage-Completion Window Candidate). *The stage-completion window candidate $\overleftrightarrow{B}_i^{\sigma,s}$ is the time interval from the start of the s-th stage of fork-join task Γ_i until the stage is completed on core σ, given previous stage-completion times $\overrightarrow{B}^{\sigma,s'}, s'{<}s$ were maximized*

At this point we can also formalize the concept of "winning" a stage and thus the critical path function. A core "wins" a stage if the stage-completion window candidate is the largest among all others.

$$\rho^{\sigma,s} = \begin{cases} 1 & \text{if } \sigma{=}\arg\max_\sigma \left(\overleftrightarrow{B}_i^{\sigma,s}\right) \\ 0 & \text{else} \end{cases} \tag{15}$$

The stage-completion time can be computed from the completion time of the previous stage plus the largest stage-completion window candidate of the s-th stage, by computing a candidate for each core and choosing the largest.

$$\overrightarrow{B}_i^s = \overrightarrow{B}_i^{s-1} + \max_{\forall\sigma}\left(\overleftrightarrow{B}_i^{\sigma,s}\right) \tag{16}$$

Computing the stage-completion time of the first stage is straight forward, as seen in the previous example. However, for the second stage in Figure 6 there are two effects that need to be considered:

1) non-critical activations which did not contribute to previous stages can be delayed and fall in the next stage (blue arrow). Hence it is *not conservative* to assume that events arrive as early as possible if a previous stage was "lost".
2) previously accounted critical activations (e.g. the first three-event burst) do not need to be reconsidered in future stages (*"pay burst only once"*).

Thus, it is evident that the event model equations δ / η cannot be directly be applied to the second stage. Therefore, a transformation is required to find the worst-case event arrival under the fork-join model.

Definition 10 (Sub Event Model). *A sub event model $\eta^{\sigma,s}(\Delta t)$ of a task (or subtask) is an upper bound on the largest number of events observable in any interval Δt of stage s on core σ, given previous stage-completion times were maximized.*

Theorem 4. *The sub event model $\eta_j^{\sigma,s}$ of some task τ_j can be computed from the input event model η_j and the number of critical events used in previous stages by the following recurrence relation:*

$$\eta_j^{\sigma,s}(\Delta t) = \min\Bigg(\eta_j(\Delta t), \\ \eta_j(\overrightarrow{B}_i^{\sigma,s-1} + \Delta t) - \\ \sum_{\forall r\in\mathbb{N}, r<s} \rho^{\sigma,r}\cdot\eta_j^{\sigma,r}(\overleftrightarrow{B}_i^{\sigma,r})\Bigg) \tag{17}$$

Proof: The proof is by induction over stages. Obviously, for the first stage the derived sub event model $\eta^{\sigma,1}$ is conservative, as it will evaluate to $\eta(\Delta t)$ which is by definition always a safe bound for any interval of length Δt in any stage.

For the induction step, we consider some stage s. We know that from the start of the busy window until the end of Δt in the s-th stage a total time of $\overrightarrow{B}^{\sigma,s-1} + \Delta t$ is spent. In this time, there can be at most $\eta(\overrightarrow{B}^{\sigma,s-1} + \Delta t)$ events in total. Of these events, some were critical and have been accounted for in previous stages, $r{<}s$. By definition the number of activations accounted in previous stages r is bound by recursively applying $\eta^{\sigma,r}(\overleftrightarrow{B}^{\sigma,r})$ but only if σ "won" that stage (i.e. $\rho^{\sigma,r}{=}1$). We conclude, that the number of events left in the s-th stage is conservatively bounded by eq. 17. ■

Now that all key concepts are introduced we can compute the stage-window completion candidate.

Theorem 5. *The stage-completion window candidate $\overleftrightarrow{B}_i^{\sigma,s}$*

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on September 28,2023 at 20:06:34 UTC from IEEE Xplore. Restrictions apply.

of fork-join task Γ_i can be computed using the following recurrence relation:

$$\overleftrightarrow{B}_i^{\sigma,s} = C_i^{\sigma,s} + I_{i,IND}^{\sigma,s}(\overleftrightarrow{B}_i^{\sigma,s}) + I_{i,FJ}^{\sigma,s}(\overleftrightarrow{B}_i^{\sigma,s}) \tag{18}$$

Where $I_{i,IND}^{\sigma,s}$ is an upper bound on the interference caused by higher priority interference of independent tasks and $I_{i,FJ}^{\sigma,s}$ bounds the higher priority interference of other fork-join tasks. The interference can be computed by using the sub event model (eq. 17):

$$I_{i,IND}^{\sigma,s}(\Delta t) = \sum_{\forall \tau_j \in hp_{ind}(i)} \eta_j^{\sigma,s}(\Delta t) \cdot C_j \tag{19}$$

$$I_{i,FJ}^{\sigma,s}(\Delta t) = \sum_{\forall \tau_j^{\sigma,s} \in hp_{fj}(i)} \eta_j^{\sigma,s}(\Delta t) \cdot C_j^{\sigma,s} \tag{20}$$

Proof: Naturally, the subtask of execution time $C^{\sigma,s}$ must be executed plus all higher priority interference released during $\overleftrightarrow{B}_i^{\sigma,s}$. The interference of higher priority tasks (fork-join and independent) is maximized as the number of events in the s-th stage is maximized by the sub-event model. The rest is analogous to the proof as in [16]. ∎

Using the established formulas we can already compute the multiple event busy time, but not decide how many activations need to be considered. This is done using the busy window concept introduced in eq. 4. To obtain the busy window, we need to know the queueing delay for fork-join tasks which has not been introduced yet. The general concept is analogous to the multiple-event busy time, thus we discuss it briefly.

Definition 11 (Stage Queuing Time)**.** *The stage queueing time $\overrightarrow{Q}_i^s$ is the largest time interval from the start of the busy window until all segments of fork-join task Γ_i get service in the s-th stage.*

Definition 12 (Stage Queuing Window Candidate)**.** *The stage queuing window candidate $\overleftrightarrow{Q}_i^{\sigma,s}$ is the largest time interval in which subtask $\tau_i^{\sigma,s}$ is blocked in the s-th stage by higher priority task interference prior to its first admission.*

To evaluate the queuing delay of the n-th activation, we must check the largest stage queueing time of the first segment of that activation. Analogous to the completion formulas we can derive the stage queuing formulas:

$$Q_i(n) = \overrightarrow{Q}_i^{s_{max} \cdot (n-1)+1} \tag{21}$$

$$\overrightarrow{Q}_i^s = \overrightarrow{B}_i^{s-1} + \max_{\forall \sigma} \left(\overleftrightarrow{Q}_i^{\sigma,s} \right) \tag{22}$$

$$\overleftrightarrow{Q}_i^{\sigma,s} = I_{i,IND}^{\sigma,s}(\overleftrightarrow{Q}_i^{\sigma,s}) + I_{i,FJ}^{\sigma,s}(\overleftrightarrow{Q}_i^{\sigma,s}) \tag{23}$$

Using above formulas, the worst-case response time can be calculated accurately. However, we made an assumption: it is conservative to greedily maximize stage completion windows. It is not intuitive that such a construction yields the global worst-case. Thus, we will now show that such an assumption is valid. To prove this, we need to show the following: A smaller completion window can only lead to a smaller stage completion time of subsequent stages. By promoting a losing candidate to a winner, all subsequent stages completion times will be decreased.

Theorem 6. *Maximizing each stage completion window $\overleftrightarrow{B}^{\sigma,s}$ independently yields the largest stage completion time $\overrightarrow{B}^{\sigma,s'}$ for any subsequent stage $s' > s$.*

Proof: The proof is by contradiction. We assume a small stage completion window $\widehat{\overleftrightarrow{B}}^{\sigma,s}$ in which segment σ wins. Then the number of critical events of task τ_j in that stage is given by $n_c = \eta_j^{\sigma,s}(\widehat{\overleftrightarrow{B}}^{\sigma,s})$. Obviously n_c must be less than or equal to the number of events that would have been used by a greedy approach $\eta_j^{\sigma,s}(\overleftrightarrow{B}^{\sigma,s})$. We call that difference n_d. The stage completion time $\overrightarrow{B}$ for stages beyond s can only increase if the amount of interference I in future stages increases. Similarly, the interference can only increase if the number of events in subsequent stages increases. Equation 17 gives the number of events in a stage if previous stage completion windows were maximized. However, for stages which succeed s, we cannot use the given equation. We now must subtract fewer critical activations, since we deliberately omitted n_d. We can rewrite the sub-event model equation 17 and add n_d to one of the sub event models according to our hypothesis:

$$\begin{aligned} \eta^{\sigma,s'}(\Delta t) = \min(\eta(\Delta t), \\ n_d + \eta(\overrightarrow{B}^{\sigma,s'-1} + \Delta t) - \\ \sum_{\forall r \in \mathbb{N}, r < s'} \rho^{\sigma,r} \cdot \eta^{\sigma,r}(\overleftrightarrow{B}^{\sigma,r})) \end{aligned} \tag{24}$$

Thus, we see that subsequent stages will at most see n_d more events, if we chose a smaller stage completion window for a previous stage. The additional interference gained in subsequent stages can be bounded by the time we shortened any previous completion window. We can conclude that by shortening a stage completion window we cannot increase the total stage completion time by more than we cropped it. ∎

By using a similar reasoning we can show that promoting a losing core, can never lead to higher interference in the following stages.

VI. Experiments

In this section we first measure the performance characteristics of some applications taken from the MiBench benchmark suite [11] executed in the *Romain* framework. Then we show how to transform these sample applications into a fork-join model. After we obtained a realistic application model we use these models together with a synthetic task set consisting of tasks with random characteristics in order to show some effects caused by fixed-priority scheduling. Therefore, we have implemented the analysis approach using the pyCPA framework [7].

A. Romain

To show the applicability of our approach we evaluated the *Romain* framework with respect to timing. *Romain* is

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on September 28,2023 at 20:06:34 UTC from IEEE Xplore. Restrictions apply.

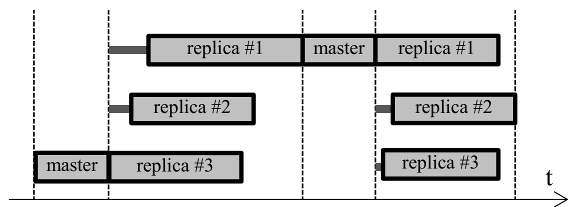

Figure 7. Replicated execution using Romain

a framework that provides software-implemented redundant multithreading [19] to user applications on top of the L4/Fiasco.OC [26] microkernel. We only give a brief overview here, as a detailed description of *Romain* was already published by Döbel et al. [8].

Romain spawns multiple replicas of an application, each of them running in a dedicated address space to facilitate fault isolation. Up to three replicas may be mapped to dedicated cores for optimal performance or they can be run on a smaller set of cores to save on resources. An additional master process maintains control of the replicas and intercepts all interactions between replicas and the outside world (e.g. system calls, page faults, CPU exceptions). The master compares the replicas' states and performs forward or backward recovery if necessary. The whole approach works, because the master process is in full control of all inputs that go into the replicas and can therefore ensure that they always execute deterministically between two state externalization events.

Looking at an application's execution as shown in Figure 7, we see a fork/join execution model: One replica acting as the master spawns the other replicas. Thereafter, the replicas execute user code concurrently. Once they reach a point where they externalize state, concurrent execution is interrupted and one of the replicas executes master code in sequential execution mode. After handling the event in master mode, control is returned to concurrent replica execution.

B. Experimental Setup and Evaluation

We show the applicability of our scheduling analysis by using results obtained from replicating a set of benchmarks from the MiBench benchmark suite [11] using *Romain*. To obtain reasonable timing models, we executed the benchmarks with one, two, and three replicas respectively on a state-of-the-art Intel Core i7 processor with 2.6 GHz [1]. We measured their execution times and specifically observed the time spent executing sequentially (state comparison, system call handling etc.), as well as the time spent executing concurrently (executing user code) in order to derive the execution times $C^{\sigma,s}$ as used in the analysis. Thus, first the number of stages is counted and the execution time spent in each stage is traced. The number of parallel segments is fixed to 2/3 due to the dual/tripple modular redundancy use case.

[1]Romain is only available for x86 architectures

Table I
NUMBER OF STAGES PER BENCHMARK

Benchmark	Stages	Total C [ms]
Security/Rijndael	14	4.9
Security/SHA	27	1.53
Automotive/Bitcount	8	271.2
Automotive/QSort	358	18.35
Networking/Dijkstra	233	9.05

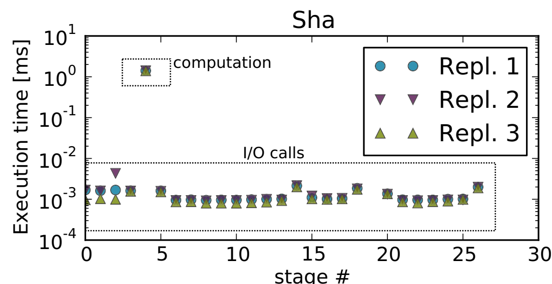

Figure 8. Execution times of segments in SHA benchmark. Most stages prepare I/O (e.g. printf).

In total we ran each benchmark 150 times to get a sufficiently large sample size. Then we removed the first samples which show major transient anomalies caused by memory layout organization. These where mostly page faults because the benchmarks did not lock pages, nor mark them as sticky in advance. These page faults only appear once at the beginning of the execution trace and vanish in the steady state as the memory is fully mapped. Also embedded processor without a MMU will not show such a behavior since the memory layout is fixed at design time. Since the benchmarks are executed on a live system, we see interference from interrupt handlers (i.e. timer) from time to time which tamper with the execution-time measurements. To filter these effects, we chose the execution times as the 0.9-quantile over the used samples per stage, assuming that rare outliers are caused by IRQ handlers.

We found that in most cases state comparison is in the order of a few μ seconds or less and compared to the computation-heavy segments of the benchmarks it is negligible small. Additionally, the system calls themselves may block due to hard drive access and other hardware interaction which is not an inherent part of the benchmark unless the operating system and hardware performance shall be evaluated which is not the case. Thus for the following experiments, we deliberately excluded time spent in system calls and only focus on usercode which is the intrinsic part of the benchmark.

C. Execution Time Measurements

An overview of the benchmark data such as the total execution time and the number of stages per benchmark can be found in Table I. We see, that the number of stages drastically differs from benchmark to benchmark ranging from 8 to 358 stages. This is mostly due to different I/O

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on September 28,2023 at 20:06:34 UTC from IEEE Xplore. Restrictions apply.

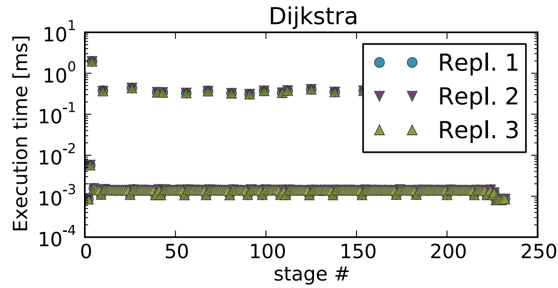

Figure 9. Execution times of segments in Dijkstra benchmark. I/O phases and computation bound phases alternate.

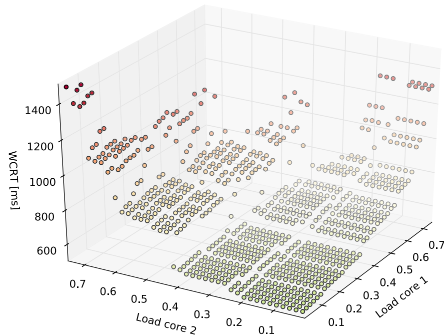

Figure 10. Rijndael: 14 stages executed on 2 cores. Lowest priority on both cores.

patterns of the benchmarks. In most cases I/O is caused by printfs used to print (intermediate) results. Figure 8 and 9 show representative execution time behavior and other benchmarks show a very similar pattern. The diagram shows the stage number on one axis and the execution time on the other. As we traced replicas independently the execution times is shown per replica.

D. Evaluating Romain Scenarios

As a replica resembles a segment in a fork-join task, we are able to model the Romain Framework as well as the benchmarks using the previously introduced fork-join task model. For the following experiment we use a dual core with the same clock frequency as the architecture used for the benchmarking (2.6 GHz).

In these experiments we evaluate the worst-case response time of the Bitcount and Rijndael benchmarks in various configurations. Therefore, we mapped 20 independent tasks as well as one fork-join application on both cores and varied the utilization in order see the effects on the response time. For the task parameters of the 20 independent tasks we use UUniFast algorithm [3].

First, we use the Rijndael benchmark (now referred to as Γ), assuming it has the lowest priority and is activated every 400ms. First, we map Γ on two cores (dual modular redundancy setup). The results are shown in Figure 10. The worst-case response time for the parallel case is fairly high,

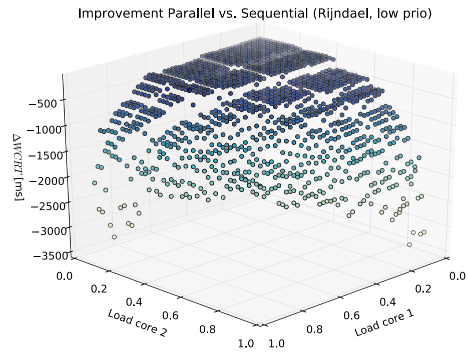

Figure 11. Rijndael: Comparison of worst-case response times for sequential mapping and a parallel mapping - low priority. Higher $\Delta WCRT \Rightarrow$ parallel mapping performs better.

considering that the execution time is in the order of 10ms.

After we made this observation, we compare the performance of the parallel setup with a purely sequential execution (Figure 11). In the sequential setup all segments are mapped to just one core (the one with the higher load). Such a mapping is called *redundancy in time* in case of replication. The graph shows the improvement of a parallel mapping over a sequential mapping.

Unexpectedly however is that the sequential mapping performs better in terms of response-time in all considered load configurations. In this setup, the stages will experience worst-case interference every second stage, thus the more contained sequential approach performs much better under heavy interference. Note, that this result is not caused by any overestimation from the analysis. That means it exists an actual event trace that is contained in the event model which causes such massive response times. In fact, the only scenario in which a parallel mapping performs better in terms of worst-case response time, is when Γ runs with highest priority.

We repeated the same experiment with the Bitcount benchmark (Figure 12) which has a significantly higher execution time but fewer stages. The results are similar but not as drastic. Response times for sequential and parallel mapping are in the same order of magnitude. Actually, if the priority is increased (priority level 3, instead of 10), the parallelized workload has a performance advantage in almost all load scenarios as shown in Figure 13. The experiments suggest that especially the number of stages and interfering tasks seems to have a high impact on the worst-case behavior and a parallelization is not reasonable in all cases.

VII. Conclusion

In this paper we have presented a worst-case response time analysis for fork-join tasks with arbitrary deadlines as well as independent tasks under the influence of fork-join tasks. We generalized the busy window approach by reducing it to a completion time and queuing delay problem. By applying the approach to the Romain framework, we were able to show that parallel workloads may behave

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on September 28,2023 at 20:06:34 UTC from IEEE Xplore. Restrictions apply.

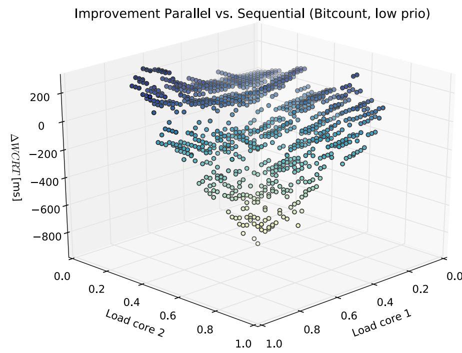

Figure 12. Bitcount: Comparison of a sequential mapping and a parallel mapping - low priority. Higher $\Delta WCRT \Rightarrow$ parallel mapping performs better.

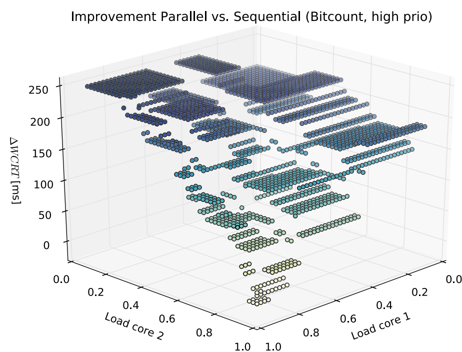

Figure 13. Bitcount: Comparison of a sequential mapping and a parallel mapping - high priority. Higher $\Delta WCRT$ means the parallel performs better.

counterintuitive: In some cases the worst-case response time is drastically larger compared to a sequentialized execution. This can be explained by the fact that a fork-join task experiences the worst-case interference of all cores in a combined fashion. This effect is not due to a conservative overestimation but is observable in real-world as long as the correctness of the used event models is guaranteed. We could show that a parallelization does not decrease the worst-case response time in all cases and is connected to subtle design decisions such as the number of stages and prioritization. The most important open question is which configurations impose the highest speedup and which setups should be avoided. That is, what is the optimal stage, segment and execution time as well as segment allocation trade-off for a given application with respect to its response time. Future work must also show how a partitioned fixed-priority scheme performs compared to Gang scheduling and hybrid policies.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of this paper.

References

[1] AUTOSAR GbR. Specification of Multi-Core OS Architecture v1.0.0. http://www.autosar.org/, November 2009.
[2] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese. A generalized parallel task model for recurrent real-time processes. pages 63–72, 2012.
[3] E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Syst.*, 30:129–154, 2005.
[4] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.
[5] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son. New strategies for assigning realtime tasks to multiprocessor systems. *IEEE TRANSACTIONS ON COMPUTERS*, 44(12):1429–1442, 1995.
[6] S. Collette, L. Cucu, and J. Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106(5):180–187, 2008.
[7] J. Diemer and P. Axer. pyCPA - a pragmatic Python implementation of Compositional Performance Analysis. http://code.google.com/p/pycpa.
[8] B. Döbel, H. Härtig, and M. Engel. Operating system support for redundant multithreading. In *Proc. of EMSOFT*, 2012.
[9] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu. A real-time scheduling service for parallel tasks. *In Proc. of RTAS*, 2012.
[10] N. Fisher, S. Baruah, and T. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *Proc. on ECRTS*, pages 10 pp. –127, 0-0 2006.
[11] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of WWC*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.
[12] M. Holenderski, R. Bril, and J. Lukkien. Parallel-task scheduling on multiple resources. In *Proc. of ECRTS*, pages 233–244. IEEE, 2012.
[13] Intel Corporation. *Intel® Threading Building Blocks*, October 2011.
[14] K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proc. of. RTSS*, pages 259 –268, 30 2010-dec. 3 2010.
[15] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *Proc. of. ECRTS*, pages 239–248, Washington, DC, USA, 2009.
[16] J. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. *Proc. 11th RTSS*, pages 201–209, Dec 1990.
[17] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *Proc. of ECRTS*, pages 321 –330, july 2012.
[18] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, 3.1 edition, July 2011.
[19] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. *SIGARCH Comput. Archit. News*, 28:25–36, May 2000.
[20] K. Richter. *Compositional scheduling analysis using standard event models*. PhD thesis, TU Braunschweig, 2005.
[21] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *Proc. of RTSS*, pages 217 –226, 29 2011-dec. 2 2011.
[22] S. Schliecker, M. Negrean, and R. Ernst. Response Time Analysis on Multicore ECUs with Shared Resources. *IEEE Transactions on Industrial Informatics*, 5(4):402–413, November 2009.
[23] S. Schliecker, M. Negrean, and R. Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Proc. of DATE*, pages 759 –764, 2010.
[24] S. Schliecker, J. Rox, M. Ivers, and R. Ernst. Providing accurate event models for the analysis of heterogeneous multiprocessor systems. In *Proc. of CODES-ISSS*, pages 185–190, October 2008.
[25] K. W. Tindell, A. Burns, and A. J. Wellings. An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. *Real-Time Systems*, 6(2):133–151, 1994.
[26] TU Dresden OS Group. L4/Fiasco.OC microkernel. http://www.tudos.org/fiasco, 2012.

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on September 28,2023 at 20:06:34 UTC from IEEE Xplore. Restrictions apply.