Compositional Path-Latency Computation using Local Busy Times

Simon Schliecker, Rolf Ernst
Institute of Computer and Communication Network Engineering
Technical University of Braunschweig
[schliecker | ernst] @ ida.ing.tu-bs.de

Abstract

This report proposes a new method for the calculation of end-to-end latencies of applications that involve the processing on multiple components in a heterogeneous multiprocessor system with real-time constraints. For this, we decompose the calculation into (i) a local part for each resource that yields the busy time for multiple coinciding processing requests and (ii) the composition of these results to receive the end-to-end latency. Our procedure precisely captures the parallel processing of multiple events along the path, thus avoiding the pay-bursts-only-once problem. Besides its accuracy, it is also very fast and can be provided for a large domain of scheduling policies and architectures.

1 Introduction

Formal performance verification is essential to safely verify the compliance of current multi-processor systems with real-time constraints. These can be in the shape of latency constraints in distributed automotive networks throughput constraints in multimedia multiprocessors. With increasing system complexity and continued functional integration, system level performance issues are becoming more complex and the analysis must often be adapted for different systems. To counter this challenge, compositional approaches have been proposed that break down the complexity into the analysis of components (i.e. processors, busses, or memories) and a description of events generated between theses components (so called event streams).

A key metric for the overall performance of multiprocessor systems, is the latency between the arrival of new data or interrupts (which we generally denote as events) and the finishing of its processing and finally the reaction. As this can involve processing on a set of different specialized hardware or software resources, a system-level approach to derive the end-to-end latency must be able to consider heterogeneously scheduled subsystems.

In this paper, we propose an algorithm to accurately compute the end-to-end latency in

multiprocessor real-time systems. This improves and extends the work in [6], by greatly increasing the overall analysis precision especially for systems under dynamic load. We keep the versatility of the approach, allowing functional and non-functional cyclic task dependencies and a multitude of heterogeneous scheduling policies.

The remainder of this paper is structured as follows:

- First, we present and evaluate the work related to our approach in Sec. 2
- We formally introduce the *multiple event busy time function* to model resource timing, which is a side-product of many known single processor analyses (Sec. 3.1).
- We present a method to derive end-to-end latencies via multiple tasks on multiple processors that allows arbitrary event models and considers event pipelining (Sec. 4). We incorporate the findings into the approach of [6], yielding an accurate and versatile multiprocessor performance analysis.
- We conclude the paper with experiments (Sec. 5) and our conclusion in Sec. 6.

2 Related Work

The performance analysis problem is addressed by various compositional approaches that separate the problem into local component analyses and the modeling of event traffic between them. In Network Calculus [9] and the Real-Time Calculus [3] based on it the local resource behavior is modeled as the execution time provided to the processing of events of a certain stream within a time window of given size Δt. Such a resource curve is depicted in Figure 1a, for minimum ($\beta^-(\Delta t)$) and maximum supplied service ($\beta^+(\Delta t)$). The approach derives output event models and remaining resource capacity by folding operations in continuous time domain.

Appropriate service curves have been provided e.g. for static priority preemptive scheduling, EDF, TDMA, and others [3]. But specific resource service curves may be difficult to derive when e.g. a preemption count is required, as for the calculation of context switch overhead or cache related preemption delay. The individual component analyses can also be computationally intensive due to the continuous time model in which both the provided resource service and requested task execution are expressed. For this reason practical simplifications have been suggested (e.g. stepwise evaluation [4], finite models of event streams [24]).

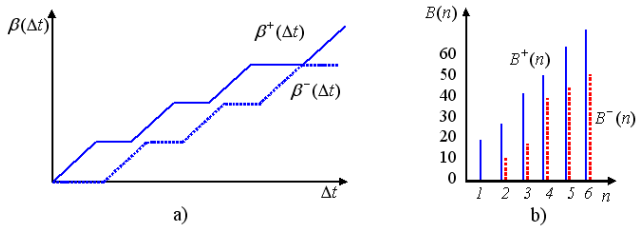

Figure 1. Models of Resource Service.

The opportunities of relying on simpler event and resource models have been explored in [6] which we present in more detail in Section 2.1. Here, the basic metric to model the real-time performance of the components are the tasks' worst (and best) case response times. This simple metric has been the focus of numerous research in single processor scheduling theory such as [8][2][22]. A common procedure for its derivation is symbolic simulation of a critical instant scenario. Various extensions have been proposed to improve the analysis results (such as offsets [12] or variable task execution times [11]) and consider realistic scheduling behavior (e.g. FlexRay protocol [15], cache related preemption delay [20]).

A different approach to multiprocessor analysis is chosen in the holistic approaches of [23][5][14] where the classical single-processor scheduling theory is systematically extended and can be tailored toward a specific combination of input event model, resource sharing and communication policy. The global view on the system allows to take global correlations into account. However, in the case of a large number of such dependencies, the complexity of the analysis grows with system size and heterogeneity. In practice, deterministic networks such as TDMA are therefore highly useful to simplify the analysis procedure. Our analysis will not rely on such a holistic view, but rather perform a hierarchical analysis, extracting all relevant information from local resources before composing them on the system level.

All of the above approaches bring a method to compute the end-to-end latency of events that are processed by sequential tasks on multiple resources. The simplest way to conservatively determine the latency is by accumulating the local worst-case response times as is done in e.g. [21] and [6]. However, this procedure is inaccurate in the case of bursty event occurrence owing to the 'pay-burst-only-once" problem. A burst of events can in general occur at the input of any task along a path — leading to large local worst case response times — but the same event processed along the path can not experience this delay at each task.

Better estimates can be achieved through the convolution of component behavior along the path as is done in [9][3]. However, these methods rely on the concept of continuous time service curves. In general the folding operations can therefore be computationally intense. The proposition of Section 4 is to perform similar operations in discrete time domain using the multiple event busy time. This naturally limits the computed values to the critical candidates.

In [7] an efficient method to compose the latency of pipeline stages is presented. The key idea is the derivation of a substitute single processor system which is then investigated for schedulability. However, it is specifically aimed at homogeneously scheduled systems, and does not allow for any cyclic functional or non-functional dependencies.

2.1 Compositional Performance Analysis Loop

In the multiprocessor performance analysis of [6] the analysis of individual components is interleaved with the propagation of event stream information. This procedure is repeated until a fix-point representing conservative estimates of the event traffic anywhere in the system is found. This framework is shown in Figure 2. A description of the analysis procedure follows.

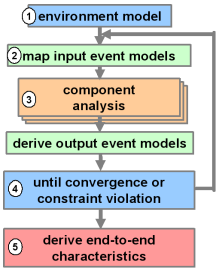

Figure 2. Compositional Analysis Loop.

1. Specification of Environmental Model. As a characterization of the system's environment, all environmental input event models are supplied. Event models represent the minimum and maximum amount of events (see Sec. 2.2). All other input event models within the system are initialized with optimistic guesses, which are iteratively refined during the analysis procedure.

2. Distribution of Input Event Models. The best known (estimated) event models for incoming traffic are supplied to each component.

3. Component Analysis. Based on the input event models each component analysis derives for each task mapped to the component the local response times and output event models on the basis of single processor scheduling theory.

4. Check for convergence. These refined output event models are compared to the previous input event model assumptions (of step 2). If all are the same, the analysis has converged. Otherwise, the according component analysis has to be redone with the refined inputs.

This procedure is monotonic, as long as the event models become increasingly more general with each iteration, and thus each iteration subsumes the previous assumptions [17]. The analysis is complete if either all event streams converge toward a fix-point, or if an abort condition, e.g. the violation of a timing constraint has been reached.

5. Derive System Properties. Once the analysis has converged, the conservative event models can be used to derive local worst case response times, output event traffic, and end-to-end latencies through the complete system.

This procedure has been fully implemented on the basis of standard event models [6] and simple propagation mechanisms. The contributions of this paper significantly extend both the applicability and accuracy of this framework.

2.2 Event Models

A key element of compositional performance analysis is expressing the traffic flow between different components with the help of *event models*. The compositional approach of [6] relies on simple *standard event models*. These are appropriate for many common real-time setups (e.g. in automotive) but inaccurately represent more complex event patterns. These may occur at the inputs to the system but also within a system where the sequential processing on different resources can significantly distort the event traffic [13]. Packetization and layered communication can aggravate this problem (as investigated in [18]). Finally, in the domain

of systems-on-chip the modeling of accesses to a shared memory is of increasing importance [19][1]. Such accesses can be very irregular and their accurate treatment is key to timing validation in MpSoCs.

In [3] and [6] event models describe the maximum and minimum number of events η that may occur during a time interval of given size Δt. Figure 3 shows such an event model representation on the left.

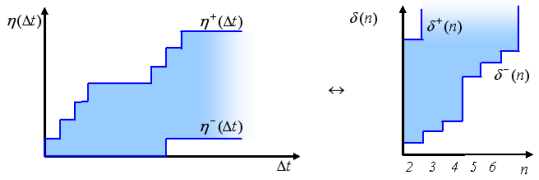

Figure 3. Event Model Representation.

An event model can also be expressed by the distances of the contained events. This is shown on the right side of Figure 3. The functions $\delta^-(n)$ and $\delta^+(n)$ represent the minimum and maximum distance between the occurrence of any n events in the stream. The δ functions are therefore sensefully defined only for $n \geq 2$. Both the δ and the η representation can be converted to each other, such that the $\delta^-(n)$ function can be represented by the $\eta^+(\Delta t)$ function and vice-versa (the same is true for $\delta^+(n)$ and $\eta^-(\Delta t)$). The conversion can be done as follows:

$$\delta^-(n) = \inf_{0 \leq \Delta t, \Delta t \in \mathbb{R}} \{\Delta t \mid \eta^+(\Delta t) = n\} \quad (1)$$

$$\eta^+(\Delta t) = \max_{2 \leq n, n \in \mathbb{N}} [\{n \mid \delta(n) < \Delta t\} \cup \{1\}] \quad (2)$$

In this paper we will mainly utilize the δ representations. This representation has the strong advantage of being a discrete function of n, rather than the η representations, which are continuous. This allows us to conveniently investigate a discrete set of events in our formulas, rather than operating with continuous functions, in which only the steps contain true information.

Correct δ functions have some fundamental properties (such as superadditivity [9]), but no rules are imposed on how they are actually represented. For a compact description the *standard event models* in [17] rely on the three parameters event stream period $\mathcal{P}$, event stream jitter $\mathcal{J}$, and minimum distance between any two events d^{min}. The δ-function of e.g. a bursty event stream can then be expressed as follows:

$$n \geq 2 : \delta^+(n) = (n-1)\mathcal{P} + \mathcal{J} \quad (3)$$

$$\delta^-(n) = \max((n-1)d^{min}, (n-1)\mathcal{P} - \mathcal{J}) \quad (4)$$

3 Multiple Event Busy Time Model

In this section we introduce our local abstraction metric that captures the timing behavior of an individual component, which in general is a task mapped to a processor. We are not

interested in its detailed functionality. What is required is the reaction and processing time to an incoming event, as well as the effect of multiple events that coincide, leading to preemptions and execution backlog. The *multiple event busy time* captures these effects, and will be defined independent of the utilized scheduling policy.

Notation We first introduce our basic application model and notation. *Tasks* represent a sequence of operations with known minimum and maximum execution time. A task is activated by an *event*, and produces an event when the execution is finished. Tasks are mapped to resources that arbitrate between the tasks mapped to it according to their scheduling policy, causing task activations to possibly interrupt each other.

All events are numbered according to the sequence of their occurrence — events occurring later receive higher numbers. We will focus on an arbitrary event 0, thus preceding events will have negative numbers. The arrival time of a event n at the resource to which task i is mapped is denoted with $a_i(n)$. The time at which the resulting task activation produces an event is denoted by $e_i(n)$. Tasks process the events of an event stream in-order. This is a typical assumption in scheduling theory matching the design practice. Prioritized events are modeled with separate event streams.

The events that belong to the same event stream are constrained by the corresponding *event model*. The event model represents the minimum and maximum distances between the events, denoted with δ_i^- and δ_i^+, where i is the task activated by the event stream. For example, the minimum and maximum arrival times $a_i(n)$ and $a_i(m)$ of two events n and m with n being an event before m ($n < m$) are constrained as follows:

$$\begin{aligned} a_i(m) &\geq a_i(n) - \delta_i^-(m-n+1) \quad \wedge \\ a_i(m) &\leq a_i(n) - \delta_i^+(m-n+1) \quad \forall n, m \in \mathbb{N}, n < m \end{aligned} \tag{5}$$

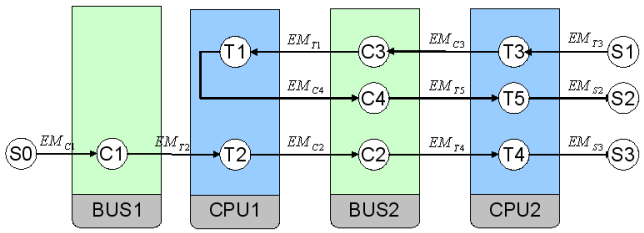

Figure 4. System Example.

The processing of an external event can involve a chain of task activations. In this case, the *output event* of one task becomes the *input event* of another, an thus $e_{i-1}(n) = a_i(n)$. For example in Figure 4, the output event stream of task $T3$ becomes the input event stream of task $C3$. Events in different streams may arbitrarily share the given resources.

3.1 Multiple Event Busy Time Model

The analysis in Section 2.1 composes the timing of multiple tasks on multiple processors to receive the system level timing. For this, the behavior of each task is abstracted by its

local worst (and best) case response times with classical scheduling methods. A side-product of response time analyses based on symbolic simulation is the *busy period*. For example, in static priority preemptive scheduling it denotes the maximum time during which the resource is not idle during a critical instant scenario.

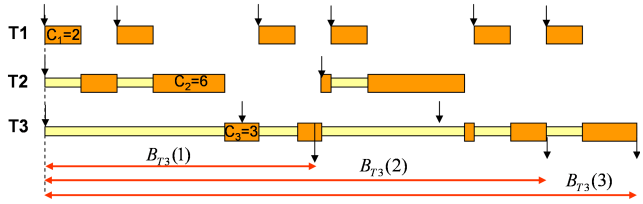

Figure 5. Multiple Event Busy Times.

We build on this concept by introducing the *multiple event busy time* function. The multiple event busy time function represents the amount of time necessary to process a certain number of events that arrive within the same busy window. For example, $B^+(1)$ is the maximum busy window inflicted by a single event that arrives after the previous was finished. $B^+(2)$ is the maximum busy window size that is spanned by two events, where the second arrives before the first is finished. The minimum busy time B^- can be defined correspondingly.

Consider the example in Figure 5. The first activation of task T3 experiences a critical instant scenario for static priority preemptive scheduling: all higher priority tasks are activated at the same time and as early as possible thereafter. This leads to a worst case busy time of $B^+_{T3}(1) = 15$, which is the sum of the involved core execution times. The next activation is processed subsequently, and finished no later than $B^+_{T3}(2) = 22$. The series of such derived busy times can be plotted, as depicted in Figure 1b. Note that in this example it is completely irrelevant for $B^+_{T3}(2)$ *when* the second activation arrives, as long as it arrives before the preceding is finished. This may not be true for all schedulers, but will be enforced by the following formal definition of the multiple event busy time.

Definition 1 (Multiple Event Busy Time)**.** *The n-event busy time $B^+_i(n)$ ($B^-_i(n)$) of a task i is given by the maximum (minimum) time it may take i to process n events, if all but the first of the n events arrive before the preceding is finished.*

By this definition, the busy time contains all effects that can delay the finishing of the task activations. In particular, this includes the interference by other tasks mapped to the same resource and the scheduler's decisions of the execution order. If present, inter-task communication, context switch overhead and other delays must be considered as well. The definition does not imply that the resource is never idle during the processing of the events, but this will usually be the case in work conserving schedulers.

This busy time has been implicitly used in many previous scheduling analyses that rely on the windowing technique such as [8][22][10][16]. During the calculation of the worst case response time, finishing times of different task activations are calculated — in a worst case scenario this corresponds to busy times as defined above. Thus, in this case the calculation of the busy times comes at no additional computational costs.

Correlations between input event streams ("offsets") can also be considered for tight estimates. For this the different offset scenarios presented in e.g. [12] need to be checked.

Only the steps in the service curves of [3] contain relevant information, the multiple event busy time allows an equally precise modeling of the resource timing.

To demonstrate feasibility of the idea, Theorem 1 derives the multiple event busy time for static priority preemptive scheduling of independent tasks without preemption costs[1] (on the basis of [22]).

Theorem 1. *The multiple event busy time $B_i^+(n)$ for a task i under static priority preemptive scheduling with independent tasks is given by*

$$B_i^+(n) = n \cdot C_i + \sum_{j \in hp(i)} (\eta_j^+(B_i^+(n)) \cdot C_j \tag{6}$$

where

C_i, C_j *is the maximum core execution time of task i, j.*

$hp(i)$ *is the set of tasks with higher priority than i.*

$\eta_i^+(\Delta t)$ *is the maximum number of events that lead to an activation of task i in a time window of size Δt.*

This equation can be solved iteratively.

Proof. The n coinciding activations of task i are finished when their combined workload has been processed (first term $n \cdot C_i$). The processing of this workload in the given scheduler can be delayed only by higher priority task activations. During the processing of the n events, $B_i^+(n)$, the maximum number of task activations is given by $\sum_{j \in hp(i)} (\eta_j^+(B_i^+(n))$ and their combined workload can not be larger than the second term. Shifting the task activations can not lead to a larger combined work load during $B_i^+(n)$.

As $B_i^+(n)$ is used on both sides of Equation 6 no direct solution is available. However, the right hand side is monotonic with respect to $B_i^+(n)$, and thus the fixpoint can be found through iteration. The fixpoint is then a valid solution, and thus the previous reasoning is correct. □

Note that as opposed to the classical worst case response time equations (such as [2]) the calculated busy time does *not depend* on the actual event model of the investigated task i. The algorithms in the following sections will explicitly reintroduce this event model in the form of the amount of events n.

4 Path Latency

A common problem in real-time systems is that multiple tasks on the same or different processors are subsequently involved in the processing of an event. For example, multiple controllers can be involved in a sensor-actor chain. But also streaming applications have

[1] In our tool framework, we have relieved these limitations.

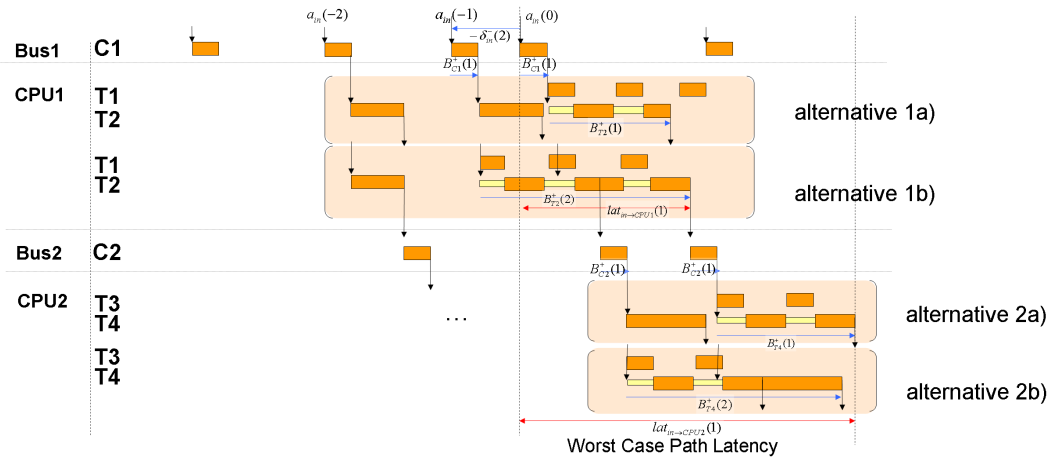

Figure 6. Example For Path Latency. On CPU2, the latest production time of event 0 by task T2 is given if the interference by T1 is aligned with the arrival of event -1 ($lat_{in->T2} = 0 - \delta_{in}(2) + B^+_{C1}(1) + B^+_{T2}(2)$). On CPU2 the worst case occurs if the interference of T3 is correctly aligned with the arrival of event 0 ($lat_{in->T4} = 0 - \delta_{in}(2) + B^+_{C1}(1) + B^+_{T2}(2) + B^+_{C2}(1) + B^+_{T4}(1)$).

throughput constraints that are determined by the sequential processing on multiple resources. Such a processing chain opens opportunities to use specialized components, and benefit from increased throughput through event pipelining. However, these benefits can only be exploited in a real-time system, if an accurate analysis is available that accurately captures the behavior.

The classical approach to derive the end-to-end latency [6][21] has been to accumulate the individual task worst case response times along the path. The simple summation is a conservative estimate of the end-to-end latency. However, it can also lead to a large overestimation in the case of bursty event arrival: If a burst enters the system this translates into large local worst case response times, as an event may have to wait for previous events to be finished. Usually, such a burst can occur anywhere along the considered path, and thus all worst case response times are relatively large. In reality however, an event that has been delayed by its predecessors on one resource can not be fully delayed by the predecessors again on the successive resource. During its waiting time the preceding events have continued to be processed on the successive resources.

Note that the calculated worst case response times and traffic estimates may be correct and conservative, only can they not be experienced by the same event traversing a path. The rationale behind the improved path latency analysis is explained in the following example.

Consider the example in Figure 6. Periodic messages with a slight jitter are transmitted via $Bus1$ to $CPU1$ where they lead to the activation of a low priority task T2. Two scenarios are now possible that may lead to a worst case latency of an arbitrary event 0:

1a) The interference by a higher priority task T1 is aligned with the arrival of event 0, and thus the corresponding activation will experience the worst case busy time $B^+_{T2}(1)$.

1b) The interference by T1 is aligned with the arrival of event -1, and thus the corresponding activation is delayed by the unfinished previous activation. In this case both events are finished $B^+_{T2}(1)$ after the arrival of the previous event

Task activations further in the past may not interfere in this example due to a sufficient distance between the activating events. Scenario 1b) produces a later production time of event 0 at the output of $CPU1$.

A different scenario determines the worst case on $CPU2$, when the interference of task T3 is aligned with the arrival of event 0 (scenario 2a). Aligning the interference with the arrival of the previous event (2b) can not lead to a larger production time of event 0 at the output of T4. Thus in this example, the worst case path latency is given by

$$lat(1) = 0 - \delta^-_{in}(2) + B^+_{C1}(1) + B^+_{T2}(2) + B^+_{C2}(1) + B^+_{T4}(1) \tag{7}$$

As a generalization of the worst case path latency, we will also consider the worst case delay that can be experienced by a number of events. The classical latency is then a special case of the n-event latency for $n = 1$.

Definition 2. *The n-event end-to-end latency of path $\mathbb{P}$ is defined as the maximum distance between the arrival of an event at the input the first task in of $\mathbb{P}$, and the production of the n-th causally dependant event at the output of the last task out of $\mathbb{P}$. An output event is causally dependent on an input event when the output event is produced by the same activation that has consumed the input event.*

$$lat(n) = (e_{out}(n-1) - e_{in}(0))^{\max}$$

We will now derive an improved maximum value for $lat(n)$ by taking into account the inherent overlapping of event processing as well as the inherent correlations sketched above that constrain the impact of multiple worst cases along the path.

Because the events arriving at the first task *in* of the path belong to the same event stream, their arrival times are correlated, so that their minimum (and maximum) distances are given by the comprising event model. Thus, if the arrival time of one event is known, all preceding and successive events have a certain minimum and maximum distance from this event according to Equation 6. Therefore, w.l.o.g. we can rearrange the time scale so that $e_{in}(0) = 0$. $lat(n)$ is then maximal, if $e_{out}(n-1)$ is maximal (with respect to event 0's arrival). For the following lemma, let $e_{i-1}(n)$ be the time event n arrives at task i and $e_i(n)$ the time the corresponding activation is finished and the output event produced.

Lemma 1. *The maximal exit time $e_i(n)$ of any event n on task i is bounded by*

$$e_i(n) \leq \max_{k \geq 0}\{e_{i-1}(n-k) + B^+_i(k+1)\}$$

where

- *$e_{i-1}(n-k)$ is the arrival time of the k-th event before event n.*
- *$B^+_i(k+1)$ is the maximum busy time for $k+1$ events to be processed by task i.*

Proof. The output event n is produced at time $e_i(n)$ by the activation of task i that has consumed the input event n at time $e_{i-1}(n)$. There are two cases: Either the previous activation of task i is finished when the input event arrives, or it is not.
Case 1: $e_{i-1}(n) > e_i(n-1)$ (previous activation finished). In this case, following the definition of the busy time, the activation n is finished no later than $e_{i-1}(n) + B_i(1)$.
Case 2: $e_{i-1}(n) < e_i(n-1)$ (previous activation not finished). The input event n arrives while at least one previous event has not been produced. Let k be the number of events that have not been processed, so that all events that have arrived $k+1$ or before have been processed. Thus, $e_i(n-k-1) < e_{i-1}(n) \leq e_i(n-k)$. (We assume that an event arriving at the instant at which a previous is finished does not fall into its busy time.)

In this case, the multiple event busy time bounds the time at which the busy interval started by event $n-k$, and which event n now contributes to is over. It is given by $e_i(n) < e_{i-1}(n-k) + B(k)$. As k is unknown we have to take the maximum over all possible values:

$$e_i(n) \leq \max_{k \geq 0}\{e_{i-1}(n-k) + B_i^+(k+1)\}$$ □

To derive the end-to-end latency from Lemma 1, $e_{out}(0)$ can recursively calculate based on timing of input events at the same and preceding tasks. In practice only a small set of k's needs to be checked, as events sufficiently in the past will not influence the latency of event 0. This makes the algorithm appropriate even for very long paths.

The minimum end-to-end latency can be calculated with a similar method as described above. However, for many realistic cases it is simply the sum of the best case response times — it is therefore not explored further in this paper.

5 Experiments

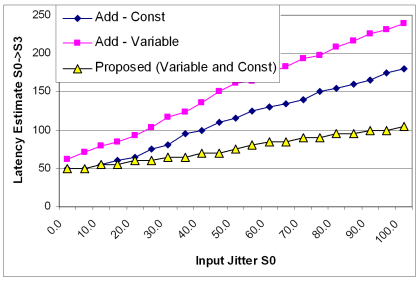

Figure 7. Comparison of Path Latencies

We have conducted a set of experiments to show the validity and precision of the presented approach. Consider the example of Figure 4 in which events are processed along the path from S0 to S3. They can be disturbed by higher priority events from another application. Two scenarios are sketched in Figure 7: In one scenario, all execution times are constantly 5 ("Const"), in the other scenario all execution times are variable between 1 and 5 ("Variable"). This causes a larger dynamism of event timing. The latency calculation that is based on the accumulation of local worst cases ("Add") draws the expected overestimation from the pay-bursts-only-once problem. This problem becomes worse, the larger

the timing uncertainty becomes (i.e. increasing jitter, or variable execution times). The proposed analysis ("Pipelined") tackles this problem through correlating the local worst case busy times. It is also insensitive to the increase of dynamism, and correctly calculates identical latencies for both setups.

Further experiments of small systems have been performed in the scope of the comparative paper [13], in which real-time calculus [3], SymTA/S [6], and other approaches were compared. The "pipelined" path latency of this paper was included in the experiments. In one experiment a short chain of 3 tasks on 2 CPUs is investigated. A particular challenge to the analysis is the correlation of events activating T3 and those activating T1. It can be seen in Figure 8 that the new path latency calculation "SymTA pipelined" is better than the simple additive calculation "SymTA add". Of all approaches under comparison, our approach is closest to actual worst cases derived with model checking ("Upaal"), often matching it accurately. In the given experiment, this is owed to the fact that the analysis on CPU1 better considers the offsets between the activations of T2 and T3 according to e.g. [12]. [13] also contains an evaluation of execution times — besides its accuracy, our analysis is also very fast.

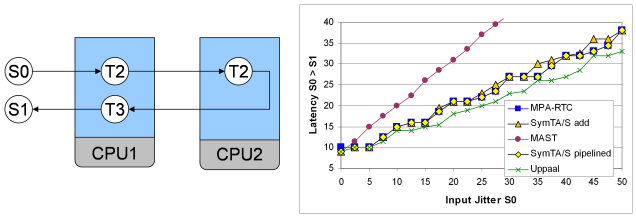

Figure 8. Example System Setup and Comparison of end-to-end latency (from [13]).

6 Conclusion

In this paper we have proposed an efficient methodology to derive path latencies in a multiprocessor system. The resource behavior is modeled using the discrete multiple event busy time function, which can be derived on the basis of classical response time analysis. Through this abstraction our method is suitable to consider arbitrary input event models in large variety of different heterogeneous scheduling policies. The approach considers pipelining and transient overload effects that surface along the path.

References

[1] K. Albers, F. Bodmann, and F. Slomka. Hierarchical Event Streams and Event Dependency Graphs: A New Computational Model for Embedded Real-Time Systems. *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 97–106, 2006.

[2] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[3] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. *Proc. 6th Design, Automation and Test in Europe (DATE)*, pages 190–195, 2003.

[4] S. Chakraborty and L. Thiele. A New Task Model for Streaming Applications and Its Schedulability Analysis. *Proceedings of the Design, Automation and Test in Europe (DATE'05) Volume 1-Volume 01*, pages 486–491, 2005.

[5] J. Gutiérrez, J. García, and M. Harbour. On the Schedulability Analysis for Distributed Hard Real-Time Systems. *Proceedings of the 9th Euromicro Workshop on Real-Time Systems, Toledo, Spain*, pages 136–143, 1997.

[6] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the symta/s approach. In *IEE Proceedings Computers and Digital Techniques*, 2005.

[7] P. Jayachandran and T. Abdelzaher. A Delay Composition Theorem for Real-Time Pipelines. *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, pages 29–38, 2007.

[8] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390, 1986.

[9] J. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001.

[10] C. Li, R. Bettati, and W. Zhao. Response time analysis for distributed real-time systems with bursty job arrivals. *Proceedings of IEEE ICPP*, 1998.

[11] A. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, 1997.

[12] J. Palencia and M. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS98)*, 1998.

[13] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. G. Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *ACM Conference on Embedded Software (EMSOFT)*, Salzburg, Austria, Oct. 2007. ACM Press.

[14] P. Pop, P. Eles, and Z. Peng. Schedulability analysis and optimization for the synthesis of multi-cluster distributed embedded systems. *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 184–189, 2003.

[15] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing Analysis of the FlexRay Communication Protocol. *Proceedings of 18th EuroMicro Conference on Real-Time Systems, Dresden*, pages 203–213, 2006.

[16] R. Racu, L. Li, R. Henia, A. Hamann, and R. Ernst. Improved Response Time Analysis of Tasks Scheduled under Preemptive Round Robin. *International Conference on Hardware Software Codesign and System Synthesis (CODES)*, 2007.

[17] K. Richter. *Compositional Scheduling Analysis Using Standard Event Models*. PhD thesis, Technical University of Braunschweig, 2004.

[18] J. Rox and R. Ernst. Construction and Deconstruction of Hierarchical Event Streams with Multiple Hierarchical Layers. *Real-Time and Embedded Technology and Applications Symposium*, 2008.

[19] S. Schliecker, M. Ivers, and R. Ernst. Memory Access Patterns for the Analysis of MPSoCs. *Circuits and Systems, 2006 IEEE North-East Workshop on*, pages 249–252, 2006.

[20] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling Analysis of Real-Time Systems with Precise Modeling of Cache Related Preemption Delay. *Real-Time Systems, 2005.(ECRTS 2005). Proceedings. 17th Euromicro Conference on*, pages 41–48, 2005.

[21] J. Sun and J. Liu. Bounding the end-to-end response time in multiprocessor real-time systems. *Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems*, 1995.

[22] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[23] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2-3):117–134, 1994.

[24] E. Wandeler. *Modular Performance Analysis and Interface-based Design of Embedded Systems*. PhD thesis, Swiss Federal Institute of Technology, 2006.