A Framework for the Busy Time Calculation of Multiple Correlated Events

Simon Schliecker, Matthias Ivers, Jan Staschulat, Rolf Ernst
Institute for Computer and Communications Network Engineering
{schliecker, ivers, staschulat, ernst}@ida.ing.tu-bs.de

Abstract

Many approaches to determine the response time of a task have difficulty to model tasks with multiple memory or coprocessor accesses with variable access times during the execution. As the request times highly depend on system setup and state, they can not be trivially bounded. If they are bounded by a constant value, large discrepancies between average and worst case make the focus on single worst cases vulnerable to overestimation.

We present a novel approach to include remote busy time in the execution time analysis of tasks. We determine the time for multiple requests by a task efficiently and and far less conservative than previous approaches. These requests may be disturbed by other events in the system. We show how to integrate such a multiple event busy time analysis to take into account behavior of tasks that voluntarily suspend themselves and require multiple data from remote parts of the system.

1 Introduction and Overview

The analysis of the worst case timing behavior of systems is facing new challenges with increasing system complexity. In order to derive reliable bounds, overestimations must often be introduced to reduce the analysis complexity. However, this will either increase costs or thwart industrial use altogether. Therefore, timing analysis must be sure to cover realistic system setups with tight timing bounds.

A particular challenge is the behavior of tasks that strongly interact with their environment during execution, e.g. through memory or coprocessor requests: The waiting for such external requests introduces additional delays. Thus the execution time can not be known without knowledge of these delays, which depend highly on system setup and state. Furthermore, if the scheduler reallocates the processor to other tasks, conserving the processor time, but also additionally delaying the requesting task, the response time can not be determined on the basis of the tasks core execution time alone.

Also, the focus on absolute worst case response times in system level analysis has impaired the analysis potential of such tasks: Assuming requests to cause a constant delay to the execution leads to a large overestimation in shared resource multi-task environments, where the worst case can outgrow the common case by very large factors.

The contribution of this paper is a new method to investigate communicating tasks which issue a large number of events during execution. We present methods to derive the total busy time of an execution seperated into multiple chunks, as well as the total busy time of multiple transactions over multiple resources. Both is integreated to find response times of communicating tasks. We closely examine a static priority preemptive (SPP) scheduler and show the improvement over previous work in experiments.

This paper is organized as follows: We will present related work on timing analysis in Section 2 and a new model for communicating tasks in Section 3. Section 4 presents our framework, which is implemented for a SPP scheduler in Section 5. We present an example and experiment in Section 6, and conclude in Section 7.

2 Related Work on Timing Analysis

Timing of real-time systems is addressed on different levels of abstraction. We will first present approaches that closely examine the *tasks* internal behavior. Approaches that work on the *resource* level take these results as the basis for a schedulability analysis. Finally, *system* level approaches investigate the system behavior to derive timing properties such as path latencies.

The timing analysis of *individual tasks* is commonly separated in two stages [7] [6]: Microarchitecutural modeling, in which the timing of sequences of instructions is investigated, and program path analysis to determine which path is executed in the worst-case. Memory or coprocessor access times, or cache miss penalties are assumed to be constant parameters in most approaches.

The interference of *multiple tasks on the same resource* is considered in the response time analysis. The growing-window technique is the prevailing method to solve worst case response time equations which do not lend themselves easily to direct solution. Originally introduced in [5], it has been extended to include arbitrary arrival patterns and additional timing effects e.g. in [8].

In a simple version, with no blocking time and no multiple releases within a busy window the worst case response time $WCRT(\tau)$ of a task τ with worst case execution time $C(\tau)$ on a resource with SPP scheduling is given by the smallest time w that fulfills the following equation:

$$w \quad = \quad C(\tau) + I_{hp}(w) \tag{1}$$

where $I_{hp}(w)$ is the worst case interference τ can experience due to the execution of higher priority tasks within a time window of size w.

Bletsas et al. have shown in [1] how to consider tasks whose execution is separated into actual execution times and known communication times in the response time analysis. Their approach accounts for the parallelism in local and remote execution, in so far that the interference by higher priority tasks is reduced by the "gaps" during which they wait for remote data. Still, the gap time is assumed to be constant and independant of previous behavior, which is not the case, e.g. when requests by different tasks are pipelined.

System level analysis is necessary to derive the *path latency* of memory transactions, or other requests that pass over multiple components of a system. To avoid confusion with paths within a tasks control flow, we call paths through the system *chains*.

The classical worst case response time calculation was extended to distributed systems in [9]. Other approaches, such as [4] break down the analysis complexity of complete systems into separate local analyses and bind them together with a description of the traffic (*event streams*). Attributing value dependent execution times (*modes*) to tasks [3] can lead to better local response times, if the event streams are enriched with a description of the type of events in the stream. Any of these conservative approaches focus solely on the worst case time of any single event.

The strict distinction between the different levels of abstraction was broken down in [2], where an integrated approach to perform program path analysis and derive coprocessor request latencies was presented, by assuming a worst case scenarios for each request.

3 Communicating Tasks

The presence of communicating tasks, which perform system-wide requests during their execution, contradicts a number of the assumptions of the classical analysis distribution, in which only "bottom-up" dependencies exist. The main problem is that the worst case execution behavior depends on the system level influences that can not easily be bounded before the execution behavior is known. If a conservative worst case can be found at all, it is many cases a high overestimation of the average case. Also, voluntary suspension of tasks can lead to additionaly scheduling delays. We will therfore introduce a model for communicating tasks, that enables improved reasoning about the distributed execution behavior.

The traditional task concept assumes tasks consisiting of basic blocks of linear code, branches, and loops, all represented in a control flow graph. Some worst case input pattern leads to a worst case timing behavior as shown in Figure 1a. The timing includes execution of instructions on the processor as well as memory or coprocessor calls.

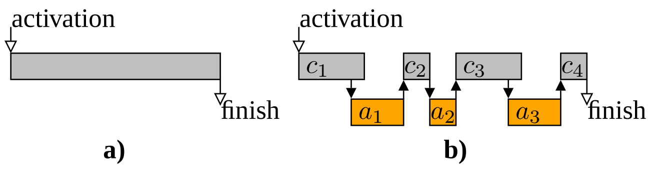

Figure 1. Task execution model.

We assume that a *communicating task* performs data requests by initiating *transactions* (a_1 to a_3) though executing an explicit instruction (`CALL a`), where `a` defines the target and type of the transaction. By calling an instruction `SYNC a`, the task will be suspended until completion of the transaction. We call the parts of a task during which no external data is required *consecutive execution sequences* (CES c_1 to c_4), each of which can be seen as corresponding to the classical task concept. A CES will often be computation but may also be communication or data storage, depending on the type of resource the CES is executed on.

A worst case task behavior exists that maximizes the time until completion, including finishing of all CESs and transactions. For the scope of this paper we assume that the maximum execution time of each CES is known, and to reduce the complexity of our problem that the amount and type of incurred transactions is not data dependant. This behavior is sketched in Figure 1b.

A transaction consists of an ordered set of *events*. Each event is the signal that causes one CES on any resource to become *ready*. This CES is then *processing* the event until it is *finished* and thus not ready anymore. When the CES is finished, the next event of the transaction becomes *ready*, activating the next CES. The first event of the transaction *initiates* the transaction and is given by the `CALL` instruction. The transaction is *finished* when the last event of the transaction was processed. When a transaction is initiated but not finished it is *ongoing* or *ready*.

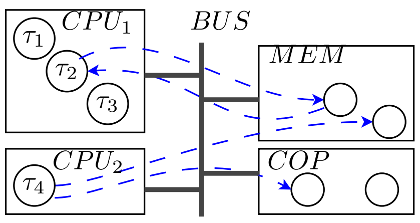

Figure 2. A Multiprocessor Setup.

Figure 2 shows the setup of an example multiprocessor system. Three tasks are mapped to processor CPU_1. Dur-

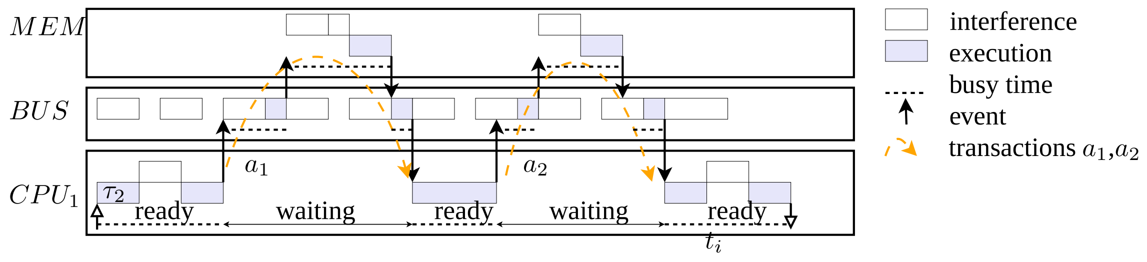

Figure 3. A distributed execution

ing its execution, τ_2 requires data from the memory MEM. A task τ_4 on CPU_2 also uses the bus and memory, interfering with the communication of τ_2. Figure 3 shows a possible Gantt diagram of this example. Task τ_2 on CPU_1 requests data from the memory two times, each time initiating a *transaction* consisting of 4 *events*. The processing of the events is delayed on the BUS and on the memory MEM, due to τ_4 performing similar accesses. As the overall time window is larger than without transactions, increased interference (for example by higher priority tasks as experienced by τ_2 at time t_i in Figure 3) occurs. The dotted lines denote the actual *overall busy time* of τ_2 which is the focus this paper.

4 A Framework for Worst Case Busy Times

We present a coupled analysis, that integrates the tasks execution behavior with the system level behavior to find the tasks response time. We set a specific focus on the analysis of multiple transaction on the system level, and the seperation of tasks into multiple parts on the local level, which is the common scenario for tasks with remote data requirements.

First, a worst case busy time analysis is introduced in Section 4.1, which allows tasks to request data multiple times during execution. This approach relies on the calculation of a local total busy time (generally addressed in Section 4.2, and specifically for SPP in Section 5), and the busy time for the memory transactions, which is calculated in Section 4.3.

4.1 Worst Case Busy Time Analysis

Extending the scope of the response time analysis from single worst case behavior to conservative bounds of multiple events requires the introduction of new terminology. Let the set of CESs of a task that leads to the largest local execution time be denoted by $\mathbb{E}$ and the set of transactions that is initiated by a task denoted by $\mathbb{Q}$. Furthermore:

The *total busy time of a set of CESs* $\mathbb{E}$ is the total amount of time during which at least one CES of $\mathbb{E}$ is ready. Thus this is the union of the times during which any single CEC is ready.

The *total busy time of transactions* $\mathbb{A}$ is the total amount of time during which at least one transaction of $\mathbb{A}$ is ready (i.e. ongoing).

The *overall busy time* is the total amount of time during which either a CES or a transaction is ready.

Figure 3 shows the execution of task τ_2, which consists of 3 local CESs on CPU_1 and initiated transactions a_1 and a_2. The dotted lines comprise the overall total busy time of the tasks CECs and transactions. The following theorem gives a overall busy time based on the definitions above.

Theorem 1. *The overall busy time of a task τ executing on resource r during a time window of size $w \geq 0$ is given by w such that*

$$w = S^r_{CES}(\mathbb{E}, w) + S_{trans}(\mathbb{A}, w) \qquad (2)$$

$\mathbb{E}$ *is the set of CESs that the task τ needs to execute locally for completion and*

$\mathbb{A}$ *is the set of transactions the task τ initiates and requires to complete execution.*

$S^r_{CES}(\mathbb{E}, w)$ *is the maximum total busy time for CESs $\mathbb{E}$ and*

$S_{trans}(\mathbb{A}, w)$ *is the maximum total busy time for transactions $\mathbb{A}$ under the assumption that all CESs and transactions can be finished within time w.*

Proof. Assume that transactions are initiated at the very last instant of each CES of τ. From the definition it follows that as soon as the transaction is finished the next CES is ready to be executed. This means that whenever no CES is ready, a transaction must be ongoing or the task is finished.

As the maximum total amount of time that transactions can be ongoing is bounded by $S_{trans}(\mathbb{A}, w)$ and the maximum amount of time CESs are ready is bounded by $S^r_{CES}(\mathbb{E}, w)$, it follows that after $S^r_{CES}(\mathbb{E}, w) + S_{trans}(\mathbb{A}, w)$ the task must be finished.

If a task issues the transactions not at the end of a CES but earlier, both the task and a transaction are ready at the same time. This can only lead to a smaller total busy time. $\square$

To solve equation 2 a growing-window technique as in [8] can be used. Initially, a non-conservative value for w can be picked. It will not be possible to perform all requested computation in time, as it will take at least until $S_{CES} + S_{trans}$ to finish. This value is used as a new w and tested. As soon as w is large enough to contain all busy times, the assumptions are correct and the analysis has converged. An example of this procedure is given in Section 6.

The above theorem is valid independently of the utilized arbitration policies, as S_{CES} and S_{trans} are unspecific. The next sections 4.2 and 4.3 focus on the derivation of these values.

4.2 Multiple Event Busy Times

One property of the memory and coprocessor requests adressed in this paper is that commonly many occur during a tasks execution. We will investigate the case of the execution of multiple CESs in a given time window. Figure 4 shows the busy time of 4 CECs that are processed in a first-in-first-out ordering. The resource is also handling requests by other tasks, which leads to delay due to interference. The dotted line depicts the searched total busy time, which is the union of the individual CESs' busy intervals (R_1 to R_4).

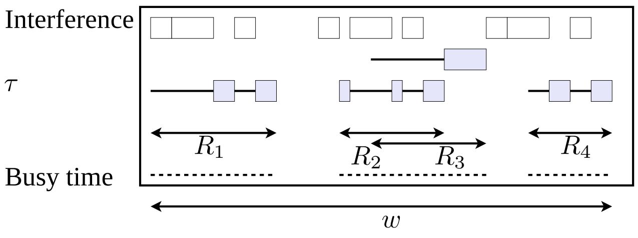

Figure 4. Single Resource Total Busy Time.

The total busy time can be used for two things: Firstly, it gives the maximum total amount of time during which this resource can be busy processing CECs that are activated by events that are part of a transaction. This is required to determine the remote total busy time S_{trans} of Theorem 1 and is investigated in Section 4.3.

Secondly, it is also an estimate on the local total busy time S_{CES} of a communicating task. The execution of such a task consists of CESs $\mathbb{E}$ (see Section 3). Although the CESs of a single task may not overlap, the total busy time is still valid.

Tindells approach to response times for bursty job arrivals [8] is not applicable, as it finds the worst case response time only within a self-inflicted and continuous *busy window*. In our case however nothing is said about the arrival times, so that events may *also* arrive completely *separated*.

Anyway, the worst case response time as derived in [8] and similar approaches may be reused: For any scheduling arbitration the maximum total busy time is bounded by the sum of the individual worst case response times $WCRT(c)$ of the CESs $c \in \mathbb{E}$.

$$S^r_{CES}(\mathbb{E}, w) \quad = \quad \sum_{c \in \mathbb{E}} WCRT(c) \tag{3}$$

Similarly, the busy time required to process a set of events $\mathbb{Q}$ to CESs on the same resource can be bounded. As each event $q \in \mathbb{Q}$ causes one CES, $c(q)$, to become ready, the set of CESs to process is given by

$$\mathbb{E} = \bigcup_{q \in \mathbb{Q}} c(q) \tag{4}$$

Note that Eq. 3 is an overestimation as can be seen in Figure 4. Firstly, not every event must wait for the processing of previous events to be finished as is assumed in the calculation of $WCRT(c)$, rather the individual busy windows overlap. Furthermore, not every request can experience the critical instant of interference by other tasks, but only a certain amount of interference can occur in in the given time window. We will therefore present an improved analysis specifically for static priority preemptive (SPP) scheduling in Section 5.

4.3 Busy Time of Transactions

In the previous section we have investigated processing of multiple events on the same resource. We will now turn to the total busy time of transactions that consists of multiple events on different resources, S_{trans}, in Eq. 2. For this, we can build on the results of section Section 4.2 (and 5).

A straightforward bound for the total busy time of transactions $\mathbb{A}$ is the sum over the worst case response times that each individual transaction would have taken. This is again an overestimation, as the worst case interference can in many cases not be imposed on every single transaction, and not every transaction may be delayed by preceding transactions. An amelioration is achieved by Theorem 2, where multiple transactions are investigated together.

Theorem 2. *Let each transaction in $\mathbb{A}$ consist of events to CESs mapped to a set of resources $\mathbb{R}$, and it is known they can be initiated and finished within a time window of size w. Let all events of the transaction on the same resource be treated with a first-in-first-out principle. Then the maximum total busy time of the transactions $\mathbb{A}$ is given by:*

$$S_{trans}(\mathbb{A}, w) \quad \leq \quad \sum_{r \in \mathbb{R}} S^r_{CES}(\mathbb{E}^{\mathbb{A}}_r, w) \tag{5}$$

where $\mathbb{E}^{\mathbb{A}}_r$ is the union of all CESs that are executed on r and activated by an event of the transaction $\mathbb{A}$ and $S^r_{CES}(\mathbb{E}^{\mathbb{A}}_r, w)$ is the maximum total busy time of these CESs.

Proof. Let $\mathbb{T}_r(c)$ be the time interval at which a specific CES c on resource r is ready. The total amount of time that resource r can be busy processing events is given by the size of the union of all times at which at least one CES of the transaction is ready on r.

As the transactions are assumed to be finished within a time window of size w, Theorem 3 bounds the total busy time of the CESs which correspond to the events in the transaction by $S^r_{CES}(\mathbb{E}^{\mathbb{A}}_r, w)$.

$$\left| \bigcup_{c \in \mathbb{E}^{\mathbb{A}}_r} \mathbb{T}_r(c) \right| \quad \leq \quad S^r_{CES}(\mathbb{E}^{\mathbb{A}}_r, w) \tag{6}$$

$\bigcup \cdot$ Produces the union of included intervals.
$|\cdot|$ Is the sum of the total size of all included intervals.

A transaction along the chain is ongoing whenever an event of the transaction is ready on any of the given resources along the chain. Therefore, the busy time of the transactions $\mathbb{A}$ is bounded by

$$S_{trans}(\mathbb{A}, w) = \left| \bigcup_{r \in \mathbb{R}} \bigcup_{c \in \mathbb{E}_r^{\mathbb{A}}} \mathbb{T}_r(c) \right| \tag{7}$$

As the size of the union of intervals can not be larger than the sum over the sizes of the intervals, equation 5 follows from equations 6 and 7. □

This framework allows to determine the worst case response time of tasks which require multiple data from other parts of the system, and initiate transactions to fetch this data. The total busy time of the transactions was determined, and the effect of the resulting voluntary suspensions of the task into multiple CESs has been taken into account in Section 4.2, albeit rather imprecisely. We will now improve the considerations about local total busy time in the following section.

5 A Static Priority Preemptive Scheduler

To show the validity of the approach presented in Section 4, we introduce a simple static priority preemptive (SPP) scheduler that arbitrates tasks with transaction and resynchronization instructions. Based on SPP scheduling, the scheduler ensures that at every time point the task with the highest priority that has all data required for execution, and has not completed execution is executing on the resource. Tasks may consist of multiple CECs, that receive the same priority as the task. All CECs with the same priority are treated first-in-first-out. For the scope of this paper, we assume that there is no blocking caused by shared critical sections, and the scheduling procedure induces no significant additional overhead.

Theorem 3. *Let a set of CESs $\mathbb{E}$ have the same priority on resource r that is scheduled with mechanisms described above. Let the processing of all CESs be started and finished within a time window of size w. Furthermore, let $C(c)$ be the worst case computation time of a CES $c \in \mathbb{E}$, and $I_{hp}(w)$ be the maximum time tasks with higher priority may be executing. Then the maximum total busy time of $\mathbb{E}$ is given by the following equation:*

$$S_{CES}^{r}(\mathbb{E}, w) \quad = \quad \sum_{c \in \mathbb{E}} C(c) \quad + I_{hp}(w) \tag{8}$$

Proof. The total busy time B is given by the sum of all times during which at least one CES is ready. Let $\text{RUN}(t)$ be the task or CES chosen by the scheduler to execute at time point t.

All times t at which $\text{RUN}(t) \in \mathbb{E}$, a CES in $\mathbb{E}$ is being executed and must therefore be ready, thus t must be included in B. This can be the case for at most $\sum_{c \in \mathbb{E}} C(c)$.

At times t, when $\text{RUN}(t) \notin \mathbb{E}$, either no CES in $\mathbb{E}$ is ready, or if one ore more CES is ready, they are kept from executing by a higher priority task that is ready. Whenever no CES in $\mathbb{E}$ is ready, B does not increase. The total amount of time higher priority tasks can be executing is limited by $I_{hp}(w)$, and in the given scheduler, at least one higher priority task is ready, only if a higher priority task is executing. Thus, whenever a CES in $\mathbb{E}$ is ready, it can not be kept from executing for more than $I_{hp}(w)$ within a time interval of size w.

Thus, the CESs in $\mathbb{E}$ can not be ready for more than $\sum_{c \in \mathbb{E}} C(c) \quad + I_{hp}(w)$ in a time interval of size w. □

Compared to section 4.2, this is a better estimate of the busy time of the CECs in $\mathbb{E}$, as now the worst case interference in the given time window w in which all processing takes place is only accounted once.

Note that the worst case interference $I_{hp}(w)$ by higher priority tasks which are allowed to suspend themselves to request data is not given by the traditional "critical instant" of all tasks being activated simultaneously [1]. Instead, the first interfering invocation has to be assumed to have performed all suspension *before* the beginning of the time window, which leads to an earlier possible activation of all successive invocations.

6 Example and Experiments

Consider the System in Figure 2 and 3. Let all resources be scheduled with the SPP scheduling as described in Section 5. We are interested in the response time of τ_2. Assume that τ_2 initiate 5 transactions $\mathbb{A}$ to the memory. Let the period and jitter be according to Table 1, and the deadline of τ_2 equal to its period. Assume that τ_1 is the only task on CPU_1 with a higher priority. On both the bus and the memory, the priority of the tasks handling τ_2's transactions are lower than the interference (I_4^1, I_4^2, and I_4^3) caused by the transactions of τ_4 on CPU_2, which occur with the period and jitter as shown. Besides their transactions, let τ_1 consist of a single CES of size 10, and τ_2 of one of size 50. Let the execution time of any execution on the bus or the memory be 10.

	CPU1		**Bus**			**Memory**	
	τ_2	τ_1	$\mathbb{A}$	I_4^1	I_4^2	$\mathbb{A}$	I_4^3
Period	400	100	n/a	100	100	n/a	100
Jitter	0	200	n/a	0	200	n/a	0

Table 1. Example Setup

A traditional response time analysis utilizes the worst case time for a single request, each delayed by the maximum amount of interference. This calculates to $WCRT_{BUS} + WCRT_{MEM} + WCRT_{BUS} = 50 + 40 + 50$ (calculation not shown). Thus for 5 requests a time of $5 * 140 = 700$ is required. Based on this, τ_2 can not keep its deadline.

Determining the worst case response time on the basis of multiple event busy times yields much tighter bounds. According to Theorem 1, the overall busy time is given by $S_{CES}^{CPU_1} + S_{trans}$, where $S_{CES}^{CPU_1}$ is denoted by S_{CPU} and S_{trans} according to Theorem 2 is given by $S_{BUS} + S_{MEM}$. Initially, it is assumed that all computation request can be handled within a time window size of $w = 50$, which is the core time of τ_2. If this were the case, the computation on CPU_1, Bus, and the memory would take 80, 130 and 80 time units respectively. Thus, the computation can not be finished within the assumed time window. A new test is done with the time window size 290, which also fails. This goes on until finally it is assumed that all computation is started and finished in a time window of size 380, thus the assumption holds, and a the worst case response time of τ_2 is found. This is an improvement towards previous approaches, as the interference in the overall busy window is only accounted for once.

	CPU1		**Bus**			**Memory**	
	$C(\tau_2) = 50$		$C(q) = 10$			$C(q) = 10$	
w	I_1	S_{CPU}	I_1	I_2	S_{BUS}	I_4	S_{MEM}
50	30	80	15	15	130	30	80
290	50	100	25	25	150	50	100
350	60	110	30	30	160	60	110
380!	60	110	30	30	160	60	110

Table 2. Calculation Procedure

We conducted a set of multiple request experiments to show the gain of our analysis ins Section 5 over the sum of worst cases approach. Figure 5 shows the estimated response times for a number of requests, from which it is known that they occur within a time window of size 300. As the number of requests increases, the total busy time only increases by the added core execution time. This results from the fact, that the complete possible interference in the time window is assumed from the beginning. This is also the reason, why the new method is inferior for $N = 1$, as in traditional WCRT only the interference during the requests busy window (not w) can interfere with the execution. As both approaches are conservative, the minimum can be used for an optimal analysis. The positive effect scales for transactions as suggested by the aboeve example.

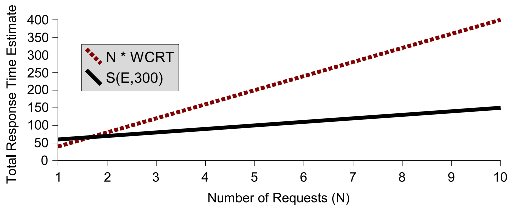

Figure 5. Multiple Request Total Busy Time.

7 Conclusion

To allow the analysis of communicating tasks that consist of local execution sequences and remote transactions we have proposed a framework that integrates the analysis over different levels of abstraction. We address multiple events together and calculate an upper bound on the total amount of busy time. This is both used for a tight estimate on the transaction latencies as well as the overall time to complete the task execution. Additionally, we presented a straight-forward analysis that accounts for the properties of static priority preemptive scheduling. For scheduling policies, where no such adopted analysis is possible or available, we have presented a conservative fall-back solution.

The experiments have shown how the consideration of a larger time window and multiple events can significantly improve the estimates on the worst case response time of a task that issues multiple memory requests.

References

[1] N. C. Audsley and K. Bletsas. Fixed priority timing analysis of real-time systems with limited parallelism. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS 04)*, Catania, Italy, jul 2004. IEEE Computer Society, IEEE.

[2] M. Ivers J. Staschulat, S. Schliecker and R. Ernst. Analysis of memory latencies in multi-processor systems. In *WCET Workshop*, Palma de Mallorca, Spain, July 2005.

[3] M. Jersak, R. Henia, and R. Ernst. Context-aware performance analysis for efficient embedded system design. In *Proceeding Design Automation and Test in Europe*, Paris, France, March 2004.

[4] M. Jersak, K. Richter, and R. Ernst. Performance analysis for complex embedded applications. *International Journal of Embedded Systems, Special Issue on Codesign for SoC*, 2004.

[5] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal (British Computer Society)*, 29(5):390–395, October 1986.

[6] Y.-T. S. Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, 1996.

[7] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by seperate cache and path analyses. *Real-Time Systems*, 18(2/3), May 2000.

[8] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time systems. *Journal of Real-Time Systems*, 6(2):133–152, March 1994.

[9] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing & Microprogramming*, 50(2-3):117–134, apr 1994.