Integrated Analysis of Communicating Tasks in MPSoCs

Simon Schliecker, Matthias Ivers, Rolf Ernst
Institute of Computer and Communication Network Engineering
Technical University of Braunschweig, Germany
{ schliecker | ivers | ernst } @ida.ing.tu-bs.de

ABSTRACT

Predicting timing behavior is key to efficient embedded real-time system design and verification. Especially memory accesses and co-processor calls over shared communication networks, basic operations of every embedded application pose a challenge for precise system analysis. Current approaches to determine end-to-end latencies in parallel heterogeneous architectures either focus on system level and allow only limited task models, or focus on activities inside a component, abstracting system level influences by overestimations.

In this paper, we identify feedbacks of the system behavior that directly or indirectly impact local execution. To tackle these complex interactions we present a novel technique to integrate an extended component level scheduling analysis with refined system level approaches. Bringing the different levels of abstraction together allows the analysis of a new class of interacting applications and architectures – which could not be addressed on a single level alone.

On the component level, we investigate two scheduling behaviors more closely, namely stalling during external requests, and allowing context-switches to other tasks that are ready. For both, we present a precise response time analysis. Finally, we compare the scheduling techniques with respect to real-time requirements.

Categories and Subject Descriptors C.4 [**Performance of Systems**]: Computer Systems Organization—*modeling techniques, performance attributes* **General Terms**: Performance, Reliability, Verification **Keywords**: real-time, multiprocessor performance analysis, memory accesses

1. INTRODUCTION

Guaranteed end-to-end latencies are often crucial to applications in the embedded systems domain. Formal analysis can be used to verify such performance constraints, but increasing system complexity introduces challanging problems to such an analysis. The increasing integration of formerly seperate components on the same chip leads to the use of the same communication and storage infrastructure for data accesses as for communication between tasks. Previous analysis approaches focus either on the behavior within a component or on abstracted system behavior. But the task's dynamic data requirements *during* their execution break the previously accepted seperation.

Classical response time analysis assumes memory accesses to be part of the worst case execution time (WCET) of a task. On the other hand, the memory access time is traditionally also considered to be out of the scope of a WCET analysis – popular methods for WCET estimation assume a given, fixed duration for the memory access. These assumptions do not fit contemporary concurrent system design. Deriving the response time for communicating tasks can therefore only be achieved by coupling the response time analysis with task *and* system level information.

CODES+ISSS'06, October 22–25, 2006, Seoul, Korea.

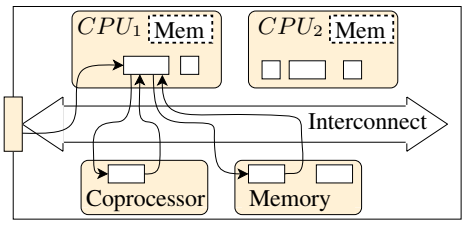

Figure 1: Multiprocessor Design Example.

Consider the simple multiprocessor architecture given in Fig 1. Two processors are connected to a memory and a coprocessor via a shared bus. Assume that CPU_1 runs a packet processing application that is part of a communication with hard deadlines. A control process on CPU_2 periodically transmits routing data, which is saved in the shared memory. When the packet processing task on CPU_1 is activated, it checks this value and potentially loads other data from the memory device. Additionally, it uses a coprocessor to speed up the packet decoding.

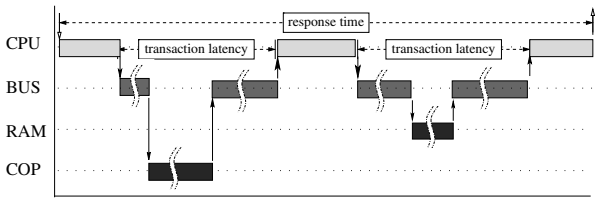

Figure 2: System-wide Scheduling Diagram.

This setup shows the analysis complexities: Memory requests generated by CPU_1 and CPU_2 have to pass a shared bus. This results in variable access time that depends not only on the target and amount of data to be transferred, but also on the current system state that is determined by all other active tasks. This leads to a feedback timing effect of the system level to the task's execution, which complicates prediction of worst case response time on the CPU_1. Deriving a static worst case memory access time a priori is only possible for a subclass of systems, and assuming this worst case access time for every request leads to large overestimation (see Fig 2).

The paper is structured as follows. We first present the necessary background on timing analysis on the system and the component level in Sec. 2. We then analyse dependencies in the system that impact our analysis in Sec. 3. Based on this we propose a refined system level analysis in Sec. 4. We put this approach into use with two specific component analyses that allow coprocessor accesses during the execution of tasks in Sec. 5. We compare the approaches in the experiments in Sec. 6 and conclude the paper in Sec. 7.

2. RELATED WORK

2.1 System Level Analysis

System level analysis is necessary to derive timing properties over multiple system components and to verify external timing constraints such as end-to-end deadlines. An important property of real time systems is the *path* or *chain latency* that is given by the worst case latency of any event from an input via a specific chain of tasks to an output.

The worst case response time of events along a chain can be derived with a number of different methods. In holistic approaches, such as [11] and [4], the classical single resource scheduling theory is extended to distributed systems. These approaches deliver good results for specific system setups, but are difficult to scale to arbitrary systems with complex dependencies.

Compositional analysis approaches [9][2] break down the analysis complexity of complete systems into separate local analyses (*component analysis*, Sec. 2.2) and integrate them using a generic description of the traffic that can lead to interference during the processing of an event (*event streams*). One such approach is presented in Sec. 2.3.

Event models are used to capture the possible patterns of task-activating event streams in a systematic, abstract fashion. They describe the minimum $\eta^{min}(\Delta t)$ and maximum $\eta(\Delta t)$ amount of events in a stream that may arise in a time window of any given size Δt. For a more compact description, they can be represented with various parameters (period, jitter, minimum distance) as is done in [2].

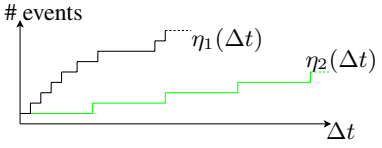

Figure 3: Different upper Bounds for Event Streams.

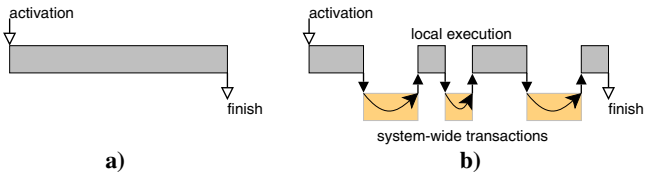

Figure 4: Task Execution Model.

2.2 Component Level

The component analysis abstracts the local effects away from the system level. Its analysis results include the response times (from any input to any output port) and the output event models at all output ports of the component.

The local worst case response time is commonly computed using scheduling analysis as in [10] based on the worst case execution times (WCET) of the individual tasks mapped to a resource, their internal messaging behavior (blocking time), and scheduling effects (overhead). In this case a detailed analysis of the tasks has to preceed the component analysis to derive the relevant properties of the tasks. The typical task model of an component analysis is depicted in Fig. 4a: The task is activated by an event token which contains all required data, executes for its worst case execution time, and emits an event token which activates the next task of the chain.

This model can be extended to *communicating tasks* that perform *transactions* during their execution as depicted in Fig. 4b. The depicted task requires three chunks of data from an external resource. It issues a *request* and may only continue execution after the transaction was e.g. transmitted over the bus, processed on the remote component and transmitted back to the requesting source.

Kim et al. [3] and Bletsas et al. [1] account for tasks with limited parallelism. They perform a response time analysis of tasks with *gaps* in their execution that exist due to computation on a different resource (i.e. an ongoing transaction). For their analysis they assume that the gap sizes are known a priori, and independent of each other. In general, however, this is not the case as unfinished transactions by other tasks may impose additional delays.

2.3 Classical Compositional Analysis

A framework to derive task response times and system latencies in a multiprocessor system has been presented in [2]. This basic analysis is presented in Fig. 5. The analysis can be separated into a component and a system level. The overall analysis procedure is then as follows:

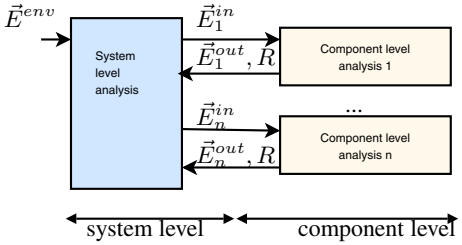

Figure 5: Analysis Concept.

1. Specify environmental model. As a characterization of the systems environment, the user supplies all environment input event models $\vec{E}^{env}$ to the system. All other input event models within the system $\vec{E}^{in}$ are initialized with optimistic guesses, which are iteratively refined during the analysis procedure.

2. Distribute input event models. The system level analysis offers event models $\vec{E}^{in}$ for incoming traffic 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 R and output event models $\vec{E}^{out}$. This can be done on the basis of classical scheduling theory (see Sec. 2.2).

4. Check for convergence. On the system level, the 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 converges, as the event models become increasingly more general with each iteration. The analysis is complete if either all event streams are stable, or if an abort condition, e.g. the violation of a timing constraint has been reached. Once the analysis has converged, the local response times can be used to derive latencies through the complete system, and the output event models describe traffic produced by the system's outputs.

3. SYSTEM INTERDEPENDENCIES

A major hurdle for the analysis of embedded systems is the high interaction and integration of its different components. Consider the multiprocessor setup in Sec. 1, where each processor has a local

memory, that are implicitly used by the tasks, and may *also* access a memory via the interconnect through explicit instructions in the source code. Under the assumption that all requests sent from the same source (i.e. processor) are treated first-come-first-serve, while the tasks on the processor itself are arbitrated with static priorities, the execution of a task τ_i can be affected by various influences:

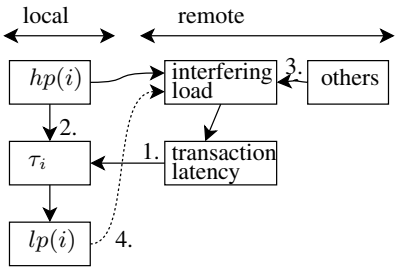

Figure 6: Dependencies in System Analysis.

1. The latency of a memory access has to be known in order to determine the WCRT of the running task, as the memory accesses are necessary for the task to finish.
2. If higher priority tasks $hp(i)$ share the processor with τ_i they may delay the execution. A bound on the possible interference must be given to determine the response time. Lower priority tasks $lp(i)$ classically only have a limited influence on the execution of τ_i that is captured a priori in the "blocking time".
3. If tasks on other processors also access the memory, their imposed load on the interconnect and the memory influences the latency of τ_i's transactions.
4. Most critically, other task on τ_i's resource may send requests to the memory. Their interference must be bounded before a response time analysis of τ_i is possible.

- Additional functional and non-functional dependencies may exist between the tasks. For simplicity, we disregard these edges.[1]

Without cyclic dependencies, an analysis is straightforward. A sequence of analyses can be derived to successively calculate intermediate results and finally the response time of τ_i. However, 4. *does* introduce such a cyclic dependency (dotted line in Fig. 3).

4. EXTENTED SYSTEM ANALYSIS

We extend the approach presented in Sec. 2.3 to allow tasks that interact with the environment during execution (Fig. 4b). Analysis of these tasks additionally requires worst case latencies for their transactions through the system.

In [8] a single communicating task on a component was investigated by incorporating independent transaction latencies into the task's control flow graph. The transactions were modelled largely uncorrelated and the question of deriving the response times of multiple communicating tasks mapped to the same resource was left open.

For the extended analysis that is able to cope with the dependencies in Sec. 3, the following additional steps are introduced, that replace *Step 3* in Sec. 2.3 for components with communicating tasks.

3a. Task analysis. Previously, the timing behavior of standard tasks was sufficiently specified with the tasks worst case execution time. For distributed tasks, the communication requirements must additionally be derived, or specified. As much information as possible about a task is gathered in this step.

[1]These dependencies may only lead to reduced possible interference, thus disregarding these edges will yield pessimistic response times.

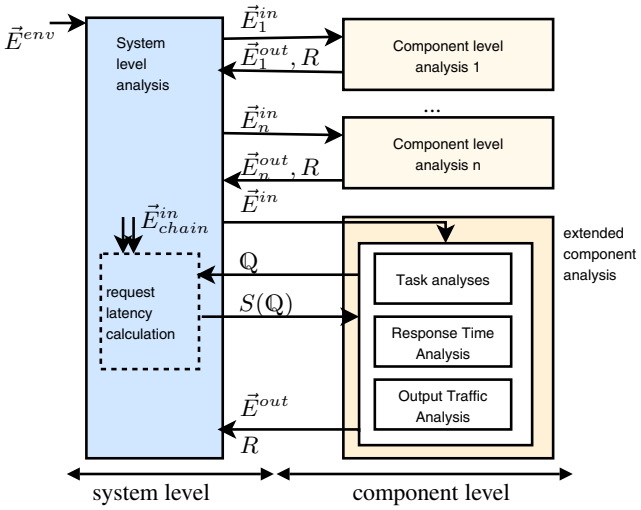

Figure 7: Analysis Concept.

3b. Derivation of latencies. A component analysis must produce output event models and response times for all tasks mapped to it. Both are (potentially) affected by the latency of the transactions issued by a task. Thus, a description of all potential transactions $\mathbb{Q}$ is compiled from the task information and sent to the system level.

On the system level, all effects that have an influence on the transaction are taken into account to derive the minimum (and maximum) latencies $S(\mathbb{Q})$ of the transactions. Bounds on all events that can interfere with the transaction along the request chain ($\vec{E}_{chain}^{in}$) must therefore be known.

3c. Extended response time analysis. Based on the information on the transaction latencies, an extended scheduling analysis must be performed to consider the delays of tasks waiting for data. Traditional scheduling analysis approaches are not valid for the communicating task model.

3d. Output event model calculation. Also, the total amount of transactions initiated from the component affects other parts of the system. Thus, a conservative bound on the produced request traffic must be determined. Although this also depends on the transaction latencies, the approach presented in [5] is conservative in any case.

The convergence of the iterative process is still ensured, because the event models are either conservatively bounded a priori, or become monotonically more generic with every iteration.

The proposed approach tackles some of the dependencies sketched in Sec. 3. Bounds on both the local and the remote interference, (2) and (3), are offered by the system level analysis. Whenever these are refined in the analysis procedure, the extended component analysis must be repeated, until the analysis has converged.

This leaves the problem of analysing the influence of transaction latencies on response time (1), and to consider the disturbance on the transaction that is sent from the *same* resource (4). This is addressed in Sec. 5.

5. EXTENDED WCRT ANALYSIS

To determine the response time of a communicating task, we assume a static priority preemptive scheduler which gives the resource to the task with the highest priority which is ready and is not blocked by an ongoing access to a critical section. For communicating tasks, we assume that they arrive periodically and their deadlines are smaller than their periods. The target of a transaction is called an external memory, but it can be a coprocessor or dedicated hardware as well. All requests are uniform and are treated along the system in a first-come-first served manner. With regard to this, the scheduler is ex-

tended to state-of-the-art behavior in Sec. 5.1 by assuming stalling of the processor during a transaction. In Sec. 5.2, we propose a multithreaded extension, where execution is possible in parallel to the memory requests and show the adapted response time equation.

5.1 Processor Stalling During Requests

Processors may allow tasks to perform coprocessor or memory accesses by offering a multi-cycle operation that stalls the complete processor until the transaction has been processed by the system. The worst case response time of a task is then not only determined by the task's worst case execution time plus the maximum amount of time the task can be kept from executing due to preemptions by higher priority tasks and blocking by lower priority tasks. A communicating task can additionally be delayed when waiting for the arrival of requested data. We can make the following observations:

1. No context switch can occur during a transaction.
2. Whenever a request is sent, no other transaction from the same processor can be ongoing.

Under these assumptions, the delay for each transaction can be bounded by a conservative worst case time. No previous requests from the same resource may be ongoing. Each transaction depends only on currentload from other processors to the memory (and bus); for which the approach presented in Sec. 4 offers conservative bounds. Let the delay for any single memory request be d^{mem}.

Delaying the tasks due to memory accesses has the same effect on lower priority tasks as increasing the core execution time: The task's finishing time is increased and no other task with lower priority may execute. The difference is that if during a transaction a higher priority tasks becomes ready, it is blocked: A context switch is not possible because the processor is stalled. This can lead to a brief *priority inversion* for the duration of one transaction. Once a higher priority task is activated no task with lower priority will have the chance to execute, and thus no further blocking may occur before the high priority task is completed. This allows us to formulate the response time equation as follows:

$$R_i = B_i + (C_i + q_i d^{mem}) + \sum_{j \in hp(i)} \eta_j(R_i) \cdot (C_j + q_j d^{mem}) \quad (1)$$

where

- R_i is the response time of τ_i.
- d^{mem} is the maximum time for one memory access.
- B_i is the maximum blocking time for τ_i: $B_i = d^{mem}$
- C_i is the worst case core execution time of τ_i.
- q_i is the maximum number of memory requests that are performed by one invocation of τ_i.
- $hp(i)$ is the set of tasks with higher priority than τ_i.
- $\eta_j(R_i)$ is the maximum amount of invocations of τ_j in a time window of size R_i.

In dynamic systems, as sketched in Sec. 1 not every memory request can experience a worst case system state, such as worst case time wheel positions in TDMA or transient overloads of priority based components. Rather, many requests in a certain amount of time can *in total* only be delayed by a certain amount interference. This overestimation can be addressed by utilizing the approach in [6] that delivers the *maximum total busy time* for multiple requests in a given time window. The total busy time is defined as the sum of all times during which at least one transaction is started but not finished. This can be used to ameliorate the response time equation by examining *all* transaction sent by the resource together:

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \eta_j(R_i) \cdot (C_j) + S(\mathbb{Q}_{i,hp(i)}, R_i) \quad (2)$$

where

- $\mathbb{Q}_{i,hp(i)}$ is the union of all requests by τ_i and all higher priority tasks.
- $\mathcal{P}$ is the set of resources involved in the processing of the transactions.
- $S_{\mathcal{P}}(\mathbb{Q}_i, R_i)$ is the maximum total busy time for the requests $\mathbb{Q}_i$ if it can be guaranteed that they can be sent and received within a time interval of size R_i under interference from other processors accessing the memory. Otherwise, it is an estimate on the smallest amount of time for which such a guarantee can be given.

Additionally, the tasks may share critical sections that may lead to additional blocking time. Memory accesses can be "nested" into critical sections, but it is technically impossible the other way around. This ensures that no new deadlocks may occur, but still allows priority inversion and thus blocking by lower priority tasks. We can map our problem to the use of critical sections as defined in [7] as follows:

- all memory accesses are accesses to the same shared resource.
- for every task τ_i that performs a memory access, the time for each access j is bounded by $d_{i,j} = d^{mem}$.
- for every task τ_i that *does not* perform a memory access we assume a "virtual" use of the shared resource with duration $d_{i,0} = 0$.

The problem is now mapped to that of nested critical sections, some of which are shared by all tasks. Access to such critical sections can be restricted with protocols such as Priority Inheritance Protocol (PIP) or Priority Ceiling Protocol (PCP). Refer to [7] for the computation of the maximum blocking time B_i. This allows us to redefine the blocking time B_i in Eq. 1 as follows

- B_i is the maximum blocking time due to shared critical sections (incl. memory accesses).

Eq. 2 shows a dependency on the knowledge of response times of higher priority tasks (edge (2)), and must therefore be evaluated "top-down". To show that Eq. 2 is suitable for the iterative response time solution, we show that it delivers a conservative workload estimation.

THEOREM 1. *If R_i is a conservative estimation of the response time, Eq. 2 delivers a conservative estimation of the maximum amount of time that τ_i may be kept from being finished in a time window of size R_i.*

PROOF. τ_i can be kept from being finished in a time window of size R_i only because the processor can either be *stalled* due to an unfinished transaction or *executing* a task (including τ_i).

Firstly, whenever the resource is *not stalled*, it must execute τ_i or a lower priority task or a higher priority task. τ_i is finished after it has executed for C_i and has not missed its deadline. It can be kept from executing by lower priority tasks and their requests maximally for a time bounded by B_i. This occurs at most once, as lower priority tasks may not execute again before τ_i is finished. Finally, higher priority tasks can not execute longer than their maximum core execution time per task activation. The maximum number of task activations in a time interval of size R_i is given by $\eta(R_i)$. Thus the maximum amount of execution that can keep τ_i from being finished is bounded by

$$exec \leq B_i + C_i + \sum_{j \in hp(i)} (\eta_j(R_i) \cdot C_j) \quad (3)$$

Secondly, the resource can only be *stalled* for the maximum amount of time that any transaction is started but not finished by any task that may send requests during the busy time of τ_i, which are τ_i and higher priority tasks. Requests by lower priority task can only delay τ_i if they occur inside their critical sections and are thus considered in the blocking time. If R_i is a conservative estimation of the response

time, all requests must be able to finish within R_i, which is the prerequisite to calculate $S(\mathbb{Q}_{i,hp(i)}, R_i)$. Then, the following bounds the maximum total busy time for all relevant transactions.

$$stall \leq S(\mathbb{Q}_{i,hp(i)}, R_i) \tag{4}$$

When the task is neither *stalled* or *executing* nothing else can keep it from being finished. Thus in a time window of size R_i the maximum amount of time that i is kept from finishing is bounded by the sum of the above. □

5.2 Multithreading Requests

Stalling the processor during each memory request is not the desirable behavior for high-throughput applications. A platform may therefore allow voluntary suspension of tasks to reuse processing time to increase processor utilization.

The modified scheduler shall behave as follows: At every point in time, the scheduler executes the task that has the highest priority and all data available to continue execution. In this case, after a transaction has been initiated, it will be executed *in parallel* to a task on the resource and therefore only has a limited effect on their worst case response times. We can assume that a transaction fully delays the execution of the sending task, but that other tasks (also of lower priority) are possibly allowed to execute.

Our observations here are:

1. The time for a transaction depends on the amount of transactions that are previously started but not finished ("open").
2. Transactions are processed in parallel, so that the processor time can be used by other tasks.
3. No blocking by lower priority tasks can occur due to their memory accesses.
4. We assume for now that all FIFOs along the request chain have sufficient size and thus no stalling can occur.

If a pure FIFO ordering is kept along the request chain, the interfering open transactions can be from tasks with higher priority, and even from tasks with lower priority.

We formulate the response time equation for multithreaded tasks as follows and show Eq. 5 is suitable for the iterative response time solution, by showing that it delivers a conservative workload estimation in the subsequent theorem.

$$R_i = C_i + S(\mathbb{Q}, \mathbb{Q}_i, R_i) + \sum_{j \in hp(i)} \eta_j(R_i + R_j) \cdot C_j \tag{5}$$

where

- $\mathbb{Q}_i$ is the set of τ_i's transactions
- $\mathbb{Q}$ is the set of all transactions that possibly interfere with $\mathbb{Q}_i$ that will be discussed in Sec. 5.2.1
- $S(\mathbb{Q}, \mathbb{Q}_i, R_i)$ is the maximum total busy time for the requests Q_i if it can be guaranteed that they can be sent and received within a time interval of size R_i under interference from other processors accessing the memory. Otherwise, it is an estimate on the smallest amount of time for which such a guarantee can be given.

THEOREM 2. *If R_i is a conservative estimation of the response time, Eq. 5 delivers a conservative estimation of the maximum amount of time that τ_i may be kept from being finished in a time window of size R_i.*

PROOF. Under the scheduler defined above, an invocation i of τ_i can either be *ready* or *waiting*. When i is *ready* it can either be running or kept from executing. i can be *ready* and kept from executing only when another invocation with higher priority is ready. In a time interval of size R_i higher priority tasks may only be ready for the maximum amount of computation jobs acquired in that time. For a task τ_j with a (known) response time R_j this is bounded by $\eta_j(R_i + R_j) \cdot C_j$. Thus, in a time interval of size R_i, τ_i can be *ready* at most for the following amount of time.

$$ready \leq C_i + \sum_{j \in hp(i)} \eta_j(R_i + R_j) C_j.$$

Additionally, i can be *waiting* for data. i can be *waiting* for data no longer than the total amount of time during which any of its requests is ongoing, which is bounded by $S(\mathbb{Q}, \mathbb{Q}_i, R_i)$. Again as in the stalling case, R_i is a conservative estimation on the overall response time, so that the prerequisites to calculate $S()$ are fulfilled.

$$waiting \leq S(\mathbb{Q}, \mathbb{Q}_i, R_i)$$

When the task is neither *ready* or *waiting* nothing else can keep it from being finished. Thus in a time window of size R_i the maximum amount of time that i is kept from finishing is bounded by the sum of the above. □

With the given response time equation, transactions and execution times are assumed to be orthogonal, so that no parallelism is assumed with respect to τ_i's response time. This appears like an overestimation, but given the task model it is a realistic scenario. Fig. 8 shows a situation where a high priority task almost fully disturbs both the execution *and* the transactions of a lower priority task. Note that such a scenario is extremely difficult to uncover by simulation.

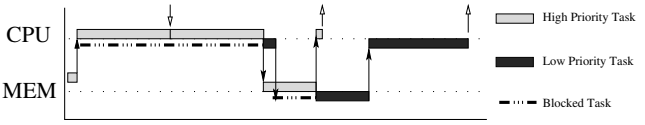

Figure 8: Example Worst-Case for Multithreaded Cores.

Parallel execution is the main objective for using multithreaded execution, but to exploit its gain in a real-time system further information about the tasks transaction distribution is required. This would then need to be taken into account by a complex phase-aware analysis. Both of which is beyond the scope of this paper.

5.2.1 Maximum delay for $\mathbb{Q}_i$

This leaves us with the problem to determine the maximum total amount of time for the transactions $\mathbb{Q}_i$ in a time window of size R_i: $S(\mathbb{Q}, \mathbb{Q}_i, R_i)$. In general, any ongoing transaction can delay a transaction of $\mathbb{Q}_i$. If the maximum amount of interfering transactions $\mathbb{Q}$ is known, we can safely assume that the total busy time is bounded by the maximum total busy time for $\mathbb{Q}_i$ *and* all transactions $\mathbb{Q}$. For this we can again use the results of [8] as in Sec. 5.1.

$$S(\mathbb{Q}, \mathbb{Q}_i, R_i) = S_{\mathcal{P}}(\mathbb{Q} \cup \mathbb{Q}_i, R_i) \tag{6}$$

A bound on $\mathbb{Q}$ can be found by investigating the maximum number of transactions per invocation of a task, and the maximum number of the tasks invocations in the time window. For this, it may be assumed that all tasks meet their deadline D_τ. For a task υ, the number of invocations in a time window of size R_i is then bounded by $\eta_\upsilon(R_i + D_\tau)$. $\mathbb{Q}$ then calculates as follows:

$$\begin{aligned} \mathbb{Q}_\upsilon &= \eta_\upsilon(R_i + D_\tau) \cdot q_i \\ \mathbb{Q} &= \cup_{\upsilon \in hp(i)} \mathbb{Q}_\upsilon \end{aligned} \tag{7}$$

However, the task deadlines of lower priority tasks can not be verified at the time of analysis of τ_i. So the scheduler must enforce it by removing any task invocation from the ready queue that has missed its deadline. Additionally, all ongoing requests of this invocation must be instantly aborted. This requires non-trivial hardware modification. Alternatively, request priorities matching task priorities can be introduced throughout the request path. Task response times can then be determined top-down, and $\mathbb{Q}_\upsilon$ calculated for all tasks with higher priority.

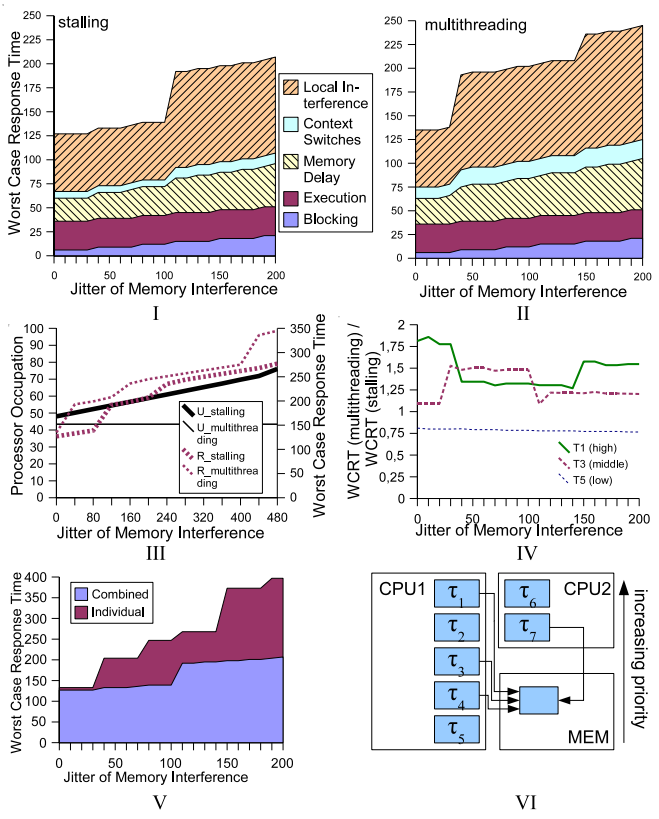

Figure 9: Various Experiments.

6. EXPERIMENTS

We have conducted a series of experiments to apply our proposed analysis. For this, our contribution was implemented into an existing real-time systems analysis framework [2].

The primary setup (see Fig. 9-VI) consist of 5 tasks that diversely access a directly coupled memory which is also shared by other components in the system. Fig. 9-I shows the response time and its components (all values are normalized time units) of task τ_3 with a medium priority when scheduled under the assumption that the processor stalls during ongoing requests. The load on the memory by other components was increased (increased jitter of memory interference), which inevitably increases the memory access time. This delays the finishing of τ_3, allowing more interference on the processor, which superlinearly increases the response time. In Fig. 9-II the same setup is shown with multithreaded behavior. In no case does this lead to a reduced worst case response time of τ_3. Rather, the additional delay by requests from *lower* priority tasks increases response time of τ_3.

A different view on system performance is given by the occupation of the processor. Stalling the processor during memory requests increases processor occupation, allowing less processing time for actual tasks. Multithreading allows a more efficient usage. Fig. 9-III shows stability of the occupation for the multithreaded case, while the occupation of the stalling resource increases with increased memory access delays. From this we can deduce that tasks that do not communicate *can* profit from multithreading as they can receive additional processing time when otherwise the processor would be stalled.

We have repeated the experiment for other tasks, namely the task with the highest priority τ_1 and the task with the lowest priority τ_5, which does not communicate at all. The relative difference to the stalling case is shown in Fig. 9-IV. τ_1 has no profit from multithreading. Rather, the effect of FCFS ordering along the request chain counters the prioritization on the resource. τ_5 on the other hand can use the gaps in the execution of the other tasks and has a 20% better response time.

Fig. 9-V benchmarks the quality of our approach to determine the response time for stalling resources. When the memory is under heavy load, we can improve the response time estimate by 50% by considering the total busy time of all requests, instead of assuming individual worst cases.

7. CONCLUSION

In this paper, we have presented a method to determine the response time of communicating tasks in dynamic multiprocessor systems. We use a compositional approach that breaks the analysis dependencies at many points: Event models are used to describe the traffic on the components and allow to determine the latency of coprocessor or memory requests. We receive tight bounds on the total request latency, by combining the scope of multiple events that may only experience worst case system state once.

We have investigated two processor behaviors: Stalling of the processor and multithreading during transactions. We presented a method to deliver tight bounds for both, with little overestimation under the given task model assumptions.

The experiments have shown that from a real-time perspective no gain can be guaranteed by introducing multithreading for communicating tasks, as both execution and transaction interference may occur during the response time. To uncover the latency hiding characteristic of multithreading more detailed behavioral models of the tasks need to be supplied.

8. REFERENCES

[1] K. Bletsas and N. Audsley. Extended analysis with reduced pessimism for systems with limited parallelism. In *RTCSA*, Hong Kong, Aug 2005. IEEE Computer Society, IEEE.

[2] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *IEE Proceedings Computers and Digital Techniques*, 152(2):148–166, March 2005.

[3] I.-G. Kim, K.-H. Choi, S.-K. Park, D.-Y. Kim, and M.-P. Hong. Real-time scheduling of tasks that contain the external blocking intervals. *RTCSA*, pages 54–59, 1995.

[4] P. Pop, P. Eles, and Z. Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *10th International Symposium on Hardware/Software Codesign*, Estes Park, Colorado, USA, May 2002.

[5] S. Schliecker, M. Ivers, and R. Ernst. Memory access patterns for the analysis of MPSoCs. In *NewCAS 2006*, Gatineau, Canada, June 2006. IEEE.

[6] S. Schliecker, M. Ivers, J. Staschulat, and R. Ernst. A framework for the busy time calculation of multiple correlated events. In *6th Intl. Workshop on WCET Analysis*, Dresden, Germany, July 2006.

[7] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), Sept. 1990.

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

[9] L. Thiele, F. Wandeler, and S. Chakraborty. Performance analysis of multiprocessor DSPs: a stream-oriented component model. *Signal Processing Magazine, IEEE*, 22(3):38–46, May 2005.

[10] 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.

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