Bounding the Shared Resource Load for the Performance Analysis of Multiprocessor Systems

Simon Schliecker, Mircea Negrean, Rolf Ernst
Institute of Computer and Network Engineering,
Technische Universität Braunschweig, D-38106 Braunschweig, Germany
{schliecker|negrean|ernst}@ida.ing.tu-bs.de

***Abstract*—Predicting timing behavior is key to reliable real-time system design and verification, but becomes increasingly difficult for current multiprocessor systems on chip. The integration of formerly separate functionality into a single multicore system introduces new inter-core timing dependencies, resulting from the common use of the now shared resources. In order to conservatively bound the delay due to the shared resource accesses, upper bounds on the potential amount of conflicting requests from other processors are required. This paper proposes a method that captures the request distances of multiple shared resource accesses by single tasks and also by multiple tasks that are dynamically scheduled on the same processor. Unlike previous work, we acknowledge the fact that on a single processor, tasks will not actually execute in parallel, but in alternation. This consideration leads to a more accurate load model. In a final step, the approach is extended to allow addressing also dynamic cache misses that do not occur at predefined times but surface dynamically during the execution of the tasks.**

I. Timing Implication of Multicore Components

Consumer demand is leading to an ever increasing application complexity in embedded systems across various domains. Strong trends towards multicore architectures can be observed in communication, media-processing, and, more recently, automotive applications, where multicore components are used to provide new functionality or to cluster previously distributed applications into a single chip.

But the application of multicore components also introduces a new level of inter-core dependencies that were previously not observed in distributed systems. The use of physically shared hardware (such as the shared memory), or synchronization via logical resources (i.e. semaphores) introduces dependencies between task executions on different cores, thus challenging the real-time behaviour of the entire system.

In single processor systems, the worst-case response time of a task under static priority preemptive scheduling classically depends on its worst-case execution time and on the maximum amount of time the task can be kept from executing due to preemptions by higher priority local tasks. When the processor is part of a multiprocessor setup, the tasks are additionally delayed when waiting for the arrival of the requested data from a shared resource (for example a shared memory). The analysis challenge lies in the fact that the access times to the shared resources are not constant, but may vary depending on the coinciding requests from other processors.

This problem is illustrated in the following example. Assume that two tasks execute on a single processor and perform requests to a remote memory (Fig. 1a), then the low priority task is kept from executing by three invocations of a high priority task. The finishing time of the lower priority task is delayed due to itself fetching data from the memory, and due to the prolonged preemptions by the higher priority task.

In a multiprocessor system the remote memory may also be accessed by tasks mapped on other processors (CPUb in Fig. 1b). Whenever requests to the now shared memory coincide with requests by tasks mapped on the other processor, the waiting time to finish the shared resource operation may increase. These conflicts contribute to the task's response time, which in the example increases so much that the task suffers a fourth preemption by the high priority task. In effect the shared resource delay challenges the safety of the task's deadline.

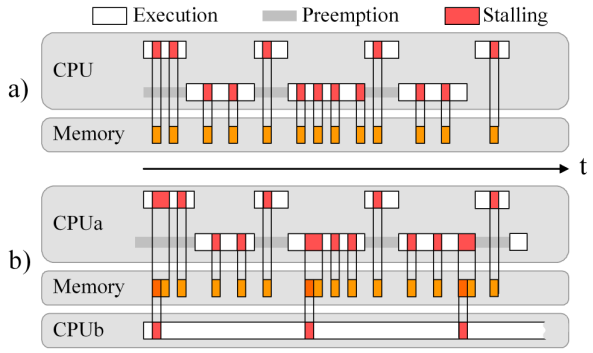

Fig. 1. a) Tasks on a Single Processor Accessing a Remote Memory b) Conflicting Accesses from Tasks Mapped on Different Processors

As the timing of the tasks in multiprocessor systems depends on the delay due to shared resource accesses, reliable and accurate upper bounds of the traffic imposed by tasks on the shared resources are required. A possible solution to this dependency is the orthogonalization of the system resources. For example, the shared resource may be arbitrated according to a static time-driven schedule, so that the shared resource delays can be determined without knowledge of the actual load imposed by the other processors. This method simplifies the timing dependency problem, but is not generally applicable, since it is not suitable for dynamic scheduling and tends to overprovision the system's resources.

Another solution is given by formal analysis approaches which can be used to find upper bounds of the timing even for dynamic system behavior. Depending on the accuracy of the request model, the analysis of the shared resource delay can vary significantly, because if larger request distances can be formally guaranteed, many conflicts can potentially be ruled

978-3-9810801-6-2/DATE10 © 2010 EDAA

out. An accurate model of the shared resource load is therefore fundamental for obtaining tight performance analysis results.

The contribution of this paper is a method for deriving the load imposed on shared resources in multiprocessor systems. The proposed method considers the timing of the task activations very generally, i.e. it is not constrained to periodic task models. In contrast to simpler approaches, the influence of scheduling is accurately captured by considering the exclusive task execution in computation of the joint shared resource traffic of multiple tasks. By relying on work from the single-processor domain, we also extend the approach to cover cache misses that occur during the execution of a task or due to the displacement of cache blocks during preemptions.

After presenting the particular challenges that arise in the presence of shared resource usage in this section, we provide the related work in Section II. Our procedure for the quantification of shared resource traffic is presented in Section III for the case that shared resource accesses are explicit instructions, and extended in Section IV to the case where these are cache misses that implicitly surface during the task execution. We provide a small example in Section V to show the applicability of the approach, and summarize our contribution in Section VI.

II. Related Work

To avoid the feedback effect of shared resource timing on the task execution, an increasingly common counter-measure is the orthogonalization of system resources [1], for example through using a system crossbar or time-driven scheduling of the memory bus (as in [2][3]). By reducing the timing interdependence, the tasks on each core can then be verified separately. While this option simplifies the verification procedure, it implies a conservative design with in general increased resource and possibly also power requirements. In [4] and [5] the worst-case cost of orthogonalization was significantly reduced by deriving optimal bus schedules given the memory access pattern of each task. However, the approaches do not support local task scheduling and are thus applicable only as long as the number of tasks in the system does not outgrow the number of processors.

Modern processor pipelines and memory architectures are becoming increasingly complex, posing several challenges to formal analyses [6][1]. To facilitate tight bounds, a certain degree of timing composability is required to constrain the state space that needs to be investigated to find tight worst-case execution time estimates. For the scope of the present paper, we assume that each processor core has a timing-compositional architecture, in the sense that any shared resource delays are additive to the execution times.

If no orthogonalization measures are in place for the shared system resources, a model on the run-time load imposed on the shared resource from different components in the system has to be established. The derivation of conservative, application dependent resource request bounds for individual tasks was the concern of [7] and [8], where the task's internal control flow was investigated. The basic assumption is that for each basic block the execution time is either constant or a minimum execution time and a maximum number of shared resource requests is known. Through program path analysis, distances between multiple requests are derived.

To overcome the problem of mutual dependency between the response time analyses of tasks on different processors in the presence of shared resources, the load imposed on the shared resources can be expressed using the event model concept [9][10], which has previously been used to model task activations [11]. Based on this, the analysis of tasks that perform accesses to a shared resource can then be decomposed into three major building-blocks: (i) the quantification of the amount of shared resource operations issued by a task and all tasks on a processor (this is the focus of the present paper); (ii) the analysis of the total latency experienced by a set of such operations on the shared resource; and (iii) the inclusion of this delay in the response time analysis of each task.

In event-driven multiprocessor systems, this analysis procedure has various mutual dependencies (between the task activating event models, the shared resource delays, and the task response times). As a solution, an iterative approach has been proposed in [10] that relies on the monotonic properties of all involved analyses to find a fixed-point that represents a conservative solution.

III. Deriving Bounds on the Shared Resource Requests

To express the shared resource load from a given processor, we rely on the concept of event models. These are described using the upper and lower event arrival functions $\eta^+(\Delta t)$ and $\eta^-(\Delta t)$ which specify the maximum respectively the minimum number of events that occur in the event stream during any time interval of length Δt. Inversely, an event model can also be specified using the functions $\delta^-(n)$ and $\delta^+(n)$ that represent the minimum and maximum time windows in which n events can be observed in the stream. Both functions can be straight-forwardly converted to each other as follows (see also Figure 2).

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

$$\eta^+(\Delta t) = \max_{n\in\mathrm{N},n\geq 1}\{n \mid \delta^-(n) \leq \Delta t\} \quad (2)$$

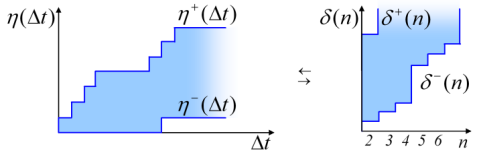

Fig. 2. Event Stream Representations.

The shared resource request bound is indicated by a $\tilde{\eta}$ and $\tilde{\delta}$ to differentiate it from task activating event models. It is defined as follows:

Definition 1. *The* Shared Resource Request Bound $\tilde{\eta}_{T\to S}(\Delta t)$ *is the maximum amount of requests that may be issued from a set of tasks T to a shared resource S within a time window of size $\Delta t > 0$. The* Shared Resource Request Distance $\tilde{\delta}_{T\to S}(n)$ *is the minimum time during which $n, n \geq 1$ requests may be issued from a set of tasks T to a shared resource S.*

If T contains only one task (e.g. j), we denote its shared resource request bound simply with $\tilde{\eta}_{j\to S}(\Delta t)$ and $\tilde{\delta}_{j\to S}(n)$; also the index S may be omitted for brevity. Task activations may overlap. If a task is re-activated before a running instance of the same task is finished, we assume that the concurrent instances are processed strictly in order.

The remainder of this section provides the resource request bound of a given task and aggregates individual task traffic to the joint traffic issued from a processor.

A. Remote Operations Initiated by a Single Task Instance

If shared resource requests are the result of explicit instructions in the source code, the amount of requests that are issued per task instance can be bounded by investigating the task's internal control flow. For example, a task may fetch data each time it executes a for-loop that is repeated several times. By multiplying the maximum number of loop iterations with the amount of fetched data, a bound on the memory accesses can be derived. Focused on the worst-case execution time problem, previous research has provided various methods to find the longest execution path through such a program description with the help of integer linear programming (see [12]). By modifying the node weights to the number of resource request, this approach can be adapted to find the path with the maximum number of requests N_j^{max} per task instance (which may not necessarily be the path with the maximum execution time).

Depending on the actual system configuration, relying solely on the upper bound N_j^{max} on the number of requests per task instance may not be sufficiently accurate. In the analysis of the shared resource contention, this may translate into an assumed burst of requests that may not occur in practice, resulting in an overestimated shared resource load. Therefore, it is worthwhile to identify a minimum time $\tilde{d}_j(n)$ that a task instance must execute in order to produce n requests.

A simple bound is given in the following lemma, which assumes that there is a known minimum distance d_{sr} between any 2 requests to the shared resource. Refined methods that consider the task's control flow have been proposed in [7], [8].

Lemma 1. *The minimum request distance $\tilde{d}_j(n)$ between any n requests issued by an instance of task j that performs a maximum of N_j^{max} requests per instance is for $2 \le n \le N_j^{max}$ bounded by*

$$\tilde{d}_j(n) \quad = \quad (n-1)\cdot d_{sr} \tag{3}$$

where d_{sr} is the minimum distance between any 2 request.

B. Multiple Instances of the Same Task

The shared resource traffic of multiple instances of the same task will exhibit a hierarchical pattern that follows from the repetitive activation of the task. The following theorem bounds the minimum distance between a number n of requests from different instances of the same task. For this, the distance between the required number of task instances (as defined by its activating event model) as well as the distance between the individual events during each task instance are considered.

Theorem 1. *If a task j is allowed to execute in such a way that each instance finishes earlier than R_j after its arrival, it will not produce more than n requests to a resource S in a time interval of size $\tilde{\delta}^a_{j\to S}(n)$:*

$$\begin{aligned}\tilde{\delta}^a_{j\to S}(n) \quad = \quad & \min_{1\le k\le \min\{n,N_j^{max}\}}\{ \quad \tilde{d}_j(k) \\ & + \quad \delta'_j(\lceil (n-k)/N_j^{max}\rceil + 1) \\ & + \quad \tilde{d}_j(n-k-(\lceil (n-k)/N_j^{max}\rceil - 1)\cdot N_j^{max})\ \}\end{aligned} \tag{4}$$

where $\tilde{d}_j(k)$ is the minimum time that an instance of task j must execute in order to produce k requests to resource S (e.g. as provided by Lemma 1), and $\delta'_j(n)$ is the minimum distance between the execution of n instances of task j as given by its activating event model and worst-case response time: $\delta'_j(n) = \max\{0, \delta_j^-(n) - R_j\}$.

Proof: Let all shared resource operations issued by task j be numbered in the order of their request times. Assume two arbitrary requests m_1 and m_2 with $m_2 = m_1 + n - 1$. Let the task instances that produce request m_1 and m_2 be denoted with $i(m_1)$ and $i(m_2)$, respectively. Let the instance $i(m_1)$ produce k of the n requests. It then must execute for at least $\tilde{d}_j(k)$ before it has done so.

The remaining $n-k$ requests involve at least $\lceil (n-k)/N_j^{max}\rceil$ further task activations, because no instance can produce more than N_j^{max} requests. The distance between the activation of instances $i(m_1)$ and $i(m_2)$ is constrained by the task's activating event model ($\delta_j^-(n)$). Because $i(m_1)$ executes for no longer than R_i, $i(m_2)$ can not be activated sooner than $\delta_j^-(\lceil (n-k)/N_j^{max}\rceil + 1)) - R_j$ after $i(m_1)$ has finished, as provided in the second line of (4). Even if the intermediate instances produce the maximum number of requests N_j^{max} each, then still $n-k-(\lceil (n-k)/N_j^{max}\rceil - 1)\cdot N_j^{max})$ requests remain to be produced by instance $i(m_2)$. In order to produce the remaining amount of requests, the task must again execute for at least as long as demanded by $\tilde{d}$ as stated in the last line of (4).

If the intermediate instances produced less than the maximum number of requests each, $i(m_2)$ would need to produce more remaining requests, which would increase the distance between m_1 and m_2. The arbitrary instance $i(m_1)$ may produce any number k of requests that is smaller than both n and N_j^{max}. The minimum of these scenarios is a lower bound on the actual distances. ∎

Equation 4 provides a minimum distance between any n requests by instances of the same task. To compute this bound the response time of the task R_j needs to be known, which may initially not be the case, because the response times have only been computed for some tasks in the system. This issue can be addressed by iteratively computing the shared resource request bounds and task response times in the system. Such a procedure has been proposed in [10].

When reasoning about the requests of multiple tasks in the next section, an important metric is the amount of computation (i.e. processor occupation) that is involved in order for a task to produce a certain number of requests. The following theorem provides a bound on this execution requirement.

Theorem 2. *In order to produce n requests to a shared resource S, instances of a task j with best-case execution times t_{BC} must execute at least*

$$\begin{aligned}\tilde{\delta}^e_{j\to S}(n) = \ & \min_{1\le k\le \min\{n,\tilde{e}_j(t_{BC})\}}\{ \quad \tilde{d}_j(k) \\ & + \max\{0, (\lceil (n-k)/\tilde{e}_j(t_{BC})\rceil - 1)\}\cdot t_{BC} \\ & + \tilde{d}_j\left(n-k-(\lceil (n-k)/\tilde{e}_j(t_{BC})\rceil - 1)\cdot \tilde{e}_j(t_{BC})\right)\}\end{aligned} \tag{5}$$

where the maximum number of requests per best-case execution time $\tilde{e}_{j\to S}(t_{BC})$ is derived from the minimum request distances $\tilde{d}_{j\to S}(n)$ according to Lemma 1 and Equation (2).

Proof: Follows along the lines of the proof for Theorem 1. The first, the last and the intermediate instances (lines 1, 3, and 2 of Equation 5) require a certain time to execute in order to provide the expected amount of requests ($\tilde{e}_j(t_{BC})$ requests per t_{BC} results in the highest possible density of requests). ■

It is worth noting that Theorem 1 makes no assumptions about the manner in which the task j is actually scheduled, while Theorem 2 makes no assumptions about the manner in which the task is activated. Thus, both theorems deliver orthogonal and equally valid minimum distances between the requests of a task.

C. Scheduling Multiple Tasks on the Same Processor

Tasks that share the same processor are executed alternately as directed by the local scheduling policy. This results in a combined request traffic $\tilde{\delta}^-_{T\rightarrow S}(n)$ for all tasks T mapped to the same processor. In this section we present two orthogonal lower bounds on the distances between the requests that are derived from the information per task of the previous section.

1) Minimum Distance Demanded By Task Execution Intervals: Theorem 1 has provided bounds on the distance between any n requests issued by instances of a task j on the basis of the distance between the task activations and its worst-case response time, but independently of the amount of actual processing time that is assigned to the task (as long as it respects its worst-case response time). Consequently, this bound is valid for every task $j \in T$ for any actual schedule.

A total of n requests can be observed in the smallest time window, in which the sum of requests is not smaller than n. This is stated in the following corollary.

Corollary 1. *The smallest time window in which n requests by tasks in a set T can be observed is larger than*

$$\tilde{\delta}^a_{T\rightarrow S}(n) = \min\{\Delta t \mid \sum_{j\in T} \tilde{\eta}^a_{j\rightarrow S}(\Delta t) \geq n\} \qquad (6)$$

with $\tilde{\eta}^a_{j\rightarrow S}(t)$ derived from $\tilde{\delta}^a_{j\rightarrow S}(n)$ according to Theorem 1 and Equation (2).

2) Minimum Distance Demanded By Exclusive Task Execution: Although Corollary 1 is conservative, it can be an underestimation of the actual distances, because the tasks mapped to the same processor do not actually run in parallel, but rather the scheduler will assign the processor exclusively to the different tasks over time. The effect of this exclusive assignment is illustrated in the following example. Assume a set of tasks T is executing on the same processor and the tasks perform accesses to a shared resource S. In order for these tasks to produce a total of n requests, the tasks have to be scheduled in such a way that the sum of the requests by each task adds up to n (i.e. if n_j is the number of requests issued by task j, we have $\sum_{j\in T} n_j = n$). In order to produce n_j requests, task j must execute for a certain amount of time $e_j(n_j)$. Because at every point in time, only one task can be executed on a processor, the total time to produce the n_j requests is thus given by $\sum_{j\in T} e_j(n_j)$.

The following theorem exploits that the execution time required for each task to produce its share of requests is at least $\tilde{\delta}^e_{j\rightarrow S}(n_j)$ according to Theorem 2.

Theorem 3. *If the tasks in a set T are scheduled alternatingly on a processor, the smallest time window in which n requests to a resource S may be observed is bounded by*

$$\tilde{\delta}^e_{T\rightarrow S}(n) = \min\{\sum_{j\in T} \tilde{\delta}^e_{j\rightarrow S}(n_j) \mid \sum_{j\in T} n_j = n\} \qquad (7)$$

where $\tilde{\delta}^e_{j\rightarrow S}(n_j)$ is the minimum time that instances of task j must execute in order to produce n_j requests to resource S.

Proof: For a total number of n requests to be issued by the tasks in the set T, the scheduler must select tasks for execution in such a way that the sum of the requests of the individual tasks is n. Thus, the problem is subject to the constraint $\sum_{j\in T} n_j = n$.

The total time that needs to pass in order for the tasks in T to produce the n requests is given by the sum over the times that the individual tasks must execute in order to produce their respective share of the n requests $\sum_{\tau_j\in T} \tilde{\delta}^e_{j\rightarrow S}(n_j)$, as bounded by Theorem 2. Consequently, if the distribution of requests to tasks is such that this sum is minimized, the amount of time to produce the n requests is minimized. ■

Finding the combination of requests per task (n_j) that leads to a minimum overall request distance in (7) is an instance of the "bounded nonlinear Knapsack problem" [13], which in general belongs to the class of NP-hard problems. However, relatively efficient solutions have been proposed e.g. in [14]. The greedy approach of iteratively taking the smallest incremental execution time does not lead to conservative results, because the functions $\tilde{\delta}^e_{j\rightarrow S}$ are not necessarily *convex* functions (only *super-additive*).

The individual shared resource request bounds of two tasks and the joint bound derived on the basis of Theorem 3 are depicted in Fig. 3. The most critical case is given if the scheduler first chooses task 1 for execution until it has produced 4 requests. In order to produce a fifth request however, task 1 would have to execute relatively long, thus executing task 2 becomes more critical. For comparison, the request distances provided by Corollary 1 (i.e. the horizontal summation of the shared resource request bounds) are shown as a dotted line.

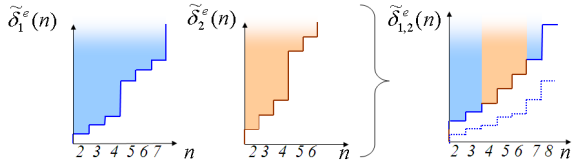

Fig. 3. Joint Request Bound under Exclusive Execution.

Both bounds from Corollary 1 and Theorem 3 make orthogonal assumptions and are equally valid. In the case of overlapping instances, $\tilde{\delta}^e_{T\rightarrow S}(n)$ is more accurate (as seen in Fig. 3), but when task activations are further apart, $\tilde{\delta}^a_{T\rightarrow S}(n)$ better captures the distances for large n.

IV. Implicit Accesses to a Shared Memory

The aggregation of request traffic in the previous sections has assumed that the requests are issued within predefined bounds during the execution of the tasks, and that these bounds are not affected by the actual scheduling decisions. This is the case when the requests are the result of explicit instructions in the source code. However, when requests are the result of cache misses, their timing is the implicit effect of the current

system state and can heavily vary due to dynamic context switching.

This challenging scenario can be addressed with the analysis proposed in the previous sections and a three-step extension: In a first step, the number of "intrinsic" cache misses of an undisturbed task execution is bounded. The second step derives the additional cache misses due to the run-time preemptions. Based on these values, the total number of cache misses is considered in the computation of the shared request bound.

A. Bounding Intrinsic Cache Misses

Bounding the tasks' intrinsic cache misses has been a heavily researched subject. By so-called abstract interpretation of the execution binary and an accompanying cache model, formal methods are able to identify for each basic block the maximum number of cache misses that may occur during the execution [12], [15]. The tasks control flow graph is then annotated with the possible cache miss delays per basic block, and the task's worst-case execution time is derived by searching for the longest path. This procedure is obviously only possible if the cache miss delay is constant, which is the case for single processor systems, but does not hold for shared memories with dynamic arbitration.

Instead, we propose to annotate the basic blocks in this case only with the amount of potential cache misses that go to a shared memory M. This then re-enables the analysis of the maximum number of cache misses per task execution $N_{j\to M}^{intrinsic}$ and minimum distance between cache misses $\tilde{d}_{j\to M}^{intrinsic}(n)$ with the methods discussed in Section III-A for explicit accesses. With this, the total number of *intrinsic* cache misses by a set of tasks T that is mapped to the same processor can be computed according to Corollary 1 and Theorem 3. Let the result of this analysis be denoted with $\tilde{\delta}_{T\to M}^{intrinsic}(n)$.

B. Preemption-Related Cache Misses

When preemptive scheduling is involved, the preemption by a prioritised task can cause some useful cache blocks of a preempted task to be replaced, such that it will suffer additional cache misses when it resumes execution. This effect is called the *cache-related preemption delay* (CRPD) and is known from single processor systems, which has been investigated with formal analyses since [16].

The approaches usually consist of two steps: Firstly, the number of replaced cache blocks per preemption is bounded as in [16], [17], [18]. This figure is given by the intersection of cache blocks that are used by the preempting task, and those that are useful for the preempted task, i.e. those cache blocks that may be used again when the preempted task resumes execution. Secondly, the number of preemptions that may occur during the response time of a task is investigated. Given a certain scheduling policy, this leads to a total delay in the response time of a task [17], [19], [20]. Again, these analyses are not directly applicable due to the variable cache miss latency, but can be adapted to the given setup as follows.

To facilitate the analysis, an upper bound on the number of *preemption-related cache misses* (PRCM) is required per task pair. These misses are the bare number of cache misses as derived by the method proposed in [16], but without factoring in the cache miss penalty. Considering the possible number of preemptions, the amount of misses is aggregated to find the total number of misses $N_{T\to M}^{prcm}(w_i)$ of all tasks T on a processor within a given time window w_i. The algorithms in [19] and [20] provide solutions to this problem for static priority preemptive scheduling with different trade-offs between complexity and accuracy. In the multiprocessor context, we apply these methods on the basis of the bare number of cache misses (instead of the delays).

C. Overall Bound on Cache Misses

Finally, the total number of cache misses consisting of the *intrinsic cache misses* and *preemption-related cache misses* can now be considered in the computation of the shared resource load.

The preemption-related cache misses cause the tasks to produce more requests in shorter time intervals. The resulting minimum distance between any n cache misses that go to a shared memory M can be computed using the bounds on the intrinsic cache misses $\tilde{\delta}_{T\to M}^{intrinsic}(n)$ and the additional scheduling-related cache misses $N_{T\to M}^{prcm}(w_i)$ on a processor within a time interval w_i:

$$\tilde{\delta}_{T\to M}^{cache}(n) = \min[0, \tilde{\delta}_{T\to M}^{intrinsic}(n - N_{T\to M}^{prcm}(w_i))] \tag{8}$$

V. Integration of Results and Experiments

To evaluate the proposed method, we integrate shared resource request bounds derived in the previous sections into the worst-case response time analysis for tasks in multiprocessor systems with shared resources (following [10]). For each task i, its response time is given by the sum of a) its own worst-case execution time, b) the possible preemption time due to higher priority tasks, and c) the delays experienced when waiting for shared resource requests to be finished (i.e. the processor is stalled during requests). The latter is computed as the sum over the duration of i's requests, the requests of higher priority preempting tasks, and the requests by tasks on other processors in the same time window.

We consider a system with 2 processors that are equipped with local caches (L1) and can access a shared memory that serves requests first-come-first-served with a constant time of 5 time units per access (i.e. there is no L2 cache). Each processor is running two tasks as presented in Table I. The tasks are actual benchmarks taken from [15] investigated with a modified version of the worst-case execution time analysis tool [21]. The derived WCET values without cache miss delays are listed in Table I. A different number of intrinsic cache misses and preemption-related cache misses is obtained for various cache configurations (cache size between 64 and 1024 Byte; direct mapped; replacement strategy least-recently-used). It can be observed that the development is not monotonic, but that there is a general tendency that the intrinsic cache misses decrease with larger cache sizes. A different tendency can be observed for the preemption-related cache misses for which the cost of a single preemption is listed in Table I. The PRCMs increase with growing cache size, because more useful cache blocks can be replaced upon a preemption. When the cache is sufficiently large, the displacement is again reduced.

TABLE I
EXPERIMENTAL SETUP AND COMPUTED CACHE MISSES

Task	Processor	Priority	Period	WCET	$N_{j \to M}^{max}$ [64,128,256,512,1024]B
countsort	CPU0	1	20000	168	[55,98,80,12,60]
whetstone	CPU0	2	75000	57253	[790,790,790,550,50]
FIR	CPU1	1	20000	2083	[155,110,8,35,8]
exchangesort	CPU1	2	40000	11011	[1115,710,710,710,710]

Preemption Scenario	64B	128B	256B	512B	1024B
"*countsort* preempts *whetstone*"	8	15	32	46	25
"*FIR* preempts *exchangesort*"	8	15	31	63	15

These values are now used to derive the load imposed from each processor to the shared memory according to Equation 8, and the delays and response times are computed as described in the first paragraph of this section.

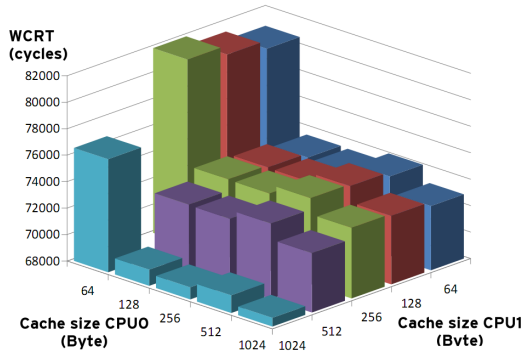

Fig. 4. Resulting Worst-Case Response Times (WCRT) for *whetstone* task on CPU0.

Fig. 4 shows the resulting response times of the *whetstone* task for the different cache configurations. As can be expected, the response time is influenced by both the local cache configurations and that of the other processor, as either misses will cause an execution delay. The result is not monotonic with respect to cache size, as the intrinsic and the preemption-related cache misses develop in different directions. One configuration has lead to a response time that was beyond the task deadline (64B cache on CPU0, and 512B cache on CPU1). The most economical configuration is a cache size of 128B on CPU0 and 64B on CPU1, as larger cache sizes do not deliver a dramatically improved performance.

VI. CONCLUSION

In this paper, we have highlighted the need for accurate models for shared resource load in multiprocessor systems. An analysis was presented to derive a shared resource request bound for single tasks, and also for task sets scheduled on the same processor. The joint bound explicitly considers the distance between successive task activations and the alternating execution of tasks on the same processor. Based on single processor cache analyses, we have provided extensions of this approach to cover the effects of intrinsic and preemption-related cache misses. The applicability was shown in an experimental section using actual benchmarks.

ACKNOWLEDGMENTS

The authors would like to thank Axel von Engel for building a prototype implementation and performing the experiments.

REFERENCES

[1] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 7, pp. 966–978, July 2009.

[2] P. Paulin, C. Pilkington, and E. Bensoudane, "StepNP: a system-level exploration platform for network processors," *Design & Test of Computers, IEEE*, vol. 19, no. 6, pp. 17–26, 2002.

[3] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. van Meerbergen, "Predictable embedded multiprocessor system design," *Proceeding of the SCOPES workshop, September*, 2004.

[4] A. Andrei, P. Eles, Z. Peng, and J. Rosen, "Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip," *21st Intl. Conference on VLSI Design*, 2008.

[5] M. Schoeberl and P. Puschner, "Is Chip-Multiprocessing the End of Real-Time Scheduling?" in *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*. Dublin, Ireland: OCG, July 2009.

[6] R. Kirner and P. Puschner, "Obstacles in worst-case execution time analysis," in *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, May 2008, pp. 333–339.

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

[8] 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*, pp. 97–106, 2006.

[9] T. Henriksson, P. van der Wolf, A. Jantsch, and A. Bruce, "Network Calculus Applied to Verification of Memory Access Performance in SoCs," in *Workshop on Embedded Systems for Real-Time Multimedia (ESTIMEDIA)*, Salzburg, Austria, October 2007.

[10] S. Schliecker, M. Negrean, and R. Ernst, "Response time analysis in multicore ecus with shared resources," *IEEE Transactions on Industrial Informatics*, vol. 5, no. 4, November 2009.

[11] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System Level Performance Analysis - The SymTA/S Approach," *IEE Proceedings Computers and Digital Techniques*, vol. 152, no. 2, pp. 148–166, March 2005.

[12] R. Wilhelm *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *Trans. on Embedded Computing Sys.*, vol. 7, no. 3, pp. 1–53, 2008.

[13] S. Martello and P. Toth, *Knapsack problems: algorithms and computer implementations*. John Wiley and Sons Ltd., 1990.

[14] D. Li, X. Sun, J. Wang, and K. Mckinnon, "Convergent lagrangian and domain cut method for nonlinear knapsack problems," *Comput. Optim. Appl.*, vol. 42, no. 1, pp. 67–104, 2009.

[15] J. Staschulat and R. Ernst, "Scalable precision cache analysis for real-time software," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 4, 2007.

[16] J. Busquets-Mataix, J. Serrano, R. Ors, P. Gil, and A. Wellings, "Adding instruction cache effect to schedulability analysis of preemptive real-time systems," in *Proc. Real-Time Technology and Applications Symposium (RTAS)*. IEEE Computer Society Washington, DC, USA, 1996.

[17] C. Lee, K. Lee, J. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim, "Bounding cache-related preemption delay for real-time systems," *IEEE Transactions on software engineering*, vol. 27, no. 9, pp. 805–826, 2001.

[18] S. Altmeyer and C. Burguière, "A new notion of useful cache block to improve the bounds of cache-related preemption delay," in *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, July 2009.

[19] S. Petters and G. Farber, "Scheduling analysis with respect to hardware related preemption delay," in *In Workshop on Real-Time Embedded Systems, London, United Kingdom, December*, vol. 3, 2001.

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

[21] J. Staschulat, "Symta/p - performance verification for complex embedded systems v1.2," http://sourceforge.net/projects/symtap/, August 2005.