Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip

Jakob Rosén, Alexandru Andrei, Petru Eles, Zebo Peng
Dept. of Computer and Information Science, Linköping University, Sweden

Abstract

In multiprocessor systems, the traffic on the bus does not solely originate from data transfers due to data dependencies between tasks, but is also affected by memory transfers as result of cache misses. This has a huge impact on worst-case execution time (WCET) analysis and, in general, on the predictability of real-time applications implemented on such systems. As opposed to the WCET analysis performed for a single processor system, where the cache miss penalty is considered constant, in a multiprocessor system each cache miss has a variable penalty, depending on the bus contention. This affects the tasks' WCET which, however, is needed in order to perform system scheduling. At the same time, the WCET depends on the system schedule due to the bus interference. In this paper we present an approach to worst-case execution time analysis and system scheduling for real-time applications implemented on multiprocessor SoC architectures. The emphasis of this paper is on the bus scheduling policy and its optimization, which is of huge importance for the performance of such a predictable multiprocessor application.

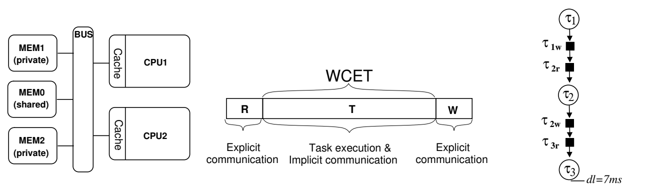
1 Introduction and Related Work

Real-time applications impose strict constraints not only in terms of their logical functionality but also with concern to timing. Classically, these are safety critical applications such as automotive, medical or avionics systems. However, recently, more and more applications in the multimedia and telecommunications area have to provide guaranteed quality of service and, thus, require a high degree of worst-case predictability [4]. Complex multiprocessor architectures implemented on a single chip are increasingly used in such embedded systems demanding performance and predictability [21].

Providing predictability, along the dimension of time, should be based on scheduling analysis which, itself, assumes as an input the worst case execution times (WCETs) of individual tasks [8, 11]. WCET analysis has been already investigated for a long time [12]. However, one of the basic assumptions of this research is that WCETs are determined for each task in isolation and then, in a separate step, task scheduling analysis takes the global view of the system [19]. This approach is valid as long as the applications are implemented either on single processor systems or on very particular multiprocessor architectures in which, for example, each processor has a dedicated, private access to an exclusively private memory. With advanced processor architectures being used in embedded systems, researchers have also considered effects due to caches, pipelines, and branch prediction, in order to determine the execution time of individual actions [9, 18, 20, 5]. However, it is still assumed that, for WCET analysis, tasks can be considered in isolation from each other and no effects produced by dependencies or resource sharing have to be taken into consideration (with the very particular exception of some research results regarding cache effects due to task preemption on monoprocessors [17, 13]). This makes all the available results inapplicable to modern multiprocessor systems in which, for example, due to the shared access to sophisticated memory architectures, the individual WCETs of tasks are depending on the global system schedule. This is pointed out as one major unsolved issue in [19] where the current state of the art and future trends in timing predictability are reviewed. The only solution for the above mentioned shortcomings is to take out WCET analysis from its isolation and place it into the context of system level analysis and optimization.

One of the major issues in the context of predictability for multiprocessor systems is the shared communication infrastructure. The traffic on the bus does not solely originate from data transfers due to data dependencies between tasks, but is also affected by memory transfers as result of cache misses. A bus access policy and bus access schedule have to be developed which (1) guarantee predictability and (2) provide efficiency in terms of system performance. In [1] we have presented our overall strategy and framework for predictable multiprocessor applications. In that paper, however, we have not addressed the issue of bus access optimization, which is a key component of the framework. Bus access optimization, the main contribution of this paper, is crucial in achieving predictability while, at the same time, maintaining efficiency in terms of performance.

Bus scheduling for real-time systems has been previously considered only in the context of inter-task communi-



(a) Target architecture (b) Task and communication model (c) Extended TG

Figure 1. System and Task Models

cation and ignoring the problem of interference caused by cache misses, [8, 11]. The issues addressed in this paper add a completely new dimension to the problem, by the dependency of WCETs on the bus schedule and by the much finer granularity at which bus transfers, due to cache misses, has to be considered.

A framework for system level task mapping and scheduling for a similar type of platforms has been presented in [3]. In order to avoid the problems related to the bus contention, they use a so called additive bus model. This assumes that task execution times will be stretched only marginally as an effect of bus contention for memory accesses. Consequently, they simply neglect the effect of bus contention on task execution times. The experiments performed by the authors show that such a model can be applied with relatively good approximations if the bus load is kept below 50%. There are two severe problems with such an approach: (1) In order for the additive model to be applicable, the bus utilization has to be kept low. (2) Even in the case of such a low bus utilization, no guarantees of any kind regarding worst-case behavior can be provided.

The remainder of the paper is organized as follows. Preliminaries regarding the system and application model are given in Section 2. Section 3 outlines the problem with motivational examples. Section 4 introduces the overall approach for implementation of predictable applications on multiprocessor SoCs and creates the context for the discussion in Section 5 regarding bus access optimization. Experimental results are given in Section 6.

2 System and Application Model

2.1 Hardware Architecture

We consider multiprocessor systems-on-chip architectures with a shared communication infrastructure that connects processing elements to the memories. The processors are equipped with instruction and data caches. Every processor is connected via the bus to a private memory. All accesses from a processor to its private memory are cached. A shared memory is used for inter-processor communication. The accesses to the shared memory are not cached. This is a typical, generic, setting for new generation multiprocessors on chip [7]. The shared communication infrastructure is used both for private memory accesses by the individual processors (if the processors are cached, these accesses are performed only in the case of cache misses) and for interprocessor communication (via the shared memory). An example architecture is shown in Fig. 1(a).

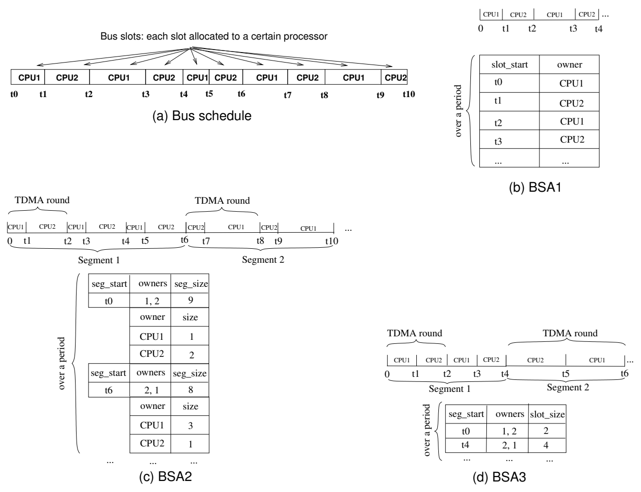
2.2 Application Model

The functionality of the software applications is captured by task graphs, $G(\Pi, \Gamma)$. Nodes $\tau \in \Pi$ in these directed acyclic graphs represent computational tasks, while edges $\gamma \in \Gamma$ indicate data dependencies between these tasks (explicit communications). The graphs (possibly also individual computational tasks) are annotated with deadlines dl that have to be met at run-time. Before the execution of a data dependent task can begin, the input data must be available. Tasks mapped to the same processor are communicating through the cached private memory. These communications are handled similarly to the memory accesses during task execution. The communication between tasks mapped to different processors is done by explicit communication via the shared memory. Explicit communication is modeled in the task graph as two communication tasks, executed by the sending (for write) and the receiving (for read) processor, respectively as, for example, τ_{1w} and τ_{2r} in Fig. 1(c). During the execution of a task, all the instructions and data are stored in the corresponding private memory, so there will not be any shared memory accesses. Whenever a cache miss occurs, the data has to be fetched from the memory, which results in memory accesses via the bus during the execution of the tasks. We will refer to these as implicit communication. This task model is illustrated in Fig. 1(b). Previous approaches that are proposing system level scheduling and optimization techniques for real-time applications only consider the explicit communication, ignoring the bus traffic due to the implicit communication [15, 11].

2.3 Bus Access Policy

In order to obtain a predictable system, which also assumes a predictable bus access, we consider a TDMA-based bus sharing policy. Such a policy can be used efficiently with the contemporary SoC buses, especially if QoS guarantees are required, [14, 10, 4].

We introduce in the following the concept of *bus schedule*. The *bus schedule* contains slots of a certain size, each with a start time, that are allocated to a processor, as shown in Fig. 2(a). The bus schedule is stored as a table in a memory that is directly connected to the bus arbiter. It is defined over one application period, after which it is periodically repeated. At run-time, the bus arbiter is enforcing the bus schedule, such that when a processor sends a bus request during a slot that belongs to another processor, the arbiter will keep it waiting until the start of the next slot that was assigned to it.



(a) Bus schedule (b) BSA1 (c) BSA2 (d) BSA3

Figure 2. Bus Schedule Table (System with Two CPUs)

The bus schedule has a huge influence on the worst-case execution time. Ideally, we would like to have an irregular bus schedule, in which slot sequences and individual slot sizes are customized according to the needs of currently active tasks. Such a schedule table is illustrated in Fig. 2(b) for a system with two CPUs. This bus scheduling approach, denoted as BSA1, would offer the best task WCETs at the expense of a very large schedule table.

Alternatively, in order to reduce the controller complexity, the bus schedule is divided into *segments*. Each *segment* is an interval in which the bus schedule follows a regular pattern, in the form of TDMA *rounds* that are repeated throughout the segment. A *round* is composed of bus slots with a certain size, each slot allocated to a different processor. In Fig. 2(c) we illustrate a schedule consisting of two bus segments with a size of 9 and 8 time units, respectively. In the first segment, the TDMA round is repeated three times. The first slot in the round is assigned to CPU_1 and has a size of 1, the second slot, with size 2, belongs to CPU_2. The second segment consists of two rounds. The first slot (size 1) belongs to CPU_2, the second one (size 3) to CPU_1. The bus scheduling approach illustrated above is denoted BSA2.

The approach presented in Fig. 2(d) and denoted BSA3 further reduces the memory needs for the bus controller. As opposed to BSA2, in this case, all slots inside a segment have the same size.

In a final approach, BSA4, all the slots in the bus have the same size and repeated according to a fix sequence.

3 Motivational Example

Let us assume a multiprocessor system, consisting of two processors CPU_1 and CPU_2, connected via a bus. Task τ_1 runs on CPU_1 and τ_2 on CPU_2. The imposed deadline is 63 time units. When τ_2 finishes, it updates the shared memory during the explicit communication E_1. We have illustrated this situation in Fig. 3(a). During the execution of the tasks τ_1 and τ_2, some of the memory accesses result in cache misses and consequently the corresponding caches must be refilled. The time interval spent due to these accesses is indicated in Fig. 3 as M_1,M_3,M_5 for τ_1 and M_2, M_4 for τ_2. The memory accesses are executed by the implicit bus transfers I_1,I_2,I_3,I_4 and I_5. If we analyze the tasks using classical WCET analysis, we conclude that τ_1 will finish at time 57 and τ_2 at 24. For this example, we have assumed that the cache miss penalty is 6 time units. CPU_2 is controlling the shared memory update carried out by the explicit message E_1 via the bus after the end of task τ_2.

A closer look at the execution pattern of the tasks reveals that the cache misses may overlap in time. For example, the cache miss I_1 and I_2 are both happening at time 0. Similar conflicts can occur between implicit and explicit communications (for example I_5 and E_1). Since the bus cannot be accessed concurrently, a bus arbiter will allow the processors to refill the cache in a certain order. An example of a possible outcome is depicted in Fig. 3(b). The bus arbiter allows first the cache miss I_1, so after 6 time units needed to handle the miss, task τ_1 can continue its execution. After serving I_1, the arbiter grants the bus to CPU_2 in order to serve the miss I_2. Once the bus is granted, it takes 6 time units to refill the cache. However, CPU_2 was waiting 6 time units to get access to the bus. Thus, handling the cache miss I_2 took 12 time units, instead of 6. Another miss I_3 occurs on CPU_1 at time 9. The bus is busy transferring I_2 until time 12. So CPU_1 will be waiting 3 time units until it is granted the bus. Consequently, in order to refill the cache as a result of the miss I_3, task τ_1 is delayed 9 time units instead of 6, until time 18. At time 17, the task τ_2 has a cache miss I_4 and CPU_2 waits 1 time unit until time 18 when it is granted the bus. Compared with the execution time from Fig. 3(a), where an ideal, constant, cache miss penalty is assumed, task τ_2 finishes at time 31, instead of time 24. Upon its end, τ_2 starts immediately sending the explicit communication message E_1, since the bus is free at that time. In the meantime, τ_1 is executing on CPU_1 and has a cache miss, I_5 at time 36. The bus is granted to CPU_1 only at time 43, after E_1 was sent, so τ_1 can continue to execute at time 49 and finishes its execution at time 67 causing a deadline violation. The example in Fig. 3(b) shows that using worst-case execution time analysis algorithms that consider tasks in isolation and ignore system level conflicts leads to incorrect results.

In Fig. 3(b) we have assumed that the bus is arbitrated using a simple First Come First Served (FCFS) policy. In order to achieve worst-case predictability, however, we use a TDMA bus scheduling approach, as outlined in Section 2.

Let us assume the bus schedule in Fig. 3(c). According

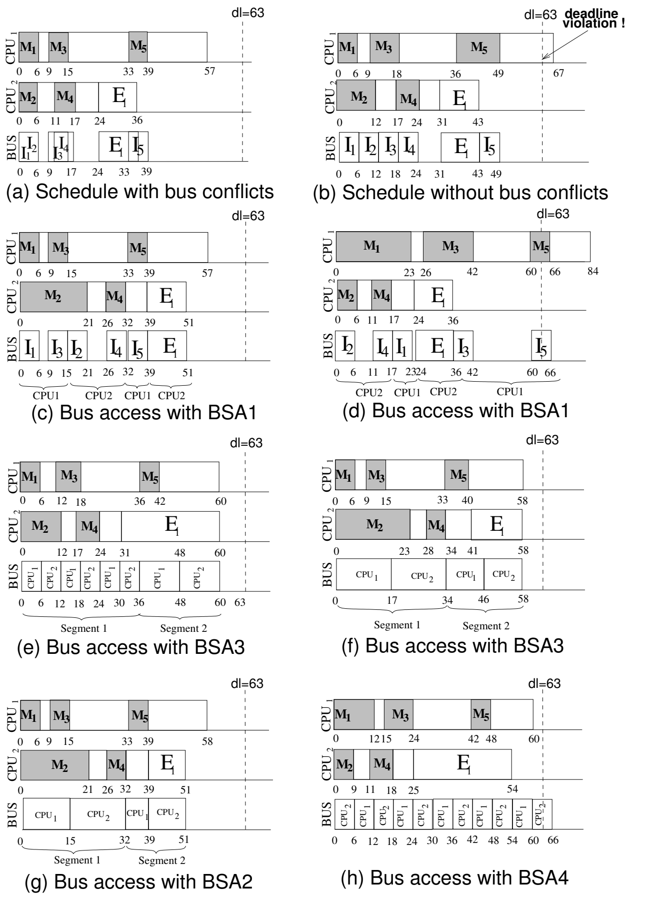


Figure 3. Schedule with Various Bus Access Policies

```
θ=0
while not all tasks scheduled
  schedule new task at $t \geq \theta$
  $\Psi$=set of all tasks that are active at time $t$
  repeat
    select bus schedule B for the
      time interval starting at $t$
    determine the WCET of all tasks in $\Psi$
  until termination condition
  $\theta$=earliest time a task in $\Psi$ finishes
end while
```

Figure 4. Overall Approach

to this schedule, processor CPU_1 is granted the bus at time 0 for 15 time units and at time 32 for 7 time units. Thus, the bus is available to task τ_1 for each of its cache misses (M_1, M_3, M_5) at times 0, 9 and 33. Since these are the arrival times of the cache misses, the execution of τ_1 is not delayed and finishes at time 57, before its deadline. Task τ_2 is granted the bus at times 15 and 26 and finishes at time 39, resulting in a longer execution time than in the ideal case (time 24). The explicit communication E_1 is started at time 39 and completes at time 51.

While the bus schedule in Fig. 3(c) is optimized according to the requirements from task τ_1, the one in Fig. 3(d) eliminates all bus access delays for task τ_2. According to this bus schedule, while τ_2 will finish earlier than in Fig. 3(c), task τ_1 will finish at time 84 and, thus, miss its deadline.

A fine grained bus schedule, such as in Fig. 3(c) and (d), potentially can provide good worst-case execution times at the expense of a complex bus arbiter that requires a very large memory for storing the schedule. We will show with the next example that a simpler bus schedule, where the allocated slots follow a certain pattern, leads to a very good compromise between arbiter complexity and task delays. For example, in Fig. 3(e), the bus access is organized according to the BSA3 approach. We divide the period into two segments. The slots from the first segment are assigned a size of 6 time units, while the slots in the second segment have 12 units. For this particular example, the order is kept the same in both segments, starting with CPU_1. Following this bus schedule τ_1 finishes latest at time 60, τ_2 at 31 and E_1 at 60. A different BSA3-based schedule, with the slot size of 17 in the first segment and 12 in the second one is illustrated in Fig. 3(f). Please note, that different BSA3-based bus schedules may lead to different worst-case execution times. In Fig. 3(e), τ_1 finishes at time 60 and τ_2 at 31, while in Fig. 3(f) τ_1 finishes at 58 and τ_2 at 41. It is crucial to choose, during the system scheduling a BSA3 bus schedule that favors the tasks on the critical path.

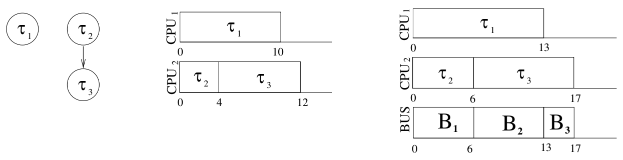
Fig. 3(g) illustrates the BSA2 approach. The bus schedule consists of two segments. The first one starts at time 0 and ends at 32. The slot sizes are 15 time units for CPU_1 and 17 for CPU_2. The second segment, starting at time 32 and finishing at 51 has a slot of 7 units allocated to CPU_1 and another slot of 12 units for CPU_2. If we compare the worst-case execution times obtained using BSA2 in Fig. 3(g), to the ones obtained using BSA3 in 3(e) and (f), we notice that BSA2 performs better, since it allows for a better customization of the bus schedule according to needs of the particular tasks.

The least flexible bus access policy, BSA4, is illustrated in Fig. 3(h). Here, the same slot size and order is kept unchanged over the whole period. This alternative produces lower quality results, but it requires a controller with a very small memory.

The examples in this section demonstrate two issues:

1) Ignoring bus conflicts due to implicit communication can lead to gross subestimations of WCETs and, implicitly, to incorrect schedules.

2) The organization of the bus schedule has a great impact on the WCET of tasks. A good bus schedule does not necessarily minimize the WCET of a certain task, but has to be fixed considering also the global system deadlines.



(a) Task graph (b) Scheduling with traditional WCET analysis (c) List scheduling with the proposed approach

Figure 5. System Level Scheduling with WCET Analysis

4 Overall Approach

4.1 Analysis, Scheduling and Optimization Flow

Let us consider a task graph mapped to a multiprocessor architecture, as described in section 2. Traditionally, after the mapping is done, the WCET of the tasks can be determined and is considered to be constant and known. However, as mentioned before, the basic problem is that memory access times are, in principle, unpredictable in the context of the potential bus conflicts. These conflicts (and implicitly the WCETs), however, depend on the global system schedule. System scheduling, on the other side, traditionally assumes that WCETs of the tasks are fixed and given as input. In order to solve this issue, we propose a strategy that is based on the following basic decisions:

1) We consider a TDMA-based bus access policy as outlined in Section 2. The actual bus access schedule is determined at design time, as will be shown in section 5 and will be enforced during the execution of the application.

2) The bus access schedule is taken into consideration during WCET estimation. WCET estimation, as well as the determination of the bus access schedule are integrated with the system level scheduling process (Fig. 4).

We present our overall strategy using a simple example. It consists of three tasks mapped on two processors, as in Fig. 5.

The system level static cyclic scheduling process is based on a list scheduling technique (outer loop in Fig. 4) [6]. For more details, please see [1].

Let us assume that, based on traditional WCET estimation (considering a given constant time for main memory access, ignoring bus conflicts), the task execution times are 10, 4, and 8 for τ_1, τ_2, and τ_3, respectively. Classical list scheduling would generate the schedule in Fig. 5(b), and conclude that a deadline of 12 can be satisfied.

In our approach, the list scheduler will choose tasks τ_1 and τ_2 to be scheduled on the two processors at time 0. However, the WCET of the two tasks is not yet known, so their worst case termination time cannot be determined. In order to calculate the WCET of the tasks, a bus configuration has to be decided on (line 06 in Fig. 4). Given a certain bus configuration, our WCET-analysis will determine the WCET for τ_1 and τ_2 (line 07). Inside an optimization loop, several alternative configurations for the current bus segment are considered (loop between the lines 05-08). The goal is to reduce the WCET of τ_1 and τ_2, having in mind the goal of reducing the global delay of the application (see section 5).

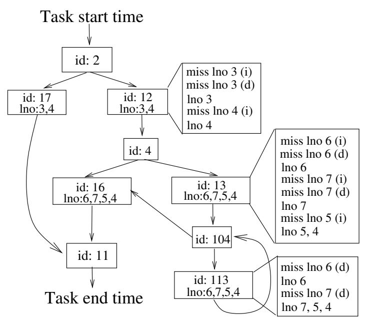
Let us assume that B1 is the selected bus configuration and the WCETs are 12 for τ_1 and 6 for τ_2. At this moment the following is already decided: τ_1 and τ_2 are scheduled at time 0, τ_2 is finishing, in the worst case, at time 6, and the bus configuration B1 is used in the time interval between 0 and 6. Since τ_2 is finishing at time 6, in the worst case, the list scheduler will schedule task τ_3 at time 6. Now, τ_3 and τ_1 are scheduled in parallel. Our WCET analysis tool will determine the WCETs for τ_1 and τ_3. For this, it will be considered that τ_3 is executing under the configuration B, and τ_1 under configuration B1 for the time interval 0 to 6, and B for the rest. Again, an optimization is performed in order to find an efficient bus configuration for the time interval beyond 6. Let us assume that the bus configuration B2 has been selected and the WCETs are 12 for τ_3 and 13 for τ_1. B2 is fixed for the interval from 6 to 13 and the procedure continues with the only task left, τ_3. The final schedule is illustrated in Fig. 5.

The overall approach is outlined in Fig. 4. At each iteration of the outer loop the set ψ of tasks that are active at the current time t, is considered. In the inner optimization loop a bus segment configuration B is fixed, as will be shown in section 5. For each candidate configuration the WCET of the tasks in the set ψ is determined. During the WCET estimation process, the bus segment configurations determined during the previous iterations are considered for the time intervals before t, and the new configuration alternative B for the time interval after t. Once a segment configuration B is decided on, θ is the earliest time a task in the set ψ terminates. The configuration B is fixed for the time interval (t, θ], and the process continues from time θ, with the next iteration.

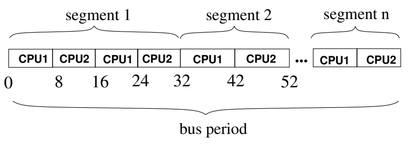
In the above discussion, we have not addressed the explicit communication of messages on the bus, to and from the shared memory. As shown in Section 2, a message exchanged via the shared memory assumes two explicit communication tasks: one for writing into the shared memory and the other for reading from the memory. A straightforward way to handle these communications would be to schedule each as one compact transfer over the bus. This, however, would be extremely harmful for the overall performance, since it would block, for a relatively long time interval, all memory access for cache misses from active processes. Therefore, the communication tasks are considered, during scheduling, similar to the ordinary tasks, but with the particular feature that they are continuously requesting for bus access (they behave like a hypothetical task that continuously generates successive cache misses such that the total amount of memory requests is equal to the worst case

```
1: void sort() {
2:   int i, temp;
3:   for (i = 0;
4:       i < 100 ;
5:       i++) {
6:         temp=a[i];
7:         a[temp]=0;
8:   }
9:}
```

(a) Task code


Task start time
id: 2
id: 17 lno:3,4
id: 12 lno:3,4
miss lno 3 (i) miss lno 3 (d) lno 3 miss lno 4 (i) lno 4
id: 4
id: 16 lno:6,7,5,4
id: 13 lno:6,7,5,4
miss lno 6 (i) miss lno 6 (d) lno 6 miss lno 7 (i) miss lno 7 (d) lno 7 miss lno 5 (i) lno 5, 4
id: 104
id: 11
id: 113 lno:6,7,5,4
miss lno 6 (d) lno 6 miss lno 7 (d) lno 7, 5, 4
Task end time

(b) Task CFG



(c) Bus schedule

Figure 6. Example Task WCET Calculation

message length). Such a task is considered together with the other currently active tasks in the set Ψ. Our algorithm will generate a bus configuration and will schedule the communications such that it efficiently accommodates both the explicit message communication as well as the memory accesses issued by the active tasks.

It is important to mention that the approach proposed in this paper guarantees that the worst-case bounds derived by our analysis are correct even when the tasks execute less than their worst-case. In [2] we have formally demonstrated this. The intuition behind the demonstration is the following:

1) Instruction sequences terminated in shorter time than predicted by the worst-case analysis cannot produce violations of the WCET.

2) Cache misses that occur earlier than predicted in the worst-case will, possibly, be served by an earlier bus slot than predicted, but never by a later one than considered during the WCET analysis.

3) A memory access that results in a hit, although predicted as a miss during the worst-case analysis, will not produce a WCET violation.

4) An earlier bus request issued by a processor does not affect any other processor, due to the fact that the bus slots are assigned exclusively to processors.

4.2 WCET Analysis

We will shortly outline the algorithm used for the computation of the worst-case execution time of a task, given a start time and a bus schedule. Our approach builds on techniques developed for "traditional" WCET analysis. Consequently, it can be adapted on top of any WCET analysis approach that handles prediction of cache misses. Our technique is also orthogonal to the issue of cache associativity supported by this cache miss prediction technique. The current implementation is built on top of the approach described in [20, 16] that supports set associative and direct mapping.

In a first step, the control flow graph (CFG) is extracted from the code of the task. The nodes in the CFG represent basic blocks (consecutive lines of code without branches) or control nodes (capturing conditional instructions or loops). The edges capture the program flow. In Fig. 6(a) and (b), we have depicted an example task containing a for loop and the corresponding CFG. For the nodes associated to basic blocks we have depicted the code line numbers. For example, node 12 (id:12) captures the execution of lines 3 ($i = 0$) and 4 ($i < 100$). A possible execution path, with the for loop iteration executed twice, is given by the node sequence 2, 12, 4 and 13, 104, 113, 104, 16, 11. Please note that the for loop was automatically unrolled once when the CFG was extracted from the code (nodes 13 and 113 correspond to the same basic block representing an iteration of the for loop). This is useful when performing the instruction cache analysis [1, 16].

We have depicted in Fig. 6(b) the resulting misses obtained after performing instruction (marked with an "i") and data (marked with a "d") cache analysis. For example, let us examine the nodes 13 and 113 from the CFG. In node 13, we obtain instruction cache misses for the lines 6, 7 and 5, while in the node 113 there is no instruction cache miss. In order to study at a larger scale the interaction between the basic blocks, data flow analysis is used. This propagates between consecutive nodes from the CFG the addresses that are always in the cache, no matter which execution path is taken. For example, the address of the instruction from line 4 is propagated from the node 12 to the nodes 13, 16 and 113.

Until this point, we have performed the same steps as the traditional WCET analysis that ignores resource conflicts. In the classical case, the analysis would continue with the calculation of the execution time of each basic block. This is done using local basic block simulations. The number of clock cycles that are spent by the processor doing effective computations, ignoring the time spent to access the cache (hit time) or the memory (miss penalty) is obtained in this way. Knowing the number of hits and misses for each basic block and the hit and miss penalties, the worst case execution time of each CFG node is easily computed. Taking into account the dependencies between the CFG nodes and their execution times, an ILP formulation can be used for the task WCET computation, [16, 13, 9].

In a realistic multiprocessor setting, however, due to the variation of the miss penalties as result of potential bus conflicts, such a simple approach does not work. The main difference is the following: in traditional WCET analysis it is sufficient for each CFG node to have the total number of misses. In our case, however, this is not sufficient in order to take into consideration potential conflicts. What is needed is, for each node, the exact sequence of misses in the worst-case and the worst-case duration of computation sequences between the misses. For example, in the case of node 13 in Fig. 6(b), we have three instruction sequences separated by cache misses: (1) line 6, (2) line 7 and (3) lines 5 and 4. Once we have annotated the CFG with all the above information, we are prepared to solve the actual problem: determine the worst-case execution time corresponding to the longest path through the CFG. In order to solve this problem, we have to determine the worst-case execution time of a node in the CFG. In the classical WCET analysis, a node's WCET is the result of a trivial summation. In our case, however, the WCET of a node depends on the bus schedule and also on the node's worst-case start time.

Let us assume that the bus schedule in Fig. 6(c) is constructed. The system is composed of two processors and the task we are investigating is mapped on $CPU1$. There are two bus segments, the first one starting at time 0 and the second starting at time 32. The slot order during both segments is the same: $CPU1$ and then $CPU2$. The processors have slots of equal size during a segment (BSA3 is used).

The start time of the task that is currently analyzed is decided during the system level scheduling (see section 4.1) and let us suppose that it is 0. Once the bus is granted to a processor, let us assume that 6 time units are needed to handle a cache miss. For simplicity, we assume that the hit time is 0 and every instruction is executed in 1 time unit.

Using the above values and the bus schedule in Fig. 6(c), the node 12 will start its execution at time 0 and finish at time 39. The instruction miss (marked with "i" in Fig. 6(b)) from line 3 arrives at time 0, and, according to the bus schedule, it gets the bus immediately. At time 6, when the instruction miss is solved, the execution of node 12 cannot continue because of the data miss from line 2 (marked with "d"). This miss has to wait until time 16 when the bus is again allocated to $CPU1$ and, from time 16 to time 22 the cache is updated. Line 3 is executed starting from time 22 until 23, when the miss generated by the line 4 requests the bus. The bus is granted to $CPU1$ at time 32, so line 4 starts to be executed at time 38 and is finished, in the worst case, at time 39.

The algorithm that performs the WCET computation for a certain task must find the longest path in the control flow graph. The worst-case complexity of the WCET analysis is exponential (this is also the case for the classical WCET analysis). However, in practice, the approach is very efficient, as experimental results presented in Section 6 show.[1] The algorithm is described in more detail in [1].

[1] In the classical approach WCET analysis returns the worst-case time interval between the start and the finishing of a task. In our case, what we determine is the worst-case finishing time of the task.

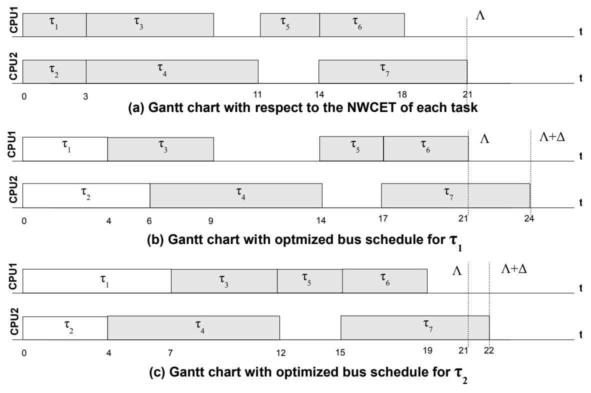


Figure 7. Estimating the Global Delay

5 Bus Schedule Optimization

In Section 2, we have introduced four bus scheduling approaches, with BSA1 being the most general and BSA4 the most restrictive. However, these two approaches are not suitable in practice due to the extreme and unmanageable memory consumption of the former and the relative non-efficiency of the latter. The other two approaches, BSA2 and BSA3, have been shown to perform close to BSA1 while keeping the memory usage on the bus arbiter low and are thus of particular interest. Consequently, in this section we propose heuristics for the bus optimization step, according to these two approaches.

5.1 Cost Function

Consider the inner optimization loop of the overall approach in Fig. 4. Given a set of active tasks $\tau_i \in \Psi$, the goal is now to generate a close to optimal bus segment schedule with respect to Ψ. An optimal bus schedule, however, is a bus schedule taking into account the global context, minimizing the global delay of the application. This global delay includes tasks not yet considered and for which no bus schedule has been defined.

This requires knowledge about future tasks, not yet analyzed, and, therefore, we must find ways to approximate their influence on the global delay.

In order to estimate the global delay, we first build a schedule S^λ of the tasks not yet analyzed, using the list scheduling technique. When building S^λ we approximate the WCET of each task by their respective worst-case execution times in the naive case, where no conflicts occur on the bus and any task can access the bus at any time. From now on we refer to this conflict-free WCET as NWCET (Naive Worst-Case Execution Time). When optimizing the bus schedule for the tasks $\tau \in \Psi$, we need an approximation of how the WCET of one task $\tau_i \in \Psi$ affects the global delay. Let D_i be the union of the set of all tasks depending directly on τ_i in the process graph, and the singleton set containing the first task in S^{λ} that is scheduled on the same processor as τ_i. We now define the tail λ_i of a task τ_i recursively as:

- $\lambda_i = 0$, if $D_i = \emptyset$
- $\lambda_i = \max\limits_{\tau_j \in D_i} (x_j + \lambda_j)$, otherwise.

where $x_j = \text{NWCET}_j$ if τ_j is a computation task. For communication tasks, x_j is an estimation of the communication time, depending on the length of the message. Intuitively, λ_i can be seen as the length of the longest (with respect to the NWCET) chain of tasks that are affected by the execution time of τ_i. Without any loss of generality, in order to simplify the presentation, only computation tasks are considered in the examples of this section. Consider Fig. 7(a), illustrating a Gantt chart of tasks scheduled according to their NWCETs. Direct data dependencies exist between tasks τ_4 & τ_5, τ_5 & τ_6, and τ_5 & τ_7; hence, for instance, $D_3 = \{\tau_5\}$ and $D_4 = \{\tau_5, \tau_7\}$. The tails of the tasks are: $\lambda_7 = \lambda_6 = 0$ (since $D_7 = D_6 = \emptyset$), $\lambda_5 = 7$, $\lambda_4 = \lambda_3 = 10$, $\lambda_2 = 18$ and $\lambda_1 = 14$.

Since our concern when optimizing the bus schedule for the tasks in Ψ is to minimize the global delay, a cost function taking λ_i into account can be formulated as follows:

$$C_{\Psi,\theta} = \max_{\tau_i \in \Psi} (\theta + \text{WCET}_i^{\theta} + \lambda_i) \qquad (1)$$

where WCET_i^{θ} is defined as the length of that portion of the worst case execution path of task τ_i which is executed after time θ.

5.2 The BSA2 Approach

The optimization algorithm for BSA2 is outlined as follows:

```
1. Calculate initial slot sizes.
2. Calculate an initial slot order.
3. Analyze the WCET of each task $\tau \in \Psi$ and
   evaluate the result according to the cost
   function.
4. Generate a new slot order candidate and
   repeat from 3 until all candidates are
   evaluated.
5. Generate a new slot size candidate and repeat
   from 2 until the exit condition is met.
6. The best configuration according to the cost
   function is then used.
```

Algorithm 1: The BSA2 Approach

These steps will now be explained in detail, starting with the inner loop that decides the order of the slots. Given a specific slot size, we search the order of slots that yields the best cost.

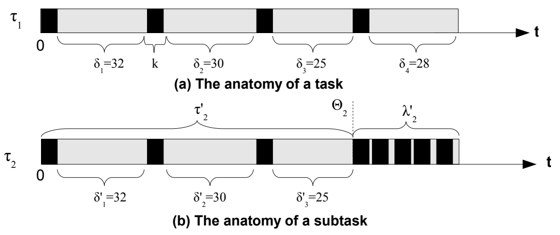


Figure 8. Close-up of Two Tasks

5.2.1 Choosing the Slot Order

At step 2 of Algorithm 1, a default initial order is set. When step 4 is reached for the first time, after calculating a cost for the current slot configuration, the task $\tau_i \in \Psi$ that is maximizing the cost function in Equation 1 is identified. We then construct $n - 1$ new bus schedule candidates, n being the number of tasks in the set Ψ, by moving the slot corresponding to this task τ_i, one position at a time, within the TDMA round. The best configuration with respect to the cost function is then selected. Next, we check if any new task τ_j, different from τ_i, now has taken over the role of maximizing the cost function. If so, the procedure is repeated, otherwise it is terminated.

5.2.2 Determination of Initial Slot Sizes

At step 1 of Algorithm 1, the initial slot sizes are dimensioned based on an estimation of how the slot size of an individual task $\tau_i \in \Psi$ affects the global delay.

Consider λ_i, as defined in Section 5.1. Since it is a sum of the NWCETs of the tasks forming the tail of τ_i, it will never exceed the accumulative WCET of the same sequence of tasks. Consequently, if we for all $\tau_i \in \Psi$ define

$$\Lambda = \max_{\tau_i \in \Psi} (\text{NWCET}_i^{\theta} + \lambda_i) \qquad (2)$$

where NWCET_i^{θ} is the NWCET of task $\tau_i \in \Psi$ counting from time θ, a lower limit of the global delay can be calculated by $\theta + \Lambda$. This is illustrated in Fig. 7(a), for $\theta = 0$. Furthermore, let us define Δ as the amount by which the estimated global delay increases due to the time each task $\tau_i \in \Psi$ has to wait for the bus.

See Fig. 7(b) for an example. Contrary to Fig. 7(a), τ_1 and τ_2 are now considered using their real WCETs, calculated according to a particular bus schedule ($\Psi = \{\tau_1, \tau_2\}$). The corresponding expansion Δ is 3 time units. Now, in order to minimize Δ, we want to express a relation between the global delay and the actual bus schedule. For task $\tau_i \in \Psi$ we define m_i as the number of remaining cache misses on the worst case path, counting from time θ. Similarly, also counting from θ, l_i is defined as the sum of each code segment and can thus be seen as the length of the task minus the time it spends using the bus or waiting for it (both

m_i and l_i are determined by the WCET analysis). Hence, if we define the constant k as the time it takes for the bus to process a cache miss, we get

$$\text{NWCET}_i^\theta = l_i + m_i k \tag{3}$$

As an example, consider Fig. 8(a) showing a task execution trace, in the case where no other tasks are competing for the bus. A black box represents the idle time, waiting for the transfer, due to a cache miss, to complete. In this example $m_1 = 4$ and $l_1 = \delta_1 + \delta_2 + \delta_3 + \delta_4 = 115$.

Let us now, with respect to the particular bus schedule, denote the average waiting time of task τ_i by d_i. That is, d_i is the average time task τ_i spends waiting, due to other processors owning the bus and the actual time of the transfer itself, every time a cache miss has to be transfered on the bus. Then, analogous to Equation 3, the WCET of task τ_i, counting from time θ, can be calculated as

$$\text{WCET}_i^\theta = l_i + m_i d_i \tag{4}$$

The dependency between a set of average waiting times d_i and a bus schedule can be modeled as follows. Consider the distribution P, defined as the set $p_1, \ldots, p_n$, where $\sum p_i = 1$. The value of p_i represents the fraction of bus bandwidth that, according to a particular bus schedule, belongs to the processor running task $\tau_i \in \Psi$. Given this model, the average waiting times can be rewritten as

$$d_i = \frac{1}{p_i} k \tag{5}$$

Putting Equations 2, 4, and 5 together and noting that Λ has been calculated as a maximum over all $\tau_i \in \Psi$, we can formulate the following system of inequalities:

$$\begin{gathered} \theta + l_1 + m_1 \frac{1}{p_1} k + \lambda_1 \le \theta + \Lambda + \Delta \\ \vdots \\ \theta + l_n + m_n \frac{1}{p_n} k + \lambda_n \le \theta + \Lambda + \Delta \\ p_1 + \cdots + p_n = 1 \end{gathered}$$

What we want is to find the bus bandwidth distribution P that results in the minimum Δ satisfying the above system. Unfortunately, solving this system is difficult due to its enormous solution space. However, an important observation that simplifies the process can be made, based on the fact that the slot distribution is represented by continuous variables p. Consider a configuration of $p_1, \ldots, p_n, \Delta$ satisfying the above system, and where at least one of the inequalities are not satisfied by equality. We say that the corresponding task τ_i is not on the critical path with respect to the schedule, meaning that its corresponding p_i can be decreased, causing τ_i to expand over time without affecting the global delay. Since the values of p must sum to 1, decreasing p_i, allows for increasing the percentage of the bus given to the tasks τ that are on the critical path. Even though the decrease might be infinitesimal, this makes the critical path shorter, and thus Δ is reduced. Consequently the smallest Δ that satisfies the system of inequalities is achieved when every inequality is satisfied by equality. As an example, consider Fig. 7(b) and note that τ_5 is an element in both sets D_3 and D_4 according to the definition in Section 5.1. This means that τ_5 is allowed to start first when both τ_3 and τ_4 have finished executing. Secondly, observe that τ_5 is on the critical path, thus being a direct contributor to the global delay. Therefore, to minimize the global delay, we must make τ_5 start as early as possible. In Fig. 7(b), the start time of τ_5 is defined by the finishing time of τ_4, which also is on the critical path. However, since there is a block of slack space between τ_3 and τ_5, we can reduce the execution time of τ_2 and thus make τ_4 finish earlier, by distributing more bus bandwidth to the corresponding processor. This will make the execution time of τ_1 longer (since it receives bus bandwidth), but as long as τ_3 ends before τ_4, the global delay will decrease. However, if τ_3 expands beyond the finishing point of τ_4, the former will now be on the critical path instead. Consequently, making task τ_3 and τ_4 end at the same time, by distributing the bus bandwidth such that the sizes of τ_1 and τ_2 are adjusted properly, will result in the earliest possible start time of τ_5, minimizing Δ. In this case the inequalities corresponding to both τ_1 and τ_2 are satisfied by equality. Such a distribution is illustrated in Fig. 7(c).

The resulting system consists of $n+1$ equations and $n+1$ variables ($p_1, \ldots, p_n$ and Δ), meaning that it has exactly one solution, and even though it is nonlinear, it is simple to solve. Using the resulting distribution, a corresponding initial TDMA bus schedule is calculated by setting the slot sizes to values proportional to P.

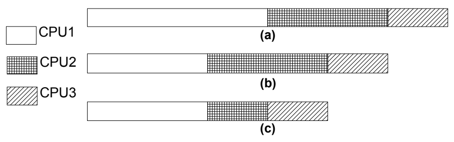


Figure 9. Calculating New Slot Sizes

5.2.3 Generating New Slot Size Candidates

One of the possible problems with the slot sizes defined as in Section 5.2.2 is the following: if one processor gets a very small share of the bus bandwidth, the slot sizes assigned to the other processors can become very large, possibly resulting in long wait times. By reducing the sizes of the larger slots while keeping, as much as possible, their mutual proportions, this problem can be avoided.

We illustrate the idea with an example. Consider a round consisting of three slots ordered as in Fig. 9(a). The slot sizes have been dimensioned according to a bus distribution $P = \{0.49, 0.33, 0.18\}$, calculated using the method in Section 5.2.2. The smallest slot, belonging to CPU 3, has been set to the minimum slot size k, and the remaining slot sizes are dimensioned proportionally [2] as multiples of k. Consequently, the initial slot sizes become $3k$, $2k$ and k. In order to generate the next set of candidate slot sizes, we define P' as the actual bus distribution of the generated round. Considering the actual slot sizes the bus distribution becomes $P' = \{0.50, 0.33, 0.17\}$. Since very large slots assigned to a certain processor can introduce long wait times for tasks running on other processors, we want to decrease the size of slots, but still keeping close to the proportions defined by the bus distribution P. Consider once again Fig. 9(a). Since, $p'_1 - p_1 > p'_2 - p_2 > p'_3 - p_3$, we conclude that slot 1 has the maximum deviation from its supposed value. Hence, as illustrated in Fig. 9(b), the size of slot 1 is decreased one unit. This slot size configuration corresponds to a new actual distribution $P' = \{0.40, 0.40, 0.20\}$. Now $p'_2 - p_2 > p'_3 - p_3 > p'_1 - p_1$, hence the size of slot 2 is decreased one unit and the result is shown in Fig. 9(c). Note that in the next iteration, $p'_3 - p_3 > p'_1 - p_1 > p'_2 - p_2$, but since slot 3 cannot be further decreased, we recalculate both P and P', now excluding this slot. The resulting sets are $P = \{0.60, 0.40\}$ and $P' = \{0.67, 0.33\}$, and hence slot 1 is decreased one unit. From now on, only slot 1 and 2 are considered, and the remaining procedure is carried out in exactly the same way as before. When this procedure is continued as above, all slot sizes will converge towards k which, of course, is not the desired result. Hence, after each iteration, the cost function is evaluated and the process is continued only until no improvement is registered for a specified number π of iterations. The best ever slot sizes (with respect to the cost function) are, finally, selected. Accepting a number of steps without improvement makes it possible to escape certain local minima (in our experiments we use $8 < \pi < 40$, depending on the number of processors).

5.2.4 Density Regions

A problem with the technique presented above is that it assumes that the cache misses are evenly distributed throughout the task. This, obviously, is not the case. A solution to this problem is to analyze the internal cache miss structure of the actual task and, accordingly, divide the worst case path into disjunct intervals, so called *density regions*. A *density region* is defined as an interval of the path where the distance between consecutive cache misses (δ in Fig. 8) differs only by a specified percentage. In this context, if we denote by α the average time between two consecutive cache misses (inside a region), the density of a region is defined as $\frac{1}{\alpha+1}$. A region with high density, close to 1, has very frequent cache misses, while the opposite holds for a low-density region.

Consequently, for each task $\tau_i \in \Psi$, we identify the next density region. Now, instead of constructing a bus schedule with respect to each entire task $\tau_i \in \Psi$, only the interval $[\theta..\Theta_i)$ is considered, with Θ_i representing the end of the density region. We call this interval of the task a subtask since it will be treated as a task of its own. Fig. 8(b) shows a task τ_2 with two density regions, the first one corresponding to the subtask τ'_2. The tail of τ'_2 is calculated as $\lambda'_2 = \lambda''_2 + \lambda_2$, with λ''_2 being defined as the NWCET of τ_2 counting from Θ_2. Furthermore, in this particular example $m'_2 = 3$ and $l'_2 = \delta_1 + \delta_2 + \delta_3 = 87$.

Analogous to the case where entire tasks are analyzed, when a partial bus segment schedule has been decided, θ' will be set to the finish time of the first subtask. Just as before, the entire procedure is then repeated for $\theta = \theta'$.

5.3 The BSA3 Approach

The bus scheduling policy BSA3 is obviously a special case of the BSA2 approach. However, due to the limitation that all slot sizes of a round now have the same size, our previous framework for calculating the distribution P makes little sense. Therefore, we propose a simpler, but quality-wise equally efficient algorithm, tailormade for the BSA3 approach. The slot ordering mechanisms are still the same as for BSA2, but the procedures for calculating the slot sizes are now vastly simplified.

```
1. Initialize the slot sizes to the minimum
   size $k$.
2. Calculate an initial slot order.
3. Analyze the WCET of each task $\tau \in \Psi$ and
   evaluate the result according to the cost
   function.
4. Generate a new slot order candidate and
   repeat from 3 until all candidates are
   evaluated.
5. Increase the slot sizes one step.
6. If no improvements were achieved during a
   specified number of iterations then exit.
   Otherwise repeat from 2.
7. The best configuration according to the
   cost function is then used.
```

Algorithm 2: The BSA3 Approach

5.4 Memory Consumption

As stated before, both BSA2 and BSA3 in their standard form do not require excessive memory on the bus arbiter. The critical factor is the number of different segments that has to be stored in memory. In order to calculate an upper bound on the number of segments needed, we make the observation that a new segment is created at every time t when

[2] While slot sizes, in theory, do not have to be multiples of the minimum slot size k, in practice this is preferred as it avoids introducing unnecessary slack on the bus.

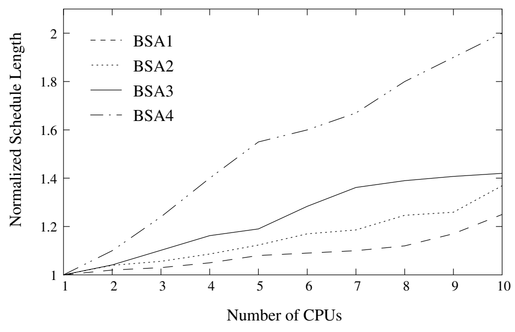


Figure 10. The Four Bus Access Policies

at least one task starts or finishes. For the case when density regions are not used, these are also the only times when a new segment will be created. Hence, an upper bound on the number of segments is $2|\Pi|$, where Π is the set of all tasks as defined in Section 2.

When using density regions, the start and finish of every region can result in a new segment each. Therefore, tasks divided into very many small density regions will result in bus schedules consuming more memory. A straightforward solution is to limit, according to the available controller memory, the minimum size of a density region. For instance, if the minimum density region size for a task τ_i is $x\%$ of the task length l_i as defined above, the number of generated segments becomes at most $2|\Pi|\frac{100}{x}$.

6 Experimental Results

The complete flow illustrated in Fig. 4 has been implemented and used as a platform for the experiments presented in this section. They were run on a dual core Pentium 4 processor at 2.8 GHz. For the WCET analysis, it was assumed that ARM7 processors are used. We have assumed that 12 clock cycles are required for a memory access due to a cache miss.

First, we have performed experiments using a set of benchmarks consisting of task graphs in which the individual tasks are corresponding to CFGs extracted from various C programs (e.g. sorting, searching, matrix multiplications, DSP algorithms). The actual task graphs were randomly generated with the number of tasks varying between 50 and 200.

We have explored the efficiency of the proposed approach in the context of the four bus scheduling approaches introduced in Section 2. We have run experiments for configurations consisting of 1, 2, 3, ..., 10 processors. For each configuration, 50 randomly generated task graphs were used. For each task graph, the worst-case schedule length has been determined in 5 cases: the four bus scheduling policies BSA1 to BSA4, and a hypothetical ideal situation in which memory accesses are never delayed. This ideal schedule length (which in practice, is unachievable, even by a theoretically optimal bus schedule) is considered as the baseline for the diagrams presented in Fig. 10. The diagram corresponding to each bus scheduling alternative indicates how many times larger the obtained bus schedule is relative to the ideal length. The diagrams correspond to the average obtained for the 50 graphs considered for each configuration.

A first conclusion is that BSA1 produces the shortest delays. This is not unexpected, since it is based on highly customized bus schedules. It can be noticed, however, that the approaches BSA2 and BSA3 are producing results that are close to those produced by BSA1, but with a much lower cost in controller complexity. It is not surprising that BSA4, which restricts very much the freedom for bus optimization, produces very low quality results.

In a second set of experiments we have focused on the comparison of the BSA2 and BSA3 policies, which are those of practical importance. In particular, we were interested in the efficiency of these policies for applications with different cache miss patterns. For these experiments we have considered task graphs consisting of 20 tasks running on 5 processors. A particular cache miss pattern, in this context, is defined by its standard deviation of the interval between the consecutive cache misses inside the task (the average distance between cache misses was 73 clock cycles). Thus, we have generated 30 task graphs with uniform distribution of the cache misses (standard deviation is 0). We also generated 30 task graphs with moderate irregularity in the distribution of cache misses (standard deviation is 50) and 30 task graphs with high irregularity (standard deviation is 150). The results in Fig. 11 show the average normalized schedule length obtained with the two approaches, relative to the ideal (practically unachievable) schedule length. It is no surprise that in case of perfect uniformity the two approaches produce identical schedule lengths. However, as the irregularity of the cache misses increases, the potential advantages of BSA2 become visible.

With a third set of experiments, we have explored the efficiency of the successive steps of the bus access optimization algorithm for BSA2 presented in section 5. We have run the experiments using the same task graphs as before, with the same cache miss patterns. The results in Fig. 12 are illustrating the normalized schedule lengths obtained after successive steps of the algorithm: with initial slot sizes (ISS, section 5.2.2), after slot size adjustment (SSA, section 5.2.3), and, finally, after the application of density regions (DS, section 5.2.4). The schedule lengths are, again, normalized relative to the length of the ideal schedule. As expected, applying density regions is not useful in the case of uniform cache miss distributions. Its efficiency increases with increased degree of irregularity.

The execution times for the whole flow, in the case of the examples consisting of 100 tasks on 10 processors are 120 min. for BSA2 and 5 min. for BSA3.

In order to validate the real-world applicability of this

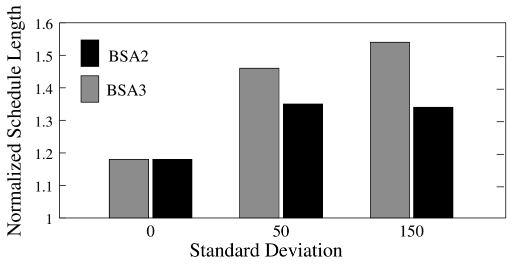


Figure 11. Comparison BSA2 vs. BSA3

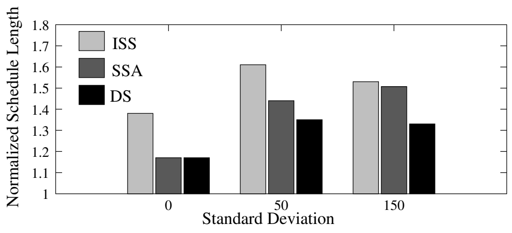


Figure 12. BSA2 Optimization Steps

approach we have analyzed a smart phone. It consists of a GSM encoder, GSM decoder and an MP3 decoder, that were mapped on 4 ARM7 processors (the GSM encoder and decoder are mapped each one on a processor, while the MP3 decoder is mapped on two processors). The software applications have been partitioned into 64 tasks. The size of one task is between 1304 and 70 lines of C code in case of the GSM codec and between 2035 and 200 lines in case of the MP3 decoder. We have assumed a 4-way set associative instruction cache with a size of 4KB and a direct mapped data cache of the same size. The results of the analysis are presented in table 1, where the deviation of the schedule length from the ideal one is presented for each bus scheduling approach.

BSA1	BSA2	BSA3	BSA4
1.17	1.33	1.31	1.62

Table 1. Results for the Smart Phone

7 Conclusions

In this paper, we have presented an approach to the implementation of predictable RT applications on multiprocessor SoCs, which takes into consideration potential conflicts between parallel tasks for memory access. We have primarily focused on the issue of bus access optimization which is of crucial importance for achieving predictability and, at the same time, performance. Experiments have demonstrated the efficiency of the proposed optimization heuristics.

References

[1] A. Andrei, P. Eles, J. Rosén, and Z. Peng. Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *submitted*, 2007.

[2] A. Andrei, P. Eles, J. Rosén, and Z. Peng. Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *Tech. Rep., Linkoping Univ.*, 2007.

[3] D. Bertozzi, A. Guerri, M. Milano, F. Poletti, and M. Ruggiero. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip. In *DATE*, pages 3–8, 2006.

[4] K. Goossens, J. Dielissen, and A. Radulescu. AEthereal Network on Chip: Concepts, Architectures, and Implementations. *IEEE Design & Test of Computers*, 2/3:115–127, 2005.

[5] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

[6] E.G. Coffman Jr and R.L. Graham. Optimal Scheduling for two processor systems. *Acta Inform.*, 1:200–213, 1972.

[7] I. A. Khatib, D. Bertozzi, F. Poletti, L. Benini, and et.all. A multiprocessor systems-on-chip for real-time biomedical monitoring and analysis: Architectural design space exploration. In *DAC*, pages 125–131, 2006.

[8] H. Kopetz. *Real-Time Systems-Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

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

[10] S. Pasricha, N. Dutt, and M. Ben-Romdhane. Fast exploration of bus-based on-chip communication architectures. In *CODES+ISSS*, pages 242–247, 2004.

[11] P. Pop, P. Eles, Z. Peng, and T. Pop. Analysis and Optimization of Distributed Real-Time Embedded Systems. *ACM Transactions on Design Automation of Electronic Systems*, Vol. 11:593–625, 2006.

[12] P. Puschner and A. Burns. A Review of Worst-Case Execution-Time Analysis. *Real-Time Systems*, 2/3:115–127, 2000.

[13] H. Ramaprasad and F. Mueller. Bounding Preemption Delay within Data Cache Reference Patterns for Real-Time Tasks. In *Real-Time and Embedded Technology and Applications Symposium*, pages 71–80, 2005.

[14] E. Salminen, V. Lahtinen, K. Kuusilinna, and T. Hamalainen. Overview of bus-based system-on-chip interconnections. In *ISCAS*, pages 372–375, 2002.

[15] S. Schliecker, M. Ivers, and R. Ernst. Integrated analysis of communicating tasks in mpsocs. In *CODES+ISSS*, pages 288–293, 2006.

[16] J. Staschulat and R. Ernst. Worst case timing analysis of input dependent data cache behavior. In *ECRTS*, 2006.

[17] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *ECRTS*, pages 41–48, 2005.

[18] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Separated Cache and Path Analysis. *Real-Time Systems*, 18(2/3):157–179, 2000.

[19] L. Thiele and R. Wilhelm. Design for Timing Predictability. *Real-Time Systems*, 28(2/3):157–177, 2004.

[20] F. Wolf, J. Staschulat, and R. Ernst. Associative caches in formal software timing analysis. In *DAC*, pages 622–627, 2002.

[21] W. Wolf. *Computers as Components: Principles of Embedded Computing System Design*. Morgan Kaufman Publishers, 2005.