21st International Conference on VLSI Design

Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip

Alexandru Andrei, Petru Eles, Zebo Peng, Jakob Rosen
Deptartment of Computer and Information Science
Linköping University, Linköping, S-58183, Sweden

***Abstract*— Worst-case execution time (WCET) analysis and, in general, the predictability of real-time applications implemented on multiprocessor systems has been addressed only in very restrictive and particular contexts. One important aspect that makes the analysis difficult is the estimation of the system's communication behavior. 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. As opposed to the analysis performed for a single processor system, where the cache miss penalty is 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 context, we propose, for the first time, an approach to worst-case execution time analysis and system scheduling for real-time applications implemented on multiprocessor SoC architectures.**

I. Introduction and Related Work

We are facing an unprecedented development in the complexity and sophistication of embedded systems. Emerging technologies provide the potential to produce complex multiprocessor architectures implemented on a single chip [24]. Embedded applications, running on such highly parallel architectures are becoming more and more sophisticated and, at the same time, will be used very often in applications for which predictability is very important. 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 [5]. Such applications impose strict constraints not only in terms of their logical functionality but also with concern to timing. The objective of this paper is to address, at the system-level, the specific issue of predictability for embedded systems implemented on current and future multiprocessor architectures. 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 [10], [14]. While WCET analysis has been an investigation topic for already a long time, the basic driving force of this research has been, and still is, to improve the tightness of the analysis and to incorporate more and more features of modern processor architectures. 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 [22]. 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.

The main problems that researchers have tried to solve are (1) the identification of the possible execution sequences inside a task and (2) the characterization of the time needed to execute each individual action [15]. With advanced processor architectures, effects due to caches, pipelines, and branch prediction have to be considered in order to determine the execution time of individual actions. There have been attempts to model both problems as a single ILP formulation [11]. Other approaches combine abstract interpretation for cache and pipeline analysis with ILP formulations for path analysis [21], or even integrate simulation into the WCET analysis flow [12], [23]. There have been attempts to build modular WCET estimation frameworks where the particular subproblems are handled separately [4], while other approaches advocate a more integrated view [7]. More recently, preemption related cache effects have also been taken into consideration [16], [20].

The basic assumption in all this research is 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, [20]). 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 [22] 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. In this paper we present the first approach in this direction.

A framework for system level task mapping and scheduling for a similar type of platforms has been presented in [2]. 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,

1063-9667/08 $25.00 © 2008 IEEE
DOI 10.1109/VLSI.2008.33

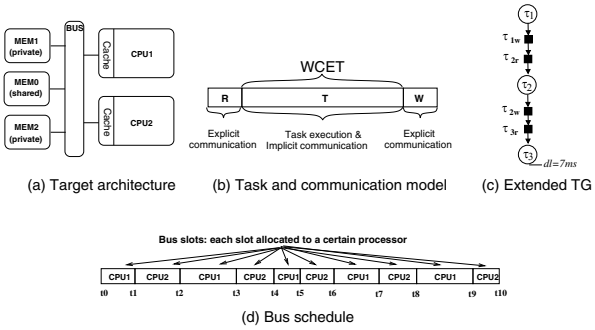

Fig. 1. System and task models

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 architecture model are given in Section II. Section III outlines the problem with a motivational example and is followed in Section IV by the description of our proposed solution. In Section V we present the approach used for the worst-case execution time analysis. The bus access policy is presented in Section VI. Experimental results are given in Section VII.

II. System and Application Model

In this paper we consider multiprocessor system-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 certain 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, [9]. 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).

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 computational tasks are annotated with deadlines dl_i 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 via the shared memory. Consequently, a message exchanged via the shared memory assumes two explicit communications: one for writing into the shared memory (by the sending task) and the other for reading from the memory (by the receiving task). Explicit communication is modeled in the task graph as two communication tasks, executed by the sending and the receiving 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. The reads and writes to and from the private memories are cached. Whenever a cache miss occurs, the data has to be fetched from the memory and a cache line replaced. This 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 [18]. We will show that this leads to incorrect results in the context of multiprocessor systems.

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, [17], [13], [5].

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. 1(d). 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. An access from the arbiter to its local memory does not generate traffic on the system bus. The bus schedule is given as input to the WCET analysis algorithm. When the application is running, 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.

III. 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 has a deadline of 60 time units. Task τ_2 runs on CPU_2. When τ_2 finishes, it updates the shared memory during the explicit communication E_1. We have illustrated this example system in Fig. 2(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. 2 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

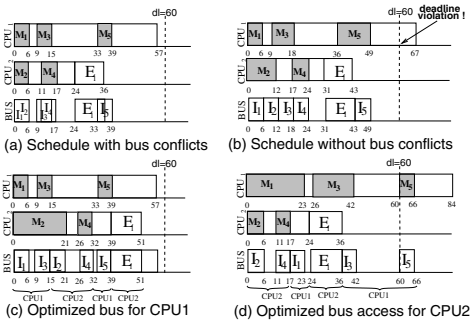

Fig. 2. Schedule with and without bus conflicts

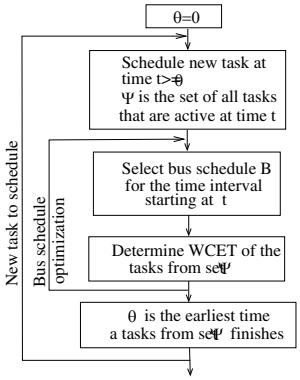

Fig. 3. Overall Approach

bus transfers I_1,I_2,I_3,I_4 and I_5. If we analyze the two tasks using a classical worst-case analysis algorithm, we conclude that task 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. This scenario is illustrated in Fig. 2(a).

A closer look at the execution pattern of the tasks reveals the fact that the cache misses may overlap in time. For example, the cache miss I_1 and I_2 are both happening at time 0 (when the tasks start, the cache is empty). 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. 2(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. 2(a), where an ideal, constant, cache miss penalty is assumed, due to the resource conflicts, 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. 2(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. 2(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 II. Let us assume the bus schedule as in Fig. 2(c). According to this bus schedule, the processor CPU_1 is granted the bus at time 0 for an interval of 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 for its cache misses at times 15 and 26 and finishes at time 31, resulting in a longer execution time than in the ideal case (time 24). The explicit communication message E_1 is started at time 39 and completes at time 51.

While the bus schedule in Fig. 2(c) is optimized according to the requirements from task τ_1, the one in Fig. 2(d) eliminates all bus access delays for task τ_2. According to this bus schedule, while τ_2 will finish earlier than in Fig. 2(c), task τ_1 will finish at time 84 and, thus, miss its deadline.

The examples presented 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.

IV. Analysis, Scheduling and Optimization Flow

We consider as input the application task graph capturing the dependencies between the tasks and the target hardware platform. Each task has associated the corresponding code and potentially a deadline that has to be met at runtime. As a first stage, mapping of the tasks to processors is performed. 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

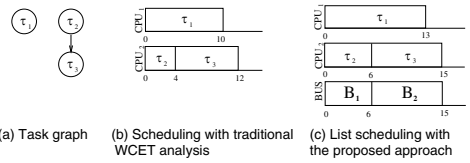

Fig. 4. System level scheduling with WCET analysis

principle, unpredictable in the context of the potential bus conflicts between the processors that run in parallel. 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. This cyclic dependency is not just a technical detail or inconvenience, but a fundamental issue with large implications and which invalidates one of the basic assumptions that support current state of the art. 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 II. The actual bus access schedule is determined at design time and will be enforced during the execution of the application.

2) The bus access schedule is taken into consideration at the WCET estimation. WCET estimation, as well as the determination of the bus access schedule are integrated with the system level scheduling process (Fig. 3).

We will present our overall strategy using a simple example. It consists of three tasks mapped on two processors, as in Fig. 4.

The system level static cyclic scheduling process is based on a list scheduling technique [8]. List scheduling heuristics are based on priority lists from which tasks are extracted in order to be scheduled at certain moments. A task is placed in the ready list if all its predecessors have been already scheduled. All ready tasks from the list are investigated, and that task τ_i is selected for placement in the schedule which has the highest priority. We use the modified partial critical path priority function presented in [14]. The process continues until the ready list is empty.

Let us assume that, using 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. 4(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. This configuration should, preferably, be fixed so that it is favorable from the point of view of WCETs of the currently running tasks (τ_1 and τ_2, in our case). Given a certain bus configuration, our WCET-analysis will determine the WCET for τ_1 and τ_2. Inside an optimization loop, several alternative bus configurations are considered. The goal is to reduce the WCET of τ_1 and τ_2, with an additional weight on reducing the WCET of that task that is assumed to be on the critical path (in our case τ_2).

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. Given a certain bus configuration B, 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 9 for τ_3 and 13 for τ_1. The final schedule is illustrated in figure 4.

The overall approach is illustrated in Fig. 3. At each iteration, the set Ψ of tasks that are active at the current time t, is considered. In an inner optimization loop a bus configuration B is fixed. For each candidate configuration the WCET of the tasks in the set Ψ is determined. During the WCET estimation process, the bus 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 bus 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 $(t, \theta]$, 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 II, a message exchanged via the shared memory assumes two explicit communications: one for writing into the shared memory (by the sending task) and the other for reading from the memory (by the receiving task). Explicit communication is modeled in the task graph as two communication tasks, executed by the sending and the receiving processor, respectively (Fig. 1 in section II). 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 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

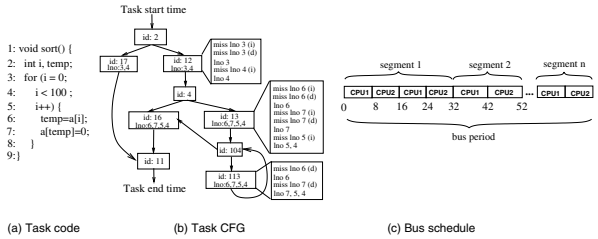

Fig. 5. Example task WCET calculation

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 [1] 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.

In the following sections we will address two aspects: the WCET estimation technique and bus access policy.

V. WCET Analysis

We will present 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 [23], [19] 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. 5(a) and (b), we have depicted an example task containing a for loop and the corresponding CFG, extracted from this task. 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. Intuitively, when executing a loop, at the first iteration all the instruction accesses result in cache misses. However, during the next iterations there is a chance to find the instructions in the cache.

We have depicted in Fig. 5(b) the resulting misses obtained after performing instruction (marked with an "i") and data (marked with an "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.

Let us consider now the data accesses. While the instruction addresses are always known, this is not the case with the data [19], [16]. This, for example, is the case with an array that is accessed using an index variable whose value is data dependent, as in Fig.5(a), on line 7. Using data dependency analysis performed on the abstract syntax tree extracted from the code of the task, all the data accesses are classified as predictable or unpredictable [19]. For example, in Fig.5(a) the only unpredictable data memory access is in line 7. The rest of the accesses are predictable. All the unpredictable memory accesses are classified as cache misses. Furthermore, they have a hidden impact on the state of the data cache. A miss resulted from an unpredictable access replaces an unknown cache line. One of the following predictable memory accesses that would be considered as hit otherwise, might result in a miss due to the unknown line replacement. Similar to the instruction cache, dataflow analysis is used for propagating the addresses that will be in the cache, no matter which program path is executed.

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, [19], [16], [11].

In a realistic multiprocessor setting, however, due to the variation of the miss penalties as a 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 and the worst-case duration of computation sequences between the misses. For example, in the case of node 13 in Fig. 5(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. 5(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.

The start time of the task that is currently analyzed is decided during the system level scheduling (see section IV) 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. 5(c), the node 12 will start its execution at time 0 and finish at time 39. The instruction miss (marked with "i" in Fig. 5(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.

In the following we will illustrate the algorithm that performs the WCET computation for a certain task. The algorithm must find the longest path in the control flow graph. For example, there are four possible execution paths (sequences of nodes) for the task in Fig. 5(a) that are captured by the CFG in Fig. 5(b):(1) 2, 17, 11, (2) 2, 12, ,4, 16, 11, (3) 2, 12, 4, 13, 104, 16, 11 and (4) 2, 12, 4, 13, 104, 113, 104, ..., 104, 16, 11. The execution time of a particular node in the CFG can be computed only after the execution times of all its predecessors are known. For example, the execution time of node 16 can be computed only after the execution time for the nodes 4 and 104 is fixed. At this point it is interesting to note that the node 104 is the entry in a CFG loop (104, 113, 104). Due to the fact that the cache miss penalties depend on

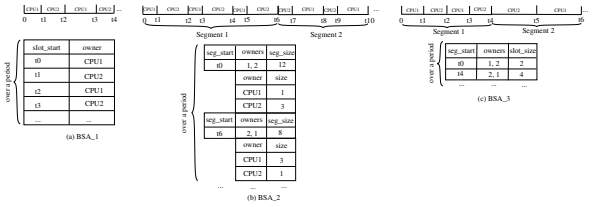

Fig. 6. Bus Schedule Table (system with two CPUs)

the bus schedule, the execution times of the loop nodes will be different at each loop iteration. Thus, loop nodes in the CFG must be visited a number of times, given by the loop bound (extracted automatically or annotated in the code). In the example from Fig. 5(b), the node 113 is visited 99 times (the loop is executed 100 times, but it was unrolled once). Each time a loop node is visited, its start time is updated and a new end time is calculated using the bus schedule, in the manner illustrated above for node 12. Consequently, during the computation of the execution time of the node 16, the start time is the maximum between the end time of the node 4 and node 104, obtained after 99 iterations. The worst-case execution time of the task will be the end time of the node 11.

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 VII show.[1]

VI. Bus Schedule

The approach described in section V relies on the fact that during a time slot, only the processor that owns the bus must be granted the access. The bus arbiter must take care that the bus requests are granted according to the bus schedule table.

The assignment of slots to processors is captured by the bus schedule table and has a strong influence on the worst-case execution time. Ideally, from the point of view of task execution times, 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. 6(a) for a system with two CPUs. This bus scheduling approach, denoted as BSA_1, would offer the best task WCETs at the expense of a very complex bus slot optimization algorithm and of a large schedule table.

Alternatively, in order to reduce the controller complexity, the bus schedule is divided in segments. Such a segment is an interval in which the bus schedule follows a regular pattern. This pattern concerns both slot order and size. In Fig. 6(b) we illustrate a schedule consisting of two bus segments. This bus scheduling approach is denoted BSA_2.

The approach presented in Fig. 6(c) and denoted BSA_3 further reduces the memory needs for the bus controller. As opposed to BSA_2, in this case, all slots inside a segment have the same size.

[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.

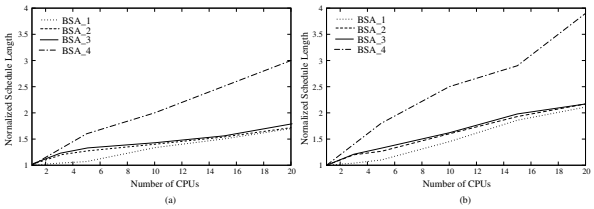

Fig. 7. Experimental results

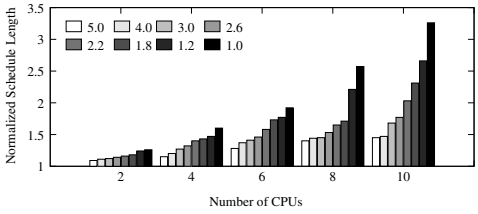

Fig. 8. Experimental results

In the final approach, BSA_4, all the slots in the bus have the same size and repeated according to a fix sequence.

The bus schedule is a key parameter that influences the worst-case execution time of the tasks. As shown in Section IV the bus schedule is determined during the system scheduling process. Referring to Fig. 3, successive portions of the bus schedule are fixed during the internal optimization loop. The aim is to find a schedule for each portion, such that globally the worst-case execution time is minimized. In order to find an efficient bus schedule for each portion, information produced by the WCET analysis is used in the optimization process. In particular, this information captures the distribution of the cache misses along the detected worst case paths, for each currently active task (for each task in set Ψ). We have deployed several bus access optimization algorithms, specific to the proposed bus schedule alternative (BSA_1, BSA_2, BSA_3, BSA_4).

In the case of BSA_1, each portion of the bus schedule (fixed during the internal optimization loop in Fig.3) is determined without any restriction. For BSA_2 and BSA_3, each portion of the bus schedule corresponds to a new bus segment, as defined above. In case of BSA_2, the optimization has to find, for each segment, the size of the slots allocated for each processor, as well as their order. The search space for BSA_3 is reduced to finding for each bus segment, a unique slot size and an order in which the processors will access the bus.

In the case of BSA_4, the slot sequence and size is unique for the whole schedule. Therefore, the scheme in Fig. 3 is changed: a bus configuration alternative is determined before system scheduling and the list scheduling loop is included inside the bus optimization loop.

The bus schedule optimization algorithms are based on simulated annealing. Due to space limitations, the actual algorithms are not presented here. They will be described in a separate paper. Nevertheless, their efficiency is demonstrated by the experimental results presented in the next section.

VII. Experimental Results

The complete flow illustrated in Fig. 3 has been implemented and used as a platform for the experiments presented in this section. They were run on a dual core Pentium4 processor at 2.8 GHz.

First, we have performed experiments using a set of synthetic benchmarks consisting of random task graphs with the number of tasks varying between 50 and 200. The tasks are mapped on architectures consisting of 2 to 20 processors. The tasks are corresponding to CFGs extracted from various C programs (e.g. sorting, searching, matrix multiplications, DSP algorithms). 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, once the bus access has been granted.

We have explored the efficiency of the proposed approach in the context of the four bus scheduling approaches introduced in Section VI. The results are presented in Fig. 7. We have run experiments for configurations consisting of 2, 4, 6, ... 20 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 BSA_1 to BSA_4, 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. 7. 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 BSA_1 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 BSA_2 and BSA_3 are producing results that are close to those produced by BSA_1, but with a much lower cost in controller complexity. It is not surprising that BSA_4, which restricts very much the freedom for bus optimization, produces very low quality results.

The actual bus load is growing with the number of processors and, implicitly, that of simultaneously active tasks. Therefore, the delays at low bus load (smaller number of processors) are close to the ideal ones. The deviation from the ideal schedule length is growing with the increased bus load due to the inherent delays in bus access. This phenomenon is confirmed also by the comparison between the diagrams in Fig. 7(a) and (b). The diagrams in Fig. 7(b) were obtained considering a bus load that is 1.5 times higher (bus speed 1.5 times smaller) than in Fig. 7(a). It can be observed that the deviation of schedule delays from the ideal one is growing faster in Fig. 7(b).

The execution times for the whole flow, in the case of the largest examples (consisting of 200 tasks on 20 processors) are as follows: 125 min. for BSA_1, 117 min. for BSA_2, 47 min. for BSA_3, and 8 min. for BSA_1.

The amount of memory accesses relative to the computations has a strong influence on the worst case execution time.

We have performed another set of experiments in order to asses this issue. The results are presented in Fig. 8. We have run experiments for configurations consisting of 2, 4, 6, ... 10 processors. For each configuration, 50 randomly generated task graphs were used. For each task graph, we have varied the ratio of clock cycles spent during computations and memory accesses. We have used eight different ratios: 5.0, 4.0, 3.0, 2.6, 2.2, 1.8, 1.4, 1.0. A ratio of 3.0 means that the number of clock cycles spent by the processors performing computations (assuming that all the memory accesses are cache hits) is three time higher than the number of cache misses multiplied with the cache miss penalty (assuming that each cache miss is handled in constant time, as if there are no conflicts on the bus). So, for example, if a task spends on the worst case CFG path 300000 clock cycles for computation and 100000 cycles for memory accesses due to cache misses (excluding the waiting time for bus access), the ratio will be 3.0. During this set of experiments we have assumed that the bus is scheduled according to the BSA_3 policy. Similar to the previous experiments from Fig. 7, the ideal schedule length is considered as the baseline for the diagrams presented in Fig. 8. Each bar indicates how many times larger the caculated worst case execution is relative to the ideal length. For example, on an a architecture with six processors and a ratio of 5.0, the worst case execution time is 1.28 times higher than the ideal one. Using the same architecture but with a smaller ratio (this means that the application is more memory intensive), the deviation increases: for a ratio of 3.0, the worst case execution time is 1.41 times the ideal one, while if the ratio is 1.0 the worst case execution time is 1.92 times higher than the ideal one.

In order to validate the real-world applicability of this approach we have analyzed a smart phone. It consists of a GSM encoder, GSM decoder [3] and an MP3 decoder [6], 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 I, where the deviation of the schedule length from the ideal one is presented for each bus scheduling approach.

BSA_1	BSA_2	BSA_3	BSA_4
1.17	1.33	1.31	1.62

TABLE I
RESULTS FOR THE SMART PHONE

VIII. CONCLUSIONS

In this paper, we have presented the first approach to the implementation of predictable RT applications on multiprocessor SoCs, which takes into consideration potential conflicts between parallel tasks for memory access. The approach comprizes WCET estimation and bus access optimization in the global context of system level scheduling. Experiments have shown the efficiency of the approach.

REFERENCES

[1] A. Andrei. *Energy Efficient and Predictable Design of Real-Time Embedded Systems*. PhD thesis, Linkoping University, Sweden, 2007.

[2] 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.

[3] Jutta Degener and Carsten Bormann. GSM 06.10 lossy speech compression. Source code available at http://kbs.cs.tu-berlin.de/∼jutta/toast.html.

[4] J. Engblom, A. Ermedahl, M. Sjodin, J. Gustafsson, and H. Hansson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer*, 4(4):437–455, 2003.

[5] 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.

[6] Johan Hagman. mpeg3play-0.9.6. Source code available at http://home.swipnet.se/∼w-10694/tars/mpeg3play-0.9.6-x86.tar.gz.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] T. Lundqvist and P. Stenstrom. An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution. *Real-Time Systems*, 17(2/3):183–207, 1999.

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

[14] 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.

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

[16] 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.

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

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

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

[20] 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.

[21] 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.

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

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

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