Predictable Worst-Case Execution Time Analysis for Multiprocessor Systems-on-Chip

Jakob Rosén[1], Petru Eles[1], Zebo Peng[1] and Alexandru Andrei[2]

[1]IDA, Linköping University
SE-58183 Linköping
Sweden
{jakob.rosen, petru.eles, zebo.peng}@liu.se

[2]Ericsson
SE-58330 Linköping
Sweden
alexandru.andrei@ericsson.com

***Abstract*—Worst-case execution time analysis is the fundament of real-time system design, and is therefore an area which has been subject to great scientific interest for a long time. However, traditional worst-case execution time analysis techniques assume that the underlying hardware is a monoprocessor system, and this class of hardware platforms is getting less suitable for modern embedded applications, which demand more and more in terms of computational power. For multiprocessor systems, traditional worst-case analysis tools do not produce correct results and can consequently not be used. To solve this problem, we have previously proposed a technique for achieving predictability on multiprocessor systems-on-chip using a shared TDMA bus. One of the main benefits with our approach is that existing, traditional worst-case execution time analysis techniques can, after some small modifications, be applied. In this paper, we describe the nature of these modifications and how to handle different types of multiprocessor architectures.**

I. Introduction and Related Work

For real-time systems, correctness of a program not only depends on the produced computational results, but also on its ability to deliver these on time to satisfy specified time constraints. Therefore, for a real-time application, predictability with respect to time is of uttermost importance. The obvious example is safety-critical hard real-time systems, such as medical and avionic applications, for which failure to meet a specified deadline not only renders the computations useless, but also can have catastrophic consequences. However, predictability is getting more and more desirable for other classes of embedded applications, for instance within the domains of multimedia and telecommunication, for which QoS guarantees are desired [1]. As these kinds of applications grow more and more complex, they also require more computational power in terms of hardware resources. In order to satisfy these demands, multicore systems implemented on a single chip are used to an increasing extent [2].

To achieve predictability with respect to time, schedulability analysis techniques are applied, assuming that the worst-case execution time (WCET) of every task is known. A lot of research has been carried out within the area of worst-case execution time analysis [3]. However, each task is, traditionally, analyzed in isolation as if it was running on a monoprocessor system. Consequently, it is assumed that memory accesses over the bus take constant amount of time to process. For multiprocessor systems with a shared communication infrastructure, however, transfer times depend on the bus load and are therefore no longer constant, causing the traditional methods to produce incorrect results [4].

The main obstacle when performing WCET analysis on multiprocessor systems is that the scheduling of tasks assumes that their worst-case execution times are known, but to calculate these worst-case execution times, knowledge about the task schedule is required. Clearly, the traditional method of separating WCET analysis and task scheduling no longer works, and new approaches are required. We have previously proposed a novel technique to achieve predictability on multiprocessor systems by doing worst-case execution time analysis and scheduling simultaneously [5], [6]. With respect to a given TDMA bus schedule, tasks are scheduled at the same time as their worst-case execution times are calculated, and the resulting worst-case global delay of the application is obtained. In order to calculate the WCET of a task, the analysis needs to be aware of the TDMA bus, taking into account that processors must only be granted the bus during their assigned time slots. To accomplish this, the analysis procedure must be modified.

II. System Model

A. Hardware Architecture

As hardware platform, we have considered a multiprocessor system-on-chip architecture with a shared communication infrastructure, as shown in Figure 1, typical for the new generation of multiprocessor system-on-chip designs [7]. Each processor has its own cache for storing data and instructions, and is connected to a private memory via the bus. For interprocessor communication, a shared memory is used. All memory accesses to the private memories are cached, as opposed to accesses to the shared memory which, in order to avoid cache coherence problems, are not cached. All memory devices are accessed using the same, shared bus. However, in the case of private memory accesses, the bus is used only when an access results in a cache miss.

Within the context of worst-case execution time analysis, hardware platforms can be divided into compositional archi-

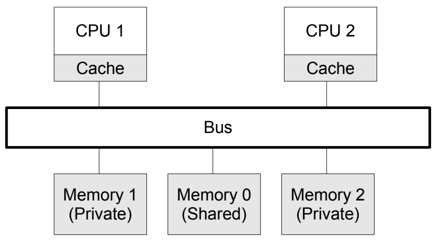


Figure 1. Hardware model

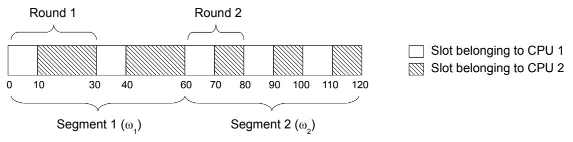


Figure 2. Example of a bus schedule

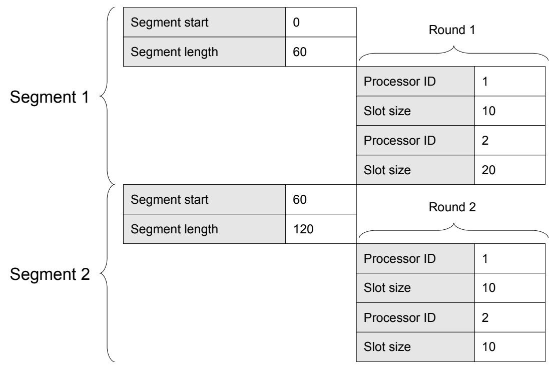
tectures and noncompositional architectures [8], depending on whether or not the platform exhibits timing anomalies [9], [10]. Timing anomalies occur when a local worst-case scenario, such as a cache miss instead of a hit, does not result in the worst case globally. This complicates the worst-case execution time analysis significantly, since no local assumptions can be made. Compositional architectures, such as the ARM7, do not exhibit timing anomalies, and the analysis can therefore be divided into disjunctive subproblems, simplifying the analysis procedure. Noncompositional architectures, on the other hand, require a far more complicated and time-consuming analysis. The PowerPC 775 is an example of a noncompositional architecture [8]. As will be described below, our approach works for both compositional architectures and noncompositional architectures.

B. Bus Model

A precondition for achieving predictability is to use a predictable bus architecture. Therefore, we are using a TDMA-based bus arbitration policy, which is suitable for modern system-on-chip designs with QoS constraints [11], [12], [1].

The behavior of the bus arbiter is defined by the *bus schedule*, consisting of sequences of slots. Each slot is owned by exactly one processor, and has an associated start time and an end time. Between these two time instants, only the processor owning the slot is allowed to use the bus. A bus schedule is divided into *segments*, and each segment consists of a *round*, that is, a sequence of slots, that is repeated periodically. See Figure 2 for an example.

The bus arbiter stores the bus schedule in a dedicated external memory, and grants access to the processors accordingly. If processor CPU$_i$ requests access to the bus in a time interval belonging to a slot owned by a different processor, the transfer will be delayed until the start of the next slot with owner CPU$_i$. A bus schedule is defined for one period of the application, and is then repeated periodically. A table representation of the bus schedule in Figure 2 can be found


Segment 1
Segment 2
Round 1
Round 2

Segment start	0
Segment length	60
Processor ID	1
Slot size	10
Processor ID	2
Slot size	20
Segment start	60
Segment length	120
Processor ID	1
Slot size	10
Processor ID	2
Slot size	10

Figure 3. Bus schedule table representation

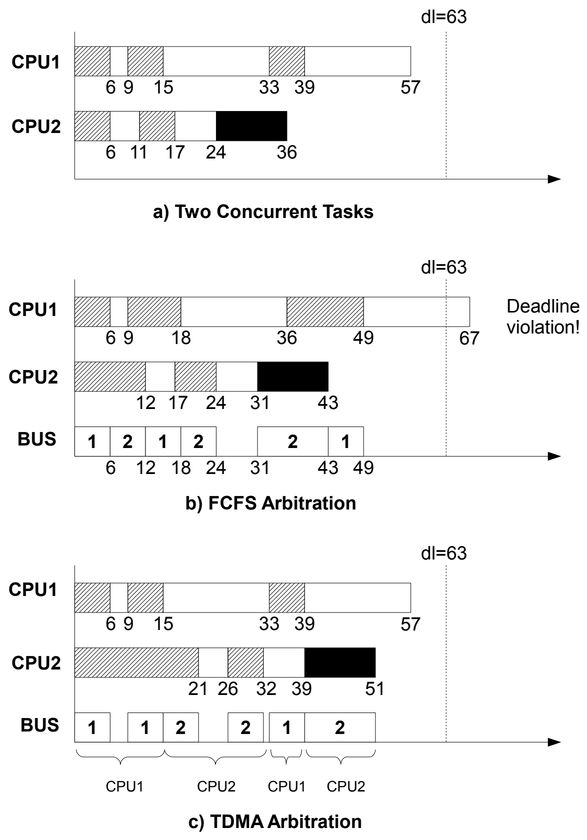


Figure 4. Motivational Example

in Figure 3.

To limit the required amount of memory on the bus controller, a TDMA round can be subject to various complexity constraints. A common restriction is to let every processor own, at most, a specified number of slots per round. Also, one can let the sizes be the same for all slots of a certain round, or let the slot order be fixed. For more details, we refer to our previous work [5].

III. MOTIVATIONAL EXAMPLE

Consider a multiprocessor system with two processors and a shared communication infrastructure according to Section

II. Each task has been analyzed with a traditional WCET tool, assuming a monoprocessor system, and the resulting Gantt chart of the respective worst-case program path is illustrated in Figure 4a. The dashed intervals represent cache misses, each of them taking six time units to serve, and the white solid areas represent segments of code not using the bus. Task 2 is also transferring data to the shared memory, and this is represented by the black solid area.

However, since the tasks are now running on a multiprocessor system with a shared communication infrastructure, they do not have exclusive access to their respective private memories. Hence, some kind of arbitration policy must be applied to distribute the bus bandwidth among the tasks. The result is that when two tasks request the bus simultaneously, one of them has to wait until the other has finished transferring. This means that transfer times are no longer constant, but now instead depend on the load on the other processors. Figure 4b shows the resulting Gantt chart when the commonly used FCFS arbitration policy is applied.

The fundamental problem when performing worst-case execution time analysis on multiprocessor systems is that the load on the other processors is in general not known. For a task, the number of cache misses and their location in time depend on the program control flow path. This means that it is very hard to foresee where there will be bus access collisions, since this will differ from execution to execution. To complicate things further, the worst-case control flow path of the task will change depending on the bus load originating from the other concurrent tasks. In order to solve this and introduce predictability, we use a TDMA bus schedule which, a priori, determines exactly when a processor is granted the bus, regardless of what is executed on the other processors. Given a TDMA bus schedule, the WCET analysis tool calculates a corresponding worst-case execution time. Therefore, it is important that a clever bus schedule, optimized to reduce the worst case, is used. We refer to our previous work for algorithms and more details about how to create a good bus schedule [5]. Figure 4c shows the same task configuration as previously, but now the memory accesses are arbitrated according to TDMA. The result is a predictable system, with optimized worst-case task execution times.

IV. TDMA-Based WCET Analysis

Performing worst-case execution time analysis with respect to a TDMA bus schedule requires not only knowledge about the number of cache misses for a certain program path, but also their location with respect to time. Hence, traditional ILP-based methods for worst-case execution time analysis cannot be applied. Instead, each memory access needs to be considered with respect to the bus schedule, granting access to the bus only during the slots belonging to the requesting processor. However, to collect the necessary information used by our worst-case execution time analysis framework, the same techniques used in traditional methods can be utilized.

Calculating the worst-case execution time has to be done with respect to the particular hardware architecture on which the task being analyzed is going to be executed. Factors such as the instruction set, pipelining complexity, caches and so on must be taken into the account by the analysis. For an application running on a compositional architecture, the analysis can be divided into subproblems processed in a local fashion, for instance on basic block level. We can be sure that the local worst-case always contributes to the worst-case globally, allowing for fast analysis techniques without the need to analyze every single program path individually. This is, unfortunately, not the case when using noncompositional architectures. The presence of timing anomalies will force the analysis to consider all possible program paths explicitly, naturally causing the analysis time to explode as the size of the tasks increase. Consequently, this kind of architectures are not suitable for hard real-time systems, and therefore, most WCET analysis tools emphasize on compositional platforms.

For a predictable multiprocessor system with a shared communication structure, as described in Section II, it is necessary to search through all feasible program paths and match each possible bus transfer to slots in the actual bus schedule, keeping track of exactly when a bus transfer is granted the bus in the worst case. This means that the execution time of a basic block will vary depending on when, in time, it is executed. Fortunately, for an application running on a compositional architecture, efficient search-tree pruning techniques dramatically reduce the search space, allowing for local analysis, just as for traditional WCET techniques.

V. Compositional WCET Analysis Flow

A typical program flow for a WCET tool operating on compositional architectures is shown in the left path of Figure 5 [13]. First, a control flow graph (CFG) is generated. A value analysis is then performed to find program characteristics such as data address ranges and loop bounds. To take into account performance-enhancing features of modern hardware, cache and pipeline analyses are carried out next. A path analysis identifies the feasible paths and an ILP formulation for calculating the worst-case program path is then produced. The information traditionally provided in this ILP formulation is, however, not sufficient for calculating the WCET on a multiprocessor system since not only the number of cache misses are needed for each basic block, but also their positions with respect to time. If necessary, an underlying WCET tool has to be modified to provide this information. A more in-depth description can be found in our previous work [14].

Our TDMA-based approach for compositional WCET analysis is illustrated in the right path of Figure 5. After the path analysis, the information from the previous steps is

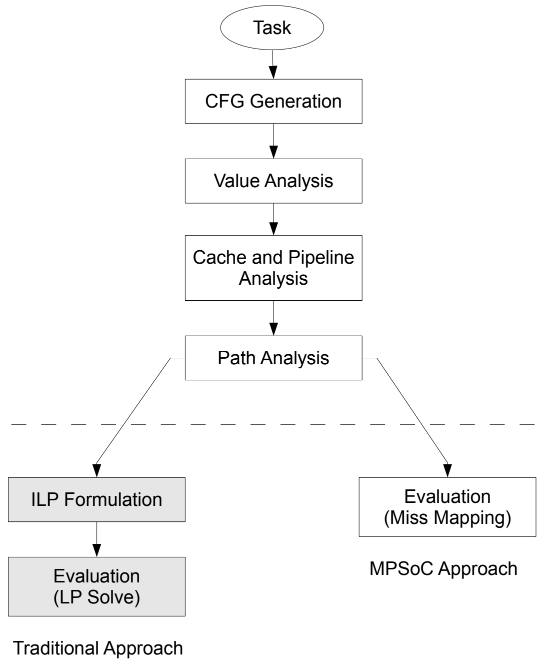


Figure 5. WCET tool program flow

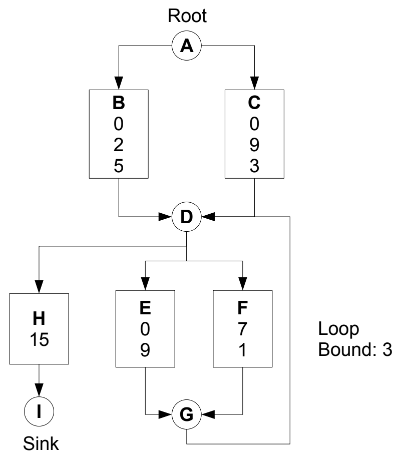


Figure 6. Example CFG

used to calculate the worst-case program path by mapping the cache misses to the corresponding bus slots in the TDMA schedule. We will now show the idea behind this with a simple example.

A. Monoprocessor WCET Example

Consider a task τ executing on a system with two processors (processor 1 and processor 2). The task is being mapped on processor 1, and has start time 0. First, an annotated control flow graph, as illustrated in Figure 6, is constructed. The rectangular elements $\mathbf{B}$, $\mathbf{C}$, $\mathbf{H}$, $\mathbf{E}$, $\mathbf{F}$ in the graph represent basic blocks, and the circles $\mathbf{A}$, $\mathbf{G}$, $\mathbf{I}$ represent control nodes gluing them together. The loop starting at control node $\mathbf{G}$ will run at most three times, so the loop bound is consequently set to 3. The annotated numbers in

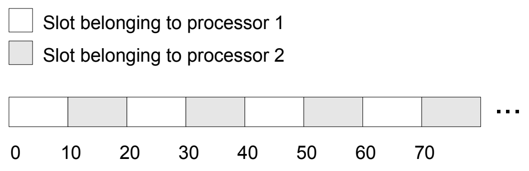


Figure 7. Example TDMA Bus Schedule

the basic blocks represent consecutive cycles of execution, in the worst case, not accessing the bus. For instance, basic block $\mathbf{B}$ will, when executed, immediately - after 0 clock cycles - issue a cache miss. After this, 10 cycles will be spent without bus accesses before the next (and last) cache miss occurs. Finally, 10 bus access-free cycles will be executed before the basic block ends. Hence, the execution time of basic block $\mathbf{B}$ will be $(0 + k_1 + 10 + k_2 + 10)$ where k_1 and k_2 represent the transfer times of the first and second cache miss respectively.

For a typical monoprocessor system, all cache misses take the same constant amount of time to process, and the execution time of basic block $\mathbf{B}$ would be known immediately. However, for multiprocessor architectures such as the one described in Section II, we must calculate the individual transfer times with respect to a given TDMA schedule.

B. Multiprocessor WCET Example

Instead of a monoprocessor system, assume a multiprocessor system, as described in Section II, using the bus schedule in Figure 7. Processor 1, on which the task is running, gets a bus slot of size 10 processor cycles periodically assigned to it every 20th cycle. In this particular example, a cache miss takes 10 cycles for the bus to transfer, resulting in the bus being granted to processor 1 *only* at times t satisfying $t \equiv 0 \pmod{20}$, where $\equiv$ is the congruence operator.

To calculate the worst-case program path, we must evaluate all feasible program paths in the control flow graph. In the very simple example in Figure 6, there are 30 program paths[1] to explore, growing exponentially with the number of branches and loop bounds. Fortunately, due to the nature of the compositional architecture and the TDMA bus, not all of them have to be investigated explicitly. In fact, in a task graph with all loops unrolled, each basic block would need be be investigated exactly once, as will be explained in the following.

Let us denote the worst-case start time of a basic block $\mathbf{Z}$ by $s(\mathbf{Z})$, and the end time in the worst case by $e(\mathbf{Z})$. The execution time of a basic block $\mathbf{Z}$, in the worst case, is then defined as $w(\mathbf{Z}) = e(\mathbf{Z}) - s(\mathbf{Z})$. Without considering bus conflicts, as in traditional methods, the worst-case execution time of the basic blocks would be $w_{\text{trad}}(\mathbf{B}) = 27, w_{\text{trad}}(\mathbf{C}) = 32, w_{\text{trad}}(\mathbf{E}) = 19, w_{\text{trad}}(\mathbf{F}) = 18$ and $w_{\text{trad}}(\mathbf{H}) = 15$. The corresponding worst-case program

[1] $2 + 2^2 + 2^3 + 2^4 = 30$

path becomes $\mathbf{C}, \mathbf{E}, \mathbf{E}, \mathbf{E}, \mathbf{H}$ resulting in a worst-case execution time of $27+19 \cdot 3+15 = 104$ clock cycles. However, this assumes that all cache misses take the same amount of time to transfer, and this is false in a multiprocessor system with a shared communication structure. In our TDMA-based approach, the execution time of a basic block depends on its start time in relation to the bus schedule. We start from the root node and successively calculate the execution time of each basic block with respect to the worst-case start time. At the same time, the worst-case path is calculated.

With respect to the TDMA schedule in figure 7, the worst-case start times of the basic blocks connected directly to the root node is 0, since they will never execute at any other time instant. The execution time of block $\mathbf{B}$, in the worst case, is $w(\mathbf{B}) = 0 + 10 + 2 + 18 + 5 = 35$ whereas the corresponding execution time of block $\mathbf{C}$ is $w(\mathbf{C}) = 0+10+9+11+3 = 33$. Note that $w(\mathbf{B}) > w(\mathbf{C})$, even though the relation is the opposite in the traditional case above where $w_{\text{trad}}(\mathbf{B}) < w_{\text{trad}}(\mathbf{C})$. In order to decide which one of these two basic blocks is on the critical path, two very important observations must be made based on the predictable nature of the TDMA bus.

1) *The absolute end time of a basic block can never increase by letting it start earlier*. That is, a basic block $\mathbf{Z}$ with $s(\mathbf{Z}) = x$ and $e(\mathbf{Z}) = y$, any start time $x' < x$ will result in an end time $y' \leq y$. The execution time of the particular basic block can increase, but the increment can never exceed the difference $x - x'$ in start time. This means that for a basic block $\mathbf{Z}$, the basic block will never end later than $e(\mathbf{Z})$ as long as it start before (or at) $s(\mathbf{Z})$. This guarantees that the worst-case calculations will never be violated, no matter what program path is taken. Note that $w(\mathbf{Z})$ is the execution time in the worst case, with respect to $e(\mathbf{Z})$, and that the time spent by executing $\mathbf{Z}$ can be greater than $w(\mathbf{Z})$ for an earlier start time than $s(\mathbf{Z})$.
2) Consider a basic block $\mathbf{Z}$ with worst-case start time $s(\mathbf{Z}) = x$ and worst-case end time $e(\mathbf{Z}) = y$. If we, instead, assume a worst-case start time of $s(\mathbf{Z}) = x''$ where $x'' > x$, the corresponding resulting absolute end time $e(\mathbf{Z}) = y''$ will always satisfy the relation $y'' \geq y$. This means that the greatest assumed worst-case start time $s(\mathbf{Z})$ will also result in the greatest absolute end time $e(\mathbf{Z})$.

Based on the second observation, we can be sure that the maximum absolute end time for the basic block ($\mathbf{E}$, $\mathbf{F}$ or $\mathbf{H}$) succeeding $\mathbf{B}$ and $\mathbf{C}$ will be found when the worst-case start time is set to 35 rather than 33. Therefore, we conclude that $\mathbf{B}$ is on the worst-case program path and, since they are not part of a loop, $\mathbf{B}$ and $\mathbf{C}$ do not have to be investigated again.

Next follow three choices. We can enter the loop by executing either $\mathbf{E}$ or $\mathbf{F}$, or we can go directly to $\mathbf{H}$ and end the task immediately. Due to observation 2 above, we can conclude that the worst-case absolute end time of $\mathbf{H}$, and thus the entire task, will be achieved when the loop iterates the maximum possible number of times, which is 3 iterations, since that will maximize $s(\mathbf{H})$. Therefore, the next step is to calculate the worst-case execution time for basic blocks $\mathbf{E}$ and $\mathbf{F}$ respectively for each of the three iterations, before finally calculating the worst-case execution time of $\mathbf{H}$. In the first iteration, the worst-case start time is $s(\mathbf{E_1}) = s(\mathbf{F_1}) = 35$ and the execution times become $w(\mathbf{E_1}) = 0 + 15 + 9 = 24$ and $w(\mathbf{F_1}) = 7 + 28 + 1 = 36$ for $\mathbf{E}$ and $\mathbf{F}$ respectively. We conclude that the worst-case program path so far is $\mathbf{B}, \mathbf{F}$ and the new start time is set to $s(\mathbf{E_2}) = s(\mathbf{F_2}) = 35 + 36 = 71$. In the second loop iteration, we get $w(\mathbf{E_2}) = 0 + 19 + 9 = 28$ and $w(\mathbf{F_2}) = 7 + 12 + 1 = 20$. Hence, in this iteration, $\mathbf{E}$ contributes to the worst-case program path and the new worst-case start time becomes $s(\mathbf{E_3}) = s(\mathbf{F_3}) = 99$. In the final iteration, the execution times are $w(\mathbf{E_3}) = 0 + 11 + 9 = 20$ and $w(\mathbf{F_3}) = 7 + 24 + 1 = 32$ respectively, resulting in the new worst-case start time $s(\mathbf{H}) = 131$. We now know that the worst-case program path is $\mathbf{B}, \mathbf{F}, \mathbf{E}, \mathbf{F}, \mathbf{H}$, and since $\mathbf{H}$ contains no cache misses, and therefore always takes 15 cycles to execute, the WCET of the entire task is $e(\mathbf{H}) = 146$.

As shown in the this example, in a loop-free control flow graph, each basic block has to be visited once. For control flow graphs containing loops, the number of investigations will be the same as for the case where all loops are unrolled according to their respective loop bounds. The result, when the graph is traversed, is a time-complexity not higher than for traditional monoprocessor worst-case execution time analysis techniques.

VI. Noncompositional Analysis

In the presence of timing anomalies, it is no longer possible to do local assumptions about the global worst case execution time. Therefore, for such architectures, every program path has to be analyzed explicitly. This is the case, not only for multiprocessor systems, but for any worst-case execution time framework operating on a noncompositional platform. Also, all steps in Figure 5, from the cache and pipeline analyses and forward, must be integrated since it, for noncompositional architectures, is impossible to assume safe initial cache and pipeline states for a basic block, regardless of the allowed pessimism. Since also traditional WCET analysis operating on noncompositional hardware has to perform a global search through all program paths, the modifications in order to make it aware of the TDMA bus is, in theory, straight-forward. To adapt a traditional noncompositional WCET analysis technique to the class of multiprocessor systems described in Section II, for each considered cache miss, the bus schedule has to be searched in order to find the start and end times of the corresponding

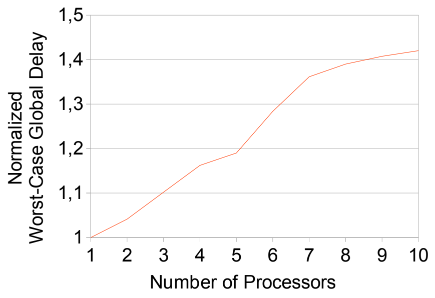


Figure 8. Predictable WCET Analysis vs Traditional WCET Analysis

bus transfer. This operation is of linear complexity and will therefore not increase the total, already exponential, complexity of the traditional worst-case execution time analysis.

VII. Experimental Results

To demonstrate the efficiency of our approach, we have performed WCET analysis on randomly generated applications running on a multiprocessor system configured as described in Section II, with 10 ARM7 cores running at 200 MHz. Each application was constituted by 50 to 200 tasks, generated according to randomized task graphs, that were mapped on 2 to 10 processors. The individual tasks were composed by commonly used routines for computations such as sorting, searching, matrix operations and DSP processing.

For all of the generated applications, we calculated the worst-case global delay, that is, the time it takes to execute a particular application in the worst case. This was done using both predictable, modified WCET analysis and the corresponding, non-modified, traditional analysis method that assumes that no conflicts can occur on the bus. Naturally, if possible bus conflicts are neglected when calculating the worst-case global delay, the result will be optimistic and therefore incorrect. Figure 8 shows how many times larger the worst-case global delay becomes when using predictable WCET analysis, relative to its traditional, incorrect counterpart (represented by the baseline). For each processor configuration, 50 applications were analyzed, and an average was calculated.

Optimization techniques described in our previous work [5] were used to compute the worst-case global delay achieved with predictable WCET analysis, and consequently the worst-case execution time of each task was calculated several times to evaluate TDMA bus schedule candidates. Still, a big application consisting of 100 tasks running on 10 processors took only 5 minutes to process on a 2.8 GHz dual core Pentium 4 computer, from CFG generation to getting the actual result. This clearly shows that the predictable, TDMA-based WCET analysis is very fast.

VIII. Conclusions

Traditional worst-case execution time analysis techniques cannot be applied directly to modern multiprocessor platforms. In this paper, we have demonstrated how to modify existing WCET analysis tools to make them function correctly for multiprocessor systems-on-chip, without significantly increasing the time complexity. For compositional architectures, our proposed modifications make it possible to analyze each basic block locally, allowing for quick analysis times, while achieving predictability. For noncompositional architectures, the necessary modifications do not increase the already high time complexity of traditional, monoprocessor WCET analysis tools operating on such platforms.

References

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

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

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

[4] L. Thiele and R. Wilhelm, “Design for Timing Predictability,” *Real-Time Systems*, vol. 28, no. 2/3, pp. 157–177, 2004.

[5] J. Rosén, A. Andrei, P. Eles, and Z. Peng, “Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip,” in *RTSS*, 2007, pp. 49–60.

[6] A. Andrei, P. Eles, Z. Peng, and J. Rosén, “Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip,” in *VLSI Conference*, 2008.

[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*, 2006, pp. 125–131.

[8] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, “Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems,” *IEEE Transactions on CAD of Integrated Circuits and Systems*, 2009.

[9] T. Lundqvist and P. Stenström, “Timing Anomalies in Dynamically Scheduled Microprocessors,” in *RTSS*, 1999, pp. 12–21.

[10] J. Reineke, B.Wachter, S. Thesing, R.Wilhelm, I. Polian, J. Eisinger, and B. Becker, “A Definition and Classification of Timing Anomalies,” in *International Workshop on WCET Analysis*, 2006, pp. 23–28.

[11] E. Salminen, V. Lahtinen, K. Kuusilinna, and T. Hamalainen, “Overview of bus-based system-on-chip interconnections,” in *ISCAS*, 2002, pp. 372–375.

[12] S. Pasricha, N. Dutt, and M. Ben-Romdhane, “Fast exploration of bus-based on-chip communication architectures,” in *CODES+ISSS*, 2004, pp. 242–247.

[13] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, , and P. Stenström, “The Worst-case Execution Time Problem – Overview of Methods and Survey of Tools,” *ACM Transactions on Embedded Computing Systems (TECS)*, 2008.

[14] Carl-Fredrik Neikter, “Cache prediction and execution time analysis on real-time mpsoc,” 2008, Master Thesis, LIU-IDA/LITH-EX-A–08/046–SE.