# Technical report IRIT/RR–2011-XX–FR
# –
# Predictable Bus Arbitration Schemes for Heterogeneous Time-Critical Workloads Running on Multicore Processors

Roman Bourgade, Christine Rochange, Pascal Sainrat
Institut de Recherche en Informatique de Toulouse
University of Toulouse, France
{bourgade,rochange,sainrat}@irit.fr

## Abstract

*Multi-core architectures are now considered as possible candidates to implement future time-critical embedded systems. The challenge is to make the worst-case execution time (WCET) of each task predictable. This requires upper-bounded memory latencies which are achievable through a WCET-aware bus arbiter. Round-robin protocols enforce the same worst-case latency for each core. In this paper we focus on heterogeneous workloads in which tasks exhibit distinct requirements in terms of bandwidth to the main memory. We investigate several arbitration schemes able to guarantee distinct worst-case latencies to the cores. This allows allocating the most requiring tasks to the best served cores. The proposed schemes perform a two-level arbitration: the cores are organized into groups and all the cores in the same group benefit from the same bandwidth. Different algorithms to share the bus slots among the groups are considered. Experimental results (WCET estimates) show an improved global WCET compared to usual round-robin schemes. This will enhance the schedulability of heterogeneous task sets.*

## 1. Introduction

Embedded systems have to fulfil strong constraints in terms of power consumption or thermal dissipation. They also must achieve high cost-efficiency in a very competitive market. For these reasons it is becoming more and more difficult to improve their performance by the only means of increasing their clock rate or making their internal architecture more complex.

A solution is the use of multicore processors (CMP or chip multiprocessors), that improve the utilization of resources by running several tasks in parallel. Multicore processors are widely used in servers and desktop computers. With the growing demand of embedded applications including time-critical software, it is expected that multicores will also be increasingly used in real-time embedded systems to achieve higher performance/throughput and cost-effectiveness.

On typical medium-size CMPs, the cores share a bus to the highest levels of the memory hierarchy. This contributes to increase memory latencies, due to conflicts in the interconnection (bus). When executing hard real-time tasks, this is a major problem because memory latencies have to be bound to allow a safe estimation of the worst-case execution times (WCETs) of tasks. In order to keep things tractable, we believe that the design of a predictable bus arbiter should be done with the goal of keeping the worst-case bus latencies for one of the tasks independent of the other tasks. This makes possible to determine the WCET of each task without any knowledge of the possible co-scheduled tasks.

A conservative technique for hard real-time systems is to solve inter-core conflicts to shared resources by using fair policy such as the round-robin protocol. It guarantees that the memory access latencies for a particular

core remain independent from the memory requests by the other cores; they can be bounded and safely included in WCET estimates [11]. This approach perfectly fits systems running homogeneous workloads in which all the tasks have similar memory demands and should be served fairly. However, in some cases fairness may prevent some tasks to meet their deadlines while favoring some highly-demanding tasks to fasten their execution may help in achieving the schedulability of the whole system. Also, when considering parallel applications with inter-task dependencies, it may be desirable to accelerate the tasks on the critical path. This motivates our work that aims at providing several levels of bandwidth while keeping the worst-case latencies computable.

In this paper we introduce two-level bus arbiters that provide an unbalanced service to the cores. This allows allocating the most demanding tasks on the cores that often get bus slots and lower demanding tasks to cores that undergo a longer latency. The objective is to improve the performance of a task set instead of the performance of individual tasks. We consider several arbitration algorithm and we show that a two-level arbiter can reduce the maximum WCET in the task set by 27% to 48% according to the traffic for data, and the sum of the WCETs over a task set by 11% to 28%.

The paper is organized as follows. Section 2 gives an overview of related work and outlines the problems that motivate our approach. We describe two-level bus arbiters in Section 3. Section 4 provides experimental results and concluding remarks are given in Section 5.

## 2. Related work

To be eligible in the context of a system supporting hard real-time threads, a bus arbiter must insure that the worst-case latency for a given core to get access the memory hierarchy can be computed. Such real-time-aware protocols exist. As an example, the round-robin policy grants the bus to each core in turn. As a result, the maximum delay a core can undergo when requesting the bus is a function of the number of cores that share the bus, and of the latency of a bus access. This delay is predictable, does not depend on the tasks running on the others cores and is the same for each core [11].

Time-Division Multiple Access (TDMA) policies are also being increasingly used in embedded hard real-time systems. As in round-robin, all the cores are served in turn, but a core can be granted several successive bus slots instead of a single one. Slot allocation is determined off-line, like in [2] and [14]. However, as mentioned in [13], it generally cannot be determined during static analysis whether a bus request will occur during one the granted slots. As a result, static WCET analysis requires considering the length of all the slots allocated to other cores as the upper bound delay which may be overwhelmingly pessimistic.

Budget schedulers ([13, 1]) have also been designed to satisfy variable bus bandwidth requirements. However, instead of determining a static bus scheduling like TDMA schemes, they allocate a given amount of bandwidth to each core. According to its priority level, and as far as it has not consumed its allocated bus slots budget, a core can be granted the bus as soon as it issues a request. The prediction of worst-case latencies within WCET computation requires considering the distance between two requests to the bus by the thread under analysis. This might be possible for some particular applications, e.g. when considering accesses to data for a data stream application, but is far more complex for irregular and dynamic accesses to memory like those required to fill the instruction cache on cache misses (the instruction cache analysis generally cannot determine the cache behavior for some of the instruction fetches).

Recently we proposed the MBBA scheme that aimed at offering distinct bus bandwidths to the cores in a multicore architecture [5]. In this paper, we improve this scheme in a way that makes it possible to specify it formally and to implement it in hardware, and we extend the concept of time-predictable two-level bus arbiters so that different arbitration algorithms can be used. We also report a stronger experimental analysis that investigates all the possible arbiter configurations and task allocations for a small set of benchmarks.

## 3. Two-level Bus Arbitration for Heterogeneous Workloads

### 3.1. Task Sensitivity to the Bus Latency

A round-robin bus arbiter serves all the tasks in a fair way. For hard real-time applications, an identical worst-case latency ($N \times L$ where $N$ is the number of cores and $L$ the bus latency) should be assumed when

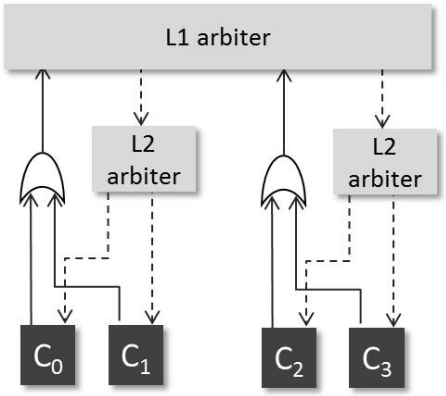


**Figure 1. A two-level bus arbiter**

analysing the worst-case execution time of any task [11]. In this paper, we consider heterogeneous workloads that include tasks with various demands to the memory system. A task issues more bus requests than another task either because it experiences a larger amount of cache misses or simply because it has a longer execution time.

We define the *normalized WCET sensitivity $\sigma_t$ to the bus latency changing from $\lambda_1$ to $\lambda_2$* of a task $\tau$ in a task set $T$, as:

$$\forall \tau \in T, \qquad \sigma_t = \frac{WCET_\tau^{\lambda_2} - WCET_\tau^{\lambda_1}}{\sum_{t \in T} WCET_t^{\lambda_1}} \tag{1}$$

where $WCET_t^\lambda$ if the WCET of task $t$ considering latency $\lambda$. The normalization of the sensitivity to the sum of all the tasks WCETs makes this measure useful to select the tasks that should benefit of a larger bandwidth to the bus so that the performance of the whole task set is improved. The arbitration schemes described in the following aim at allowing an allocation of the bus slots to the cores that improves the global performance of a task set instead of enforcing fairness.

### 3.2. Two-level bus arbitration

We introduce *two-level* bus arbiters based on *groups* of cores: in the first level, the arbiter allocates a slot to one group; in the second level, one core in the group is selected and granted the bus. This is illustrated in Figure 1, considering two groups of two cores each: solid lines show *request* signals and dashed lines show *grant* signals. The arbitration policies at each of the two levels define the bus latency seen by each core.

In this paper, we focus on time-critical systems: only time-predictable arbitration strategies are considered. The round-robin (RR) algorithm [11] fulfils our needs. In addition, we consider the GL (*Geometric Latencies*) algorithm described in Section 3.4.1: it allocates different partitions of the bus bandwidth to the cores with power-of-two worst-case bus latencies. Considering these two algorithms, four arbiters can be built. However, GL does not make much sense for a small number of cores. For this reason we only consider the RR algorithm in the second level (to arbitrate among the cores in a group) and we consider both the RR and GL schemes at the first level (to select a group). This makes two arbiters: GRR (*Group Round Robin*) and GGL (*Geometric Group Latencies*). Note that GGL is pretty close to the MBBA scheme we introduced recently [5] but GGL results from a few enhancements that make a more formally specification possible.

Each scheme must be configured: each core must be assigned to one group (in this paper we consider up to 3 groups). For a given first-level algorithm, the number of cores in a group determines the worst-case latency seen by each core in the group. Let $L_{G_i}$ be the worst-case bus latency for group $G_i$, $N_i$ be the number of cores in $G_i$ and $N$ be the number of groups. If the cores in a group are arbitrated with a RR protocol, each core sees a worst-case bus latency equal to:

$$\begin{aligned} \forall i \in [0..N-1], \forall j \in [0..N_i - 1], \\ L_{C_{j \in G_i}} = N_i \times L_{G_i} \end{aligned} \tag{2}$$

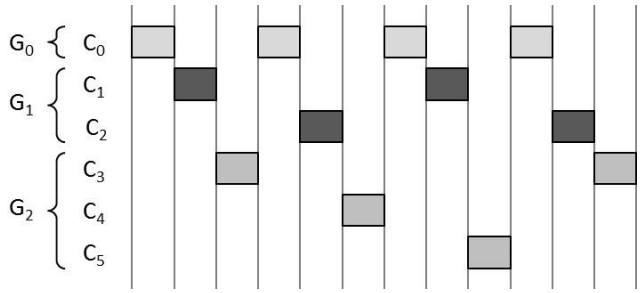


**Figure 2. Example GRR schedule**

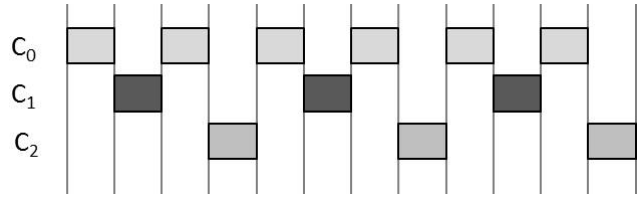


**Figure 3. Example GL schedule**

### 3.3. Group Round Robin (GRR) protocol

Figure 2 illustrates the GRR protocol considering 6 cores partitioned into 3 groups and assuming each core permanently issues requests to the bus. Each group gets the bus every third cycle and its cores share the granted slots in a round-robin fashion.

Let $L$ be the bus latency for any request. For the GRR arbiter with $N$ groups and $N_i$ cores in group $G_i$, the worst-case bus latency seen by a group is given by:

$$\forall i \in [0..N-1], L_{G_i} = N \times L \tag{3}$$

From Equation 2, we get the worst-case latency seen by a core:

$$\begin{aligned} \forall i \in [0..N-1], \forall j \in [0..N_i-1], \\ L_{C_{j \in G_i}} = N_i \times (N \times L) \end{aligned} \tag{4}$$

### 3.4. Geometric Group Latencies

The GGL protocol is based on the GL algorithm used to select groups in the first level. In the following, we first explain the GL algorithm before analysing the latencies for GGL.

#### 3.4.1 Geometric Latencies

The main principle of the GL single-level arbiter is to enforce bus latencies that grow as a geometric series: core $C_0$ sees a latency of 2 ($\times L$), core $C_1$ a latency of 4, core $C_2$ a latency of 8, etc. (the two last cores have the same latency). The behavior of the GL arbiter is shown in Figure 3.

Formally, the GL arbiter can be specified as follows. The control signals involved in an hardware implementation of the protocol include $r_i$ that indicate (when set) that core $C_i$ has at least a pending request and $g_i$ is set for the core that is granted the bus and cleared for all the other cores. In addition, state variable $p_i$ is set when core $C_i$ should be given priority on the next slot left by the lower-range cores ($C_j$ with $j < i$). In the following, $r_i$, $g_i$ and $p_i$ are considered as boolean variables.

The following equations formally specify the GL protocol:

$$g_i^t = r_i^t . p_i^t . \prod_{j=0}^{i-1} \overline{p_j^t} \tag{5}$$

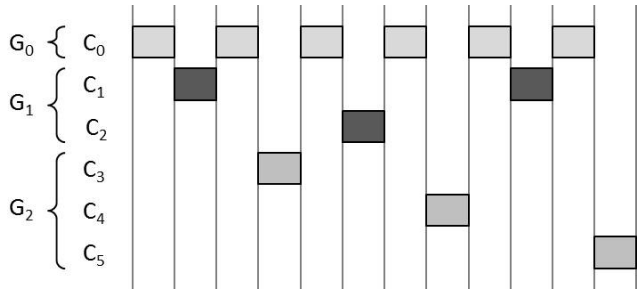


**Figure 4. Example GGL schedule**

$$\begin{cases} p_i^{t+1} = p_i^t \bigoplus \prod_{j=0}^{i-1} \overline{p_j^t} & \forall i \neq N-1 \\ \\ p_{N-1}^{t+1} = \overline{p_{N-2}^{t+1}} \end{cases} \tag{6}$$

Equation 5 tells that the bus is allocated to core $C_i$ at cycle $t$ if this core has requested the bus, it has priority over higher-range cores and no lower-range core has priority over it. Equation 6 indicates that core $C_i$ will has priority over higher-range cores at cycle $t+1$ if (a) it has this priority at cycle $t$ but cannot get the bus because a lower-range core has priority over it; or (2) core $C_i$ has not priority over higher-range cores but no lower-range core has priority over it. Last core, $C_{N-1}$ is served alternately with core $C_{N-2}$.

In addition to providing a formal specification of the protocol, these equations give insight into how it could be implemented in hardware as a Mealy machine.

From these equations, we get the expression of the worst-case latency for a core (proof is given in Appendix):

$$\begin{cases} L_{C_i} = 2^{i+1} \times L & \forall i < n-1 \\ \\ L_{C_{N-1}} = 2^{N-1} \times L \end{cases} \tag{7}$$

### 3.4.2 Geometric Group Latencies

Figure 4 shows an example behavior of the GGL protocol with 6 cores partitioned into 3 groups that permanently issue requests to the bus. The core in $G_0$ gets every second bus slot. Group $G_2$ is granted every fourth slot and these slots are fairly shared by the three cores in this group.

Considering $N$ groups, the worst-case latency for a group is given by Equation 7 and the worst-case latency for a core $C_j$ in group $G_i$ (with $N_i$ cores: $j \in [0..N_i - 1]$) is:

$$\begin{cases} L_{C_j \in G_i} = (N_i.2^{i+1}) \times L \quad for\ i \in [0..N-2] \\ \\ L_{C_j \in G_{N-1}} = (N_{N-1}.2^{N-1}) \times L \end{cases} \tag{8}$$

## 4. Experimental results

### 4.1. Methodology

The protocol presented in this paper has been modeled within OTAWA, a framework dedicated to static WCET analysis [4] that includes:

- a binary code loader that supports various instruction sets (PowerPC, ARM, TriCore, etc.)
- an instruction cache analyzer based on abstract interpretation techniques [7] [3]
- a timing analyzer that evaluates the worst-case execution time of basic blocks taking into account the target architecture and the results of the instruction cache analysis [12]

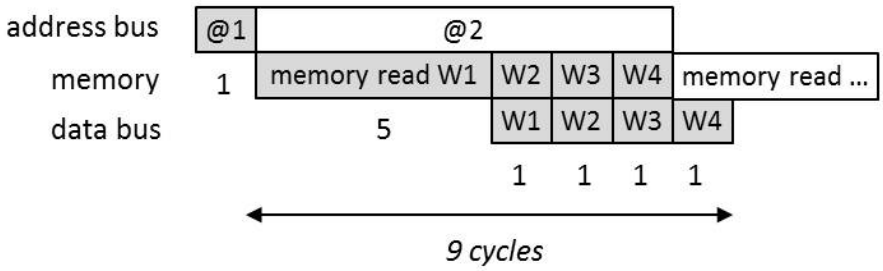


**Figure 5. Bus latency**

| | |
|---|---|
| **Fetch stage width** | 4 |
| **Fetch queue size** | 8 |
| **Decode stage width** | 2 |
| **Commit stage width** | 2 |
| **Functional units** *(latency)* | |
| **integer ALU** *(1 cycle)* | 1 |
| **fp ALU** *(3 cycles)* | 1 |
| **multiplier** *(6 cycles)* | 1 |
| **divider** *(15 cycles)* | 1 |
| **memory** *(2 cycles)* | 1 |

**Table 1. Core configuration.**

- a flow-fact loader that reads flow fact annotations provided by the oRange tool [6]
- a WCET computer that builds an integer linear program according to the IPET method [10]. This program is solved using the lp_solve tool[1]

The experiments reported in this paper were carried out considering an 8-core CMP with in-order superscalar cores implementing the PowerPC ISA configured as shown in Table 1. Each core includes private instruction and data caches. Since we have considered benchmarks with limited code sizes, we considered small instruction caches so that the benchmark codes do not fit in: 2 KB, 2-way associative with 16-byte lines. Our tool does not model data caches, so we have run the experiments considering first always-hit, then always-miss data caches with 16-byte cache lines. The bus is 32-bit wide. Its latency is one cycle and the memory worst-case latency for a read or write operation is 5 cycles. Figure 5 shows how the latency of fetching a cache line from the memory: considering a sequence of fetches, the latency of one such operation is 9 cycles (the address phase overlaps the previous request). It is 10 cycles for the first request of the sequence. Then, considering a simple round-robin arbiter for 8 cores, the worst-case latency seen by one core is 73 cycles.

The benchmarks considered in this study are listed in Table 2. The three first belong to the Mälardalen collection [8] (some functions have been resized to get sufficiently long execution times), the other ones are functions from the Susan benchmark in the MiBench suite [9]. Since we want to perform this analysis independently from the task scheduling algorithm, we consider a set of 8 tasks only for our 8-core architecture so that each task runs on one core.

Table 3 gives the WCET sensitivities of the tasks, computed from Equation 1 with the reference latency ($\lambda_1$) equal to 73 cycles which is the worst-case latency observed when the bus is scheduled by a simple round-robin arbiter. Several latency values ($\lambda_2$) are considered: they are taken from the list of all the possible values related to the possible configurations of a two-level arbiter. For example, for a GGL-$\{$*2-2-4*$\}$ configuration, each of the 2 (resp. 2 and 4) cores in group $G_0$ (resp. $G_1$ and $G_2$) see a 37-cycle (resp. 73-cycle and 145-cycle) latency. The numbers in the table should be read as follows: guaranteeing a latency of 19 cycles instead of

[1] `http://lpsolve.sourceforge.net/`

| Benchmark | Function |
|---|---|
| nsichneu | Simulates an extended Petri Net. Automatically generated code containing large amounts of if-statements (more than 250). |
| statemate | Automatically generated code. Generated by the STAtechart Real-time-Code generator STARC. |
| compress | Data compression program. Adopted from SPEC95 for WCET-calculation. Only compression is done on a buffer (small one) containing totally random data. |
| susan_corners_quick | Corner finding algorithm from SUSAN Low Level Image Processing program. |
| susan_edges_small | Edge finding algorithm from SUSAN program. |
| susan_principle | Filter application algorithm from SUSAN program. |
| edge_draw | Edge drawing algorithm from SUSAN program. |
| corner_draw | Corner drawing algorithm from SUSAN program. |

**Table 2. Benchmarks.**

| Data cache | *always hits* | | | | | *always misses* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Bus latencies** | **19** | **37** | **91** | **145** | **217** | **19** | **37** | **91** | **145** | **217** |
| nsichneu | -8.1% | -5.4% | +2.7% | +10.8% | +21.5% | -1.7% | -1.1% | +0.6% | +2.2% | +4.5% |
| statemate | -11.6% | -7.7% | +3.9% | +15.4% | +30.8% | -2.3% | -1.5% | +0.8% | +3.0% | +6.1% |
| susan_corners_quick | -26.7% | -17.8% | +8.9% | +35.6% | +71.1% | -13.6% | -9.1% | +4.5% | +18.2% | +36.4% |
| susan_edges_small | -16.4% | -11.0% | +5.5% | +21.9% | +43.8% | -18.5% | -12.4% | +6.2% | +24.7% | +49.4% |
| compress | -1.1% | -0.8% | +0.4% | +1.5% | +3.0% | -1.9% | -1.3% | +0.6% | +2.6% | +5.2% |
| susan_principle | -0.3% | -0.2% | +0.1% | +0.4% | +0.9% | -13.0% | -8.7% | +4.3% | +17.4% | +34.8% |
| edge_draw | -0.0% | -0.0% | +0.0% | +0.0% | +0.0% | -10.1% | -6.7% | +3.4% | +13.5% | +27.0% |
| corner_draw | -0.0% | -0.0% | +0.0% | +0.0% | +0.0% | -11.6% | -7.7% | +3.9% | +15.5% | +30.9% |

**Table 3. WCET sensitivities to the bus latency (reference latency is 73).**

73 cycles to task nsichneu improves the sum of the WCETs by 8.1% with an always-hit data cache. From this table, it appears that degrading the latency for some of the tasks (e.g. compress) will not impact much the global WCET of the task set while the set will benefit from reducing the latency of some other tasks (e.g. susan_corners_quick).

### 4.2. Results

Our objective in this section is to show how two-level arbitration schemes can help in improving the global worst-case performance of a task set. The two-level schemes are based on up to three groups, since four groups do not make sense for 8 cores.

Our reference is a simple round-robin algorithm that provides the same worst-case latency to each task in the set. The two-level arbiters need to be configured to decide which task is allocated to which group. Table 4 shows the characteristics of configurations considered in the following. Columns 1-3 specify the number of cores in each group. Note that redundant configurations have been omitted (for example, $\{6,1,1\}$ is the same as $\{1,1,6\}$ for GRR and the same as $\{2,6\}$ for GGL). Columns 4-6 (resp. 7-9) give the latencies seen by the cores in each of the three groups for the GRR (resp. GGL) arbiter. Finally column 10 indicates the number of possible allocations of the 8 tasks considering a given configuration.

Table 5 gives the results obtained when considering an *always-miss* data cache. Columns 1-3 show how the cores are partitioned into groups. For each configuration and every possible allocation of the tasks (see Table 4) we have determined the *maximum WCET* over the task set (if the tasks were threads of a parallel application, the maximum WCET would impact the global WCET of the application) and the *sum* of all the tasks WCETs (that gives insight into the utilization of the cores). Columns 4-5 (resp. 6-7) give the best numbers observed over the set of possible tasks allocations for the GRR (resp. GGL) algorithm. The first line is for a simple round-robin arbiter (a single group of 8 cores that all benefit from the same bandwidth): the results for this configuration are our reference numbers. The numbers in bold face show the lowest and highest value of each measure. It can be observed that the difference between results of different configurations can be huge. The best results are for

| Configuration | | | GRR latencies | | | GGL latencies | | | number of |
|---|---|---|---|---|---|---|---|---|---|
| $N_0$ | $N_1$ | $N_2$ | $L_{C_i \in G_0}$ | $L_{C_i \in G_1}$ | $L_{C_i \in G_2}$ | $L_{C_i \in G_0}$ | $L_{C_i \in G_1}$ | $L_{C_i \in G_2}$ | tasks allocations |
| 8 | - | - | 73 | - | - | - | - | - | 1 |
| 1 | 7 | - | - | - | - | 19 | 127 | - | 8 |
| 2 | 6 | - | - | - | - | 37 | 109 | - | 28 |
| 3 | 5 | - | - | - | - | 55 | 91 | - | 56 |
| 1 | 1 | 6 | 28 | 28 | 163 | 19 | 37 | 217 | 56 |
| 1 | 2 | 5 | 28 | 55 | 136 | 19 | 73 | 181 | 168 |
| 1 | 3 | 4 | 28 | 82 | 109 | 19 | 109 | 145 | 280 |
| 2 | 1 | 5 | 55 | 28 | 136 | 37 | 37 | 181 | 168 |
| 2 | 2 | 4 | 55 | 55 | 109 | 37 | 73 | 145 | 420 |
| 2 | 3 | 3 | 55 | 82 | 82 | 37 | 109 | 109 | 560 |
| 3 | 1 | 4 | 82 | 28 | 109 | 55 | 37 | 145 | 280 |
| 3 | 2 | 3 | 82 | 55 | 82 | 55 | 73 | 109 | 560 |
| 4 | 1 | 3 | 109 | 28 | 82 | 73 | 37 | 109 | 280 |
| 5 | 1 | 2 | 136 | 28 | 55 | 91 | 37 | 73 | 168 |
| | | | | | | | | *total* | 3,033 |

**Table 4. Arbiter configurations**

the GGL arbiter: for the best task allocation on the GGL-{4-1-3} configuration, the maximum WCET over the task set is improved by 26.7% compared to a simple RR scheme; and the greatest reduction of the sum of the tasks WCETs (-11.1%) is obtained with the best allocation on the GGL-{3-2-3} configuration. A closer look at the task allocation that exhibits the best maximum WCET shows that the task that has the highest sensitivity to the bus latency (`susan_edges_small` – see Table 3) is in group $G_0$ with a latency of 19 cycles and the three tasks having the lowest sensitivity (`nsichneu`, `statemate` and `compress`) are in group $G_2$ with a latency of 109 cycles.

Now, considering an always-miss data cache is pessimistic and leads to assuming a traffic on the bus that is far denser than it would be with a real cache. As said before, our WCET analysis tool still does not perform full data cache analysis. So, to get insight into how the two-level arbiters would behave in the case of a lower traffic on the bus, we also have considered a perfect (always-hit) data cache. The best results are again reached with the GGL arbiter: the maximum WCET over the task set is improved by up to 48.0% and the total WCET is lowered by up to 27.9%.

These results show that it is possible to improve the performance of a simple round-robin scheme with only slightly more complex schemes as those proposed in in this paper (GRR and GGL two-level arbiters) while still offering the possibility of determining worst-case latencies that do not depend on the co-scheduled tasks. This property is the key for keeping WCET analysis tractable. Now, in this work we have explored all the parameter space, i.e. all the possible configurations (number of groups and number of cores for each group) and all the possible allocations of tasks for each configuration. As future work, we will investigate joined allocation and scheduling algorithms that will exploit the possibility of our arbitration scheme.

## 5. Conclusion

Multicore architectures appear to be relevant candidates to implement embedded systems: they make it possible to improve the computing power while exhibiting essential qualities: high hardware integration, low thermal dissipation and low energy consumption. However, some critical embedded applications are subject to strict timing constraints. Thus, it is necessary to be able to analyze their worst-case execution time to check that deadlines can be met.

Much work has been done these last fifteen years to set up techniques to compute safe WCETs. However, their validity relies on the assumption that the application can run on the target architecture without being impacted by external events that would impair determinism. In a multicore processor, such events could be

| Configuration | | | GRR arbitration | | GGL arbitration | |
|---|---|---|---|---|---|---|
| $N_1$ | $N_2$ | $N_3$ | best *maximum WCET* | best *sum of WCETs* | best *maximum WCET* | best *sum of WCETs* |
| 8 | - | - | *46,021,703* | *181,588,379* | *46,021,703* | *181,588,379* |
| 1 | 7 | - | 58,496,999 | 146,514,523 | 58,496,999 | 246,514,523 |
| 2 | 6 | - | 48,134,578 | 191,861,771 | 48,134,578 | 191,861,771 |
| 3 | 5 | - | 36,030,829 | 170,941,235 | 36,030,829 | 170,941,235 |
| 1 | 1 | 6 | **71,810,338** | **255,936,614** | **95,486,098** | **328,264,133** |
| 1 | 2 | 5 | 53,581,234 | 195,898,091 | 71,131,639 | 248,212,769 |
| 1 | 3 | 4 | 37,865,309 | 184,518,725 | 50,244,323 | 233,040,281 |
| 2 | 1 | 5 | 53,581,234 | 195,898,091 | 71,131,639 | 227,136,803 |
| 2 | 2 | 4 | 37,561,765 | **166,591,427** | 49,802,341 | 181,418,639 |
| 2 | 3 | 3 | **36,296,698** | 174,423,776 | 48,134,578 | 191,861,771 |
| 3 | 1 | 4 | 37,865,309 | 184,518,725 | 49,802,341 | 174,759,233 |
| 3 | 2 | 3 | **36,296,698** | 174,423,776 | 34,808,477 | **161,371,013** |
| 4 | 1 | 3 | 37,865,309 | 184,518,725 | **33,738,971** | 166,302,383 |
| 5 | 1 | 2 | 53,581,234 | 195,898,091 | 36,030,829 | 175,872,605 |

**Table 5. Best WCET results with an *always-miss* data cache (# cycles)**

related to conflicts on the shared bus with tasks that run on another core. As a consequence, it is necessary to implement a bus arbitration scheme that makes it possible to upper bound the bus latencies, in such a way that the WCET of one task can be computed without any knowledge about the tasks that may be executed simultaneously on the multicore.

Several WCET-aware bus arbitration schemes have been proposed: for some of them, the predictability of the latencies during the WCET computation process is questionable; for other ones, like Round-Robin [11], the predictability is assessed but the algorithm does not fit well unbalanced workloads in which some threads require more bus bandwidth than the other ones.

In this paper, we have introduced two-level bus arbiters that feature a certain flexibility in the way bus slots are granted to the different cores, while keeping the predictability of the round-robin protocol. The cores are partitioned into groups and groups are guaranteed either a fair latency (GRR scheme) or a geometric (power-of-two) latency (GGL scheme). The cores in a group shared the group bus slots following a round-robin algorithm. The way the arbiter is configured (number of groups and number of cores in each group) determined the range of latencies seen by the cores. This flexibility allows offering a larger bandwidth to the tasks that have the largest requirements so that the worst-case performance of the task set is improved. Experimental results show that the GGL protocol achieves a maximum WCET and a sum of WCETs over our test-case task set respectively improved by 27% and 11% with a pessimistic (always miss) data cache and by 48% and 28% with a perfect (always hit) data cache.

Future work will focus on setting up strategies to determine optimal configurations for the GRR and GGL protocols as well as investigating the possibilities of improving the system-level scheduling of a set of tasks considering this kind of bus arbitration.

## 6. Appendix: Proof for Equation 7

Let $\delta_i$ be the shortest distance between two slots with $\prod_{j=0}^{i-1} \overline{p_j} = 1$.

Let us assume that $g_i^t = 1$. According to Equation 5, this means $p_i^t = 1$ and $\prod_{j=0}^{i-1} \overline{p_j^t} = 1$. According to Equation 6, we get $p_i^{t+1} = 0$. In addition, $\forall d | 1 \leq d < \delta_i, \prod_{j=0}^{i-1} \overline{{p_j}^{t+d}} = 0$ and thus $p_i^{t+d+1} = 0$.

Now, $\prod_{j=0}^{i-1} \overline{p_j^{t+\delta_i}} = 1$. Since $p_i^{t+\delta_i} = 0$, we get $g_i^{t+\delta_i} = 0$ (Equation 5). Moreover, Equation 6 tells: $\forall d | \delta_i + 1 \leq d < 2\delta_i, \prod_{j=0}^{i-1} \overline{p_j}^{t+d} = 0$ and thus $p_i^{t+d+1} = 1$. As a result, $g_i^{t+2\delta_i} = 1$. This proves that the shortest distance between two slots with $g_i = 1$ is $2\delta_i$.

Additionally, $\forall d | 2\delta_i \leq d < 3\delta_i, \prod_{j=0}^{i-1} \overline{p_j}^{t+d} = 0$ and $p_i^{t+d+1} = 0$. Then the distance with two slots with $\prod_{j=0}^{i} \overline{p_j} = 1$, i.e $\prod_{j=0}^{i-1} \overline{p_j} = 1$ and $p_i = 0$, also referred to as $\delta_{i+1}$, is given by $3\delta_i - \delta_i$: $\delta_{i+1} = 2\delta_i$. As a consequence, the shortest distance between two slots with $g_{i+1} = 1$ is twice the distance between two slots with $g_i = 1$.

Now, Equations 5 and 6 for $G_0$ can be simplified into: $g_0^t = r_0^t.p_0^t$ and $p_0^{t+1} = \overline{p_0^t}$. This gives: $\delta_0 = 2$. By induction, we get: $\delta_i = 2^{i+1}$.

## References


[1] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-time scheduling using credit-controlled static-priority arbitration. *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '08)*, pages 3–14, 2008.

[2] A. Andrei, P. Eles, Z. Peng, and J. Rosen. Predictable implementation of real-time applications on multiprocessor system-on-chip. *International Conference on VLSI Design*, pages 103–110, 2008.

[3] C. Ballabriga and H. Cassé. Improving the first-miss computation in set-associative instruction caches. *Euromicro Conference on Real-Time Systmes (ECRTS '08)*, 2008.

[4] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: an open toolbox for adaptive wcet analysis. In *IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, 2010.

[5] R. Bourgade, C. Rochange, M. de Michiel, and P. Sainrat. MBBA: a multi-bandwidth bus arbiter for hard real-time. In *5th Int'l Conference on Embedded and Multimedia Computing (EMC)*, 2010.

[6] M. de Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.

[7] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1997.

[8] J. Gustafsson, B. Lisper, A. Betts, and A. Ermedahl. The mälardalen wcet benchmark: Past, present and future. In *10th Workshop on Worst-Case Execution Time Analysis*, 2010.

[9] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization*, 2001.

[10] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, and Tools for Real-time Systems*, 1995.

[11] M. Paolieri, E. Quinones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. *Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09)*, pages 57–68, 2009.

[12] C. Rochange and P. Sainrat. A context-parameterized model for static analysis of execution times. *Transactions on HiPEAC, Springer*, 2(3), 2007.

[13] J. Staschulat and M. Bekooij. Dataflow models for shared memory access latency analysis. *Proceedings of the seventh ACM international conference on Embedded software (EMSOFT '09)*, pages 275–284, 2009.

[14] E. Wandeler and L. Thiele. Optimal tdma time slot and cycle length allocation for hard real-time systems. *Proceedings of the 2006 Asia and South Pacific Design Automation Conference (ASP-DAC '06)*, pages 479–484, 2006.