Predictable Two-Level Bus Arbitration for Heterogeneous Task Sets

Roman Bourgade, Christine Rochange, Pascal Sainrat

Predictable Two-Level Bus Arbitration for Heterogeneous Task Sets

Roman Bourgade, Christine Rochange, and Pascal Sainrat

IRIT - University of Toulouse, France
{bourgade,rochange,sainrat}@irit.fr

Abstract. In a multicore processor, arbitrating the shared resources so as to ensure predictable latencies for hard real-time tasks is challenging. In [1], we have introduced a two-level bus arbitration scheme that fits the needs of heterogeneous task sets, when some tasks have a higher demand to memory than others. In this paper, we show how this scheme can be used to optimise the overall utilisation of the cores while enforcing the schedulability of the whole task set. Our approach both configures the bus arbiter and maps the tasks onto the cores. Experimental results show that it reduces the global utilisation of the cores compared to the traditional round-robin scheme.

Keywords: Real-time, multicore, bus arbitration, task mapping, task scheduling.

1 Introduction

Multicore processors (CMP or chip multiprocessors) are becoming essential in the design of constrained embedded systems due to their high performance and efficiency in terms of power consumption, thermal dissipation, cost. This efficiency is reached by sharing resources among the cores. For example, in a typical medium-scale multicore, the cores share a bus to the highest levels of the memory hierarchy.

Resource sharing engenders conflicts between cores trying to accessing the same resource simultaneously. A consequence is that the resource latency seen by each core is higher than it would have been in a single-core processor. In real-time systems, this can be acceptable if and only if a safe upper-bound of the latency is known: this bound is required to determine the worst-case execution times (WCETs) of critical tasks.

In this paper, we focus on shared buses. Several schemes have been studied in the past to perform time-predictable bus arbitration, i.e. arbitration that makes it possible to predict the worst-case bus latency seen by a core [2][3][4]. In a recent paper, we have introduced a two-level scheme that was specially designed to support heterogeneous task sets, in which tasks exhibit various levels of demands to the shared bus [1]. With this scheme, the cores undergo different latencies, and the ones that see the shortest latencies should host the highest demanding tasks. In this paper, we propose an approach to exploit such a scheme,

i.e. to select the best arbiter configuration and to map the tasks onto the cores, so that the overall core utilisation is minimised.

We report a reduction of the global utilisation of up to 29.1% compared to a solution with the traditional round-robin algorithm. In addition, we show that our two-level bus arbiter combined to our mapping approach may allow mapping a given task set on a smaller number of cores compared to round-robin.

The paper is organised as follows. Section 2 gives an overview of related work. Our bus arbiter is described in Section 3 and the mapping approach is introduced in Section 4. Section 5 reports experimental results. We conclude the paper in Section 6.

2 Related Work

A real-time aware bus arbiter allows computing the worst-case latency for a given core to be granted the bus. Several such schemes have been proposed in the last years. In [2], a round-robin algorithm is considered. Each core is granted the bus in turn, then the maximum delay it can undergo when requesting the bus is a linear function of the total number of cores. The worst-case latency is predictable and identical for all the cores. This approach fits homogeneous workloads. However, for heterogeneous task sets, it may be desirable to fasten the execution of high demanding tasks by granting them the bus more often than less demanding tasks. For parallel applications with inter-task dependencies, it may also be needed to accelerate the tasks on the critical path.

Time-Division Multiple Access (TDMA) policies allocate time slots for bus access to the cores. The bus schedule is determined off-line [3][4]. However, it may be hard to determine with enough accuracy whether an access to the bus falls within a slot allocated to the hosting core or not, at WCET analysis time. To predict latencies, the alignment of basic block time-stamps to the allocated bus slots may be analyzed [5]. However, it makes the WCET analysis of one task dependent on the bus scheduling and thus on the co-running tasks. We instead aim at keeping the worst-case bus latencies for any task independent of the other tasks. This simplifies the analysis and favours timing composition.

3 Time-Predictable Bus Arbitration

A round-robin bus arbiter generates the same worst-case latency for each core in a shared-bus multicore: $N \times L$, where N is the number of cores and L the latency of the longest bus transaction. This latency, computed by considering that each core is permanently requesting the bus, is generally overestimated but safe. This may be acceptable when all the tasks exhibit homogeneous bus demands. Otherwise, its impact might be negligible for low-demanding tasks but disastrous for high-demanding tasks. For this reason, we have introduced a new bus arbitration scheme that enforces different worst-case latencies for the cores: some cores see a low latency and should host the most-demanding tasks, while other cores undergo longer latencies and should run low-demanding tasks [1].

Figure 1 gives an overview of our two-level arbiter. The cores are organised into groups, and the L1 arbiter grants permission to one group. Then the L2 arbiter selects one core in this group. The worst-case latency to be considered for the tasks running on a given core depends on: (a) the number of groups and the L1 arbitration policy; and (b) the number of cores in each group and the L2 arbitration policy.

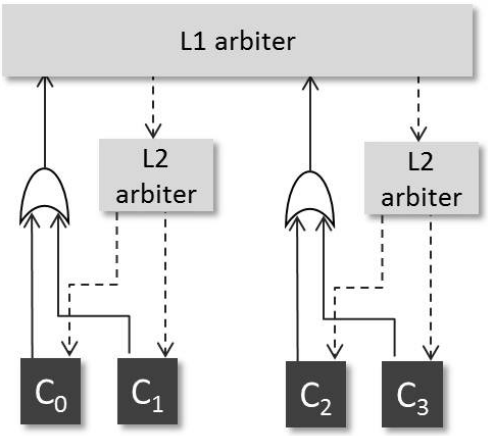

Fig. 1. Two-level bus arbiter

For the sake of simplicity, we consider a round-robin algorithm in level L2. This means that the cores that belong to the same group see the same latency, which can be computed as: $L_c = n_c \times L_g$, where n_c is the number of cores in the group and L_g is the worst-case delay for the group to be selected by the L1 arbiter.

We consider two possible policies for the L1 arbiter: (a) the round-robin algorithm, and (b) an original scheme which we call *Geometric Latencies*. The resulting two-level schemes are denoted *GRR* and *GGL*, respectively.

For *GRR*, the worst-case bus latency for a group is given by $L_g = n_g \times L$, where n_g is the number of groups. As a result, the worst-case latency seen by one core is $L_c = n_c \times n_g \times L$.

The *Geometric Latencies* arbitration policy generates worst-case latencies that follow a geometric series: group G_0 sees a latency of $2 \times L$, group G_1 a latency of $4 \times L$, group G_2 a latency of $8 \times L$, etc. The two last groups have the same latency. In [1], we provide a formal specification of the scheme that also describes its possible implementation in logic. The worst case latency of group G_i ($\forall i < n_g - 1$) is given by $L_{g_i} = 2^{i+1} \times L$, and $L_{g_{(n_g-1)}} = 2^{n_g-1} \times L$ (see proof in [6]).

With the *GGL* scheme, the latency to be considered for a task running on one core in group G_i is then:

$$\begin{cases} L_c = n_{c_i} \times 2^{i+1} \times L \quad for\ 0 \leq i < n_g - 1 \\ \\ L_c = n_{c_i} \times 2^{n_g - 1} \times L \quad if\ i = n_g - 1 \end{cases} \tag{1}$$

where n_{c_i} is the number of cores in group G_i.

To summarise, the worst-case latency seen by one core depends on the number of cores in the same group in both schemes. In addition, it depends on the group's rank in *GGL*.

Configuring our two-level arbiter means mapping the cores to groups. The arbiter configuration impacts the bus latency seen by each core, and then the worst-case execution times of the tasks it runs. In next section, we introduce an algorithm to select an arbiter configuration together with a mapping of tasks to cores that both minimise the overall utilisation of cores while ensuring the schedulability of the whole task set.

4 Task Mapping Optimisation

The problem addressed in this section breaks down into: (a) determining the best configuration of our bus arbiter for a given task set; (b) finding out the best mapping of tasks to cores considering this configuration. Both should be solved simultaneously.

Algorithm 1. Computation of time/utilisation vectors

```
1  for n ← 1 to num_cores do
2  |  Γ_n ← PossibleConfigurations(n);
3  |  Λ_n ← ∅;
4  |  foreach γ ∈ Γ_n do
5  |  |  Λ_n ← Λ_n ∪ PossibleLatencies(γ);
6  |  end
7  |  foreach task τ ∈ T do
8  |  |  foreach λ ∈ Λ_n do
9  |  |  |  t_τ^λ ← WCET(τ, λ);
10 |  |  |  u_τ^λ ← t_τ^λ / P_τ;
11 |  |  end
12 |  end
13 end
```

4.1 Preliminary Computations

The WCET of a task, and thus its core utilisation, depends on the bus latency seen by the core that hosts it, which in turn depends on the arbiter configuration. Algorithm 1 computes the WCET and utilisation of each task, considering each possible value of the bus latency (the possible values result from the various possible arbiter configurations). These results are later used to analyse the schedulability of the task set.

4.2 General Approach to Task Mapping

The problem of mapping tasks together with configuring the two-level arbiter exhibits several degrees of liberty: L1 arbitration policy (GRR or GGL), number of groups, number of cores, number of cores in each group, assignment of each task to a core. Only schedulable solutions should be retained.

Our approach is shown in Algorithm 2. To keep the problem resolution tractable, the problem is split into several sub-problems. Each sub-problem consists in finding out the best task mapping for a given arbiter configuration (*line 5*): our approach is explained in Section 4.3. If one solution is found, it is added to the set of possible solutions (*line 7*). The set of possible configurations is exhaustively scanned in the two outer loops (*lines 2 and 4*). As we will see, splitting the whole problem into sub-problems allows using Integer Linear Programming (ILP) techniques.

Algorithm 2. General approach to task mapping

```
1  M ← ∅;
2  for n ← 1 to num_cores do
3  |  Γ_n ← PossibleConfigurations(n);
4  |  foreach γ ∈ Γ_n do
5  |  |  (schedulable, m) ← FindSolution(γ, T);
6  |  |  if (schedulable) then
7  |  |  |  M ← M ∪ {γ, m};
8  |  |  end
9  |  end
10 end
```

4.3 Task Mapping for a Given Arbiter Configuration

Overview. Algorithm 3 describes the process of searching the best task mapping for a given arbiter configuration. We consider the aggregated utilisation of the cores (i.e. the sum of the individual utilisations) as the primary criterion to estimate the quality of a particular mapping. A lower global utilisation leaves computing resources to run additional tasks.

Now, to be acceptable, a task mapping must also be globally schedulable. In this paper, we consider a partitioned scheduling strategy. Then each subset of tasks assigned to one core must be shown schedulable on this core. In the following, we assume non-preemptive EDF scheduling and we use the related schedulability test [7].

In Algorithm 3, a loop (*line 5*) iterates until a schedulable solution has been found or no other task mapping exists. In each iteration, an integer linear program is used to find a task mapping (*line 6*). If a solution is found, it is tested for schedulability on each core (*line 10*). When a subset assigned to one core is found schedulable, it is locked to this core (*line 11*). Otherwise, it is appended to the black list and the task mapping is invalidated (*lines 13-14*). In the next iteration, the integer linear program is enriched with information about the locked and black-listed subsets to search for another task mapping. Locking task subsets is a way to accelerate the mapping process, as we will see in Section 5.

Algorithm 3. Mapping of tasks for a given arbiter configuration ($FindSolution(\gamma, T)$)

```
1  foreach k ∈ [1..n] do
2  |  B_k ← ∅; L_k ← ∅;
3  end
4  mappable ← true; schedulable ← false;
5  while mappable && !schedulable do
6  |  (mappable, m) ← Map(n, T, B, L);
7  |  if mappable then
8  |  |  schedulable ← true;
9  |  |  foreach k ∈ [1..n] do
10 |  |  |  if IsSchedulable(m_k) then
11 |  |  |  |  L_k ← m_k;
12 |  |  |  else
13 |  |  |  |  B_k ← B_k ∪ m_k;
14 |  |  |  |  schedulable ← false;
15 |  |  |  end
16 |  |  end
17 |  end
18 end
19 return (schedulable, m);
```

Basic ILP Formulation of the Task Mapping Problem. The mapping of a task set onto cores for a given arbiter configuration γ is described by the following set:

$$\{\mu_{\tau,k} | \tau \in T, 0 \leq k < n\}$$

with: $\mu_{\tau,k} = \begin{cases} 1 \; if \; task \; \tau \; is \; mapped \; to \; core \; k \\ 0 \; otherwise \end{cases}$

Let $\lambda_{k,\gamma}$ be the bus latency for core k considering configuration γ. The ILP formulation is the following:

	$min : \mathcal{U} = \sum_{k=0}^{n} \mathcal{U}_k$	/* objective: minimising the global utilisation */
$\forall k \vert 0 \leq k < n,$	$\mathcal{U}_k = \sum_{\tau \in T} \mu_{\tau,k}.u_{\tau}^{\lambda_{k,\gamma}}$	/* utilisation of one core is the sum of the utilisations of the tasks it runs */
$\forall k \vert 0 \leq k < n,$	$\mathcal{U}_k \leq 1$	/* utilisation of one core cannot exceed 1*/
$\forall \tau \in T,$	$\sum_{k=0}^{n-1} \mu_{\tau,k} = 1$	/* a task cannot be mapped on several cores */

Additional Constraints for Locked Task Subsets and Black Lists. These constraints are used to accelerate the search for a schedulable task mapping. They avoid exploring new possible tasks subsets when one schedulable subset has been found for a given core. In addition, they avoid considering subsets that have already been shown unschedulable.

$\forall k \vert 0 \leq k < n, \forall \tau \in \mathcal{L}_k,$	$\mu_{\tau,k} = 1$	/* locked task τ is mapped onto core k */
$\forall k \vert 0 \leq k < n, \forall \tau \notin \mathcal{L}_k,$	$\mu_{\tau,k} = 0$	/* locked task τ is not mapped onto core k */
$\forall k \vert 0 \leq k < n,$	$\sum_{\tau \in \mathcal{B}_k} \mu_{\tau,k} < \vert\mathcal{B}_k\vert$	/* tasks in the blacklist for core k cannot be mapped together on core k */

5 Experimental Results

5.1 Methodology

In the following, we consider an 8-core architecture, with in-order 2-way superscalar cores supporting the PowerPC ISA. Each core has a 2-Kbyte 2-way associative level-1 instruction cache with 16-byte cache lines. We consider a perfect level-1 data cache [1].

We consider a 32-bit bus, with a bus latency of 1 cycle. The memory latency is 5 cycles for the first word of a cache line and one additional cycle for each subsequent word.

Our task set includes 32 tasks, that is four instances of each of the tasks described in Table 1. They belong to the Mälardalen Benchmark Suite [8] (`nsischneu` and `statemate`), the SPEC95 suite[2] (`compress`) and the MiBench

[1] This assumption is only due to the fact that our WCET analysis does not completely handle data caches. We also have performed experiments considering no data cache, which are not reported here for the sake of clarity, and we have found similar conclusions to the ones drawn with perfect caches.

[2] `www.spec.org`

suite [9] (`susan`). These tasks have been selected because of their heterogeneous demands to the bus. Figure 2 show the variability of the WCET for each of these tasks as a function of the bus latency. The reference value is the WCET found considering the 8-core round-robin scheme (that gives a 73-cycle latency). The latency values are those observed in various configurations of our arbiter, as will be shown later.

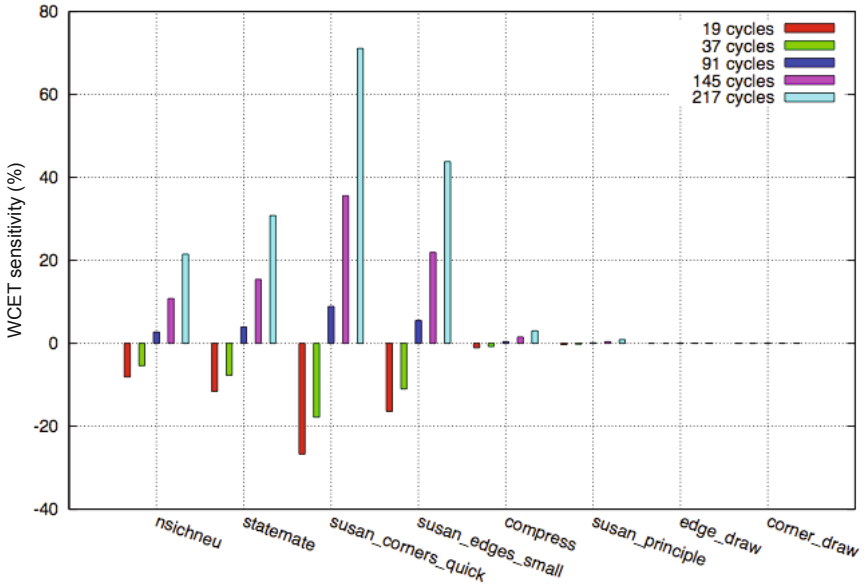

Fig. 2. Sensitivity of the tasks WCETs to the bus latency

For each task, considered as a real-time task, a period must be specified. In this paper, we consider a uniform core utilisation among the tasks. The utilisation of a task is computed as the ratio of its (worst-case) execution time to its period. To choose its value, we have performed some experiments considering a round-robin bus arbiter. We have observed that our task set is schedulable if the utilisation of tasks is not higher than 0.21. Since the schemes we propose aim at performing better than the round-robin algorithm, we have decided to use this value for the rest of the experiments. Then, the period of each task (used to decide on the task set schedulability) is computed as the ratio of its WCET to 0.21.

The worst-case execution times are analysed using our OTAWA/oRange toolset [10][11]. It implements static analysis techniques to build a representation of the binary code of the application (a Control Flow Graph), to determine flow facts (e.g. loop bounds), to derive the worst-case execution costs of basic blocks, and to determine the global WCET with the IPET technique [12].

Table 1. Benchmark tasks

Task name	Function
nsischneu	Simulation of an extended Petri net
statemate	Automatically generated code (STARC tool)
compress	Data compression
susan_corners_quick susan_edges_small susan_principle edge_draw corner_draw	Image processing (SUSAN)

The algorithms presented in this paper are implemented in Perl and the integer linear programs are solved with the CPLEX tool[3].

5.2 Results

Quantitative results for eight cores are provided in Table 2. For each configuration, described by the number of cores in each group (limited to three groups), it gives:

- the latency seen by any core in each group;
- the minimum global utilisation and the number of iterations needed to find it (see Algorithm 3), for both schemes (GRR and GGL).

When several configurations enforce the same values for bus latencies, only one of them has been considered. Doubles do not appear in the table, or with their latencies italicised. Configurations for which no schedulable mapping could be found are marked *n.s.*

Reference Value. The first configuration (all the eight cores in group G_0) enforces the same latency for each core; it is equivalent to the traditional round-robin scheme. The corresponding minimum utilisation (6.72) will then be considered as our reference value.

Performance of the GRR and GGL Schemes. Our two-level schemes both help in lowering the global utilisation of the cores compared to the round-robin algorithm: GRR can reduce it by 25.1% and GGL by 29.1%. GGL configurations that have a low number of cores in the highest priority group (G_0) perform better than GRR.

In addition, our schemes may be able to map and schedule the task set on a smaller number of cores. For example, the task set considered in these experiments cannot be scheduled on six or seven cores with the round-robin scheme. But we can find schedulable mappings considering both GRR and GGL. They

[3] www.ibm.com/software/integration/optimization/cplex-optimizer/

Table 2. Minimum utilisation obtained for all the possible configurations (8 cores)

Configuration			GRR					GGL				
N_0	N_1	N_2	L_{c_0}	L_{c_1}	L_{c_2}	U_{min}	$\#iter$	L_{c_0}	L_{c_1}	L_{c_0}	U_{min}	$\#iter$
8	-	-	73	-	-	6.72	101	*73*	-	-	-	-
1	7	-	19	127	-	5.30	28	*19*	*127*	-	-	-
2	6	-	37	127	-	5.86	9	*37*	*127*	-	-	-
3	5	-	55	91	-	6.15	13	*55*	*91*	-	-	-
1	1	6	28	28	163	5.03	18	19	27	217	4.76	62
1	2	5	28	55	136	5.30	75	19	73	181	4.86	7
1	3	4	28	82	109	5.46	5	19	109	145	5.05	32
2	1	5	*55*	*28*	*136*	-	-	37	37	181	n.s.	
2	2	4	55	55	109	6.15	10	37	73	145	5.61	111
3	1	4	*82*	*28*	*109*	-	-	55	37	145	5.72	20
3	2	3	82	55	82	6.35	19	55	73	109	6.42	18
4	1	3	*109*	*28*	*82*	-	-	73	37	109	5.95	28
5	1	2	*136*	*28*	*55*	-	-	91	37	73	5.86	10

even exhibit slightly lower utilisations: 5.01 and 4.70, respectively, for seven cores.

Finally, both GRR and GGL have solutions for this task set considering an utilisation of up to 0.27 for each task. The global utilisation then reaches 6.8 for eight cores. On the contrary, the task set cannot be mapped and scheduled with the round-robin scheme if the individual task utilisation exceeds 0.21.

Impact of Locking Schedulable Task Subsets. In Algorithm3, we have introduced task subsets locking as a way of making the computation faster: as soon as the mapping algorithm finds a solution with some of the task subsets being schedulable on their assigned core, these task subsets are locked on their core. Subsequent iterations of the loop, that may be necessary if some subsets are still not schedulable, consider the locked task subsets as a starting point and focus on mapping the remaining tasks only. Our experiments have shown that this drastically limits the number of needed iterations: without this feature, as many as 7 290 iterations are needed for some configurations; the maximum number of iterations with subsets locking is lowered down to 111.

6 Conclusion

Time-predictable resource sharing is a key feature that will allow using multicore architectures for hard real-time systems. In this paper, we propose an approach

to efficiently use a time-predictable bus arbiter which was introduced in a recent paper [1]. This arbiter enforces different worst-case bus latencies for the different cores in order to meet the requirements of heterogeneous workloads. The approach presented here determines the best arbiter configuration and mapping of the tasks to the cores. Experimental results show a reduction of the multicore utilisation by more than 29% compared to a round-robin bus arbiter.

References

1. Bourgade, R., Rochange, C., Sainrat, P.: Predictable Bus Arbitration Schemes for Heterogeneous Time-Critical Workloads Running on Multicore Processors. In: Emerging Technologies and Factory Automation (ETFA). IEEE (September 2011)
2. Paolieri, M., Quiñones, E., Cazorla, F.J., Bernat, G., Valero, M.: Hardware support for wcet analysis of hard real-time multicore systems. In: Proc. 36th Annual International Symposium on Computer Architecture, ISCA 2009, pp. 57–68 (2009)
3. Andrei, A., Eles, P., Peng, Z., Rosen, J.: Predictable implementation of real-time applications on multiprocessor system-on-chip. In: International Conference on VLSI Design, pp. 103–110 (2008)
4. Wandeler, E., Thiele, L.: Optimal tdma time slot and cycle length allocation for hard real-time systems. In: Proceedings of the 2006 Asia and South Pacific Design Automation Conference, pp. 479–484 (2006)
5. Chattopadhyay, S., Roychoudhury, A., Mitra, T.: Modeling shared cache and bus in multi-cores for timing analysis. In: Proc. 13th Int'l Workshop on Software & Compilers for Embedded Systems, SCOPES 2010, pp. 6:1–6:10 (2010)
6. Bourgade, R., Rochange, C., Sainrat, P.: Predictable bus arbitration schemes for heterogeneous time-critical workloads running on multicore processors. Technical Report 2011-19, IRIT (2011)
7. Jeffay, K., Stanat, D.F.: On non-preemptive scheduling of periodic and sporadic tasks. In: Real-Time Systems Symposium (1991)
8. Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B.: The Mälardalen WCET benchmarks – past, present and future. In: Int'l Workshop on WCET Analysis (2010)
9. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: A free, commercially representative embedded benchmark suite. In: Int'l Workshop on Workload Characterization (2001)
10. Ballabriga, C., Cassé, H., Rochange, C., Sainrat, P.: Otawa: An open toolbox for adaptive wcet analysis. In: IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems (2010)
11. Michiel, M.D., Bonenfant, A., Cassé, H., Sainrat, P.: Static loop bound analysis of c programs based on flow analysis and abstract interpretation. In: Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (2008)
12. Li, Y.T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: ACM/IEEE Design Automation Conf., pp. 456–461 (June 1995)