Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems

Marco Paolieri
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
marco.paolieri@bsc.es

Eduardo Quiñones
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
eduardo.quinones@bsc.es

Francisco J. Cazorla
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
francisco.cazorla@bsc.es

Guillem Bernat
Rapita System Ltd
York, UK
bernat@rapitasystems.com

Mateo Valero
Universitat Politècnica de Catalunya and BSC
Barcelona, Spain
mateo@ac.upc.edu

ABSTRACT

The increasing demand for new functionalities in current and future hard real-time embedded systems like automotive, avionics and space industries is driving an increase in the performance required in embedded processors. Multicore processors represent a good design solution for such systems due to their high performance, low cost and power consumption characteristics. However, hard real-time embedded systems require time analyzability and current multicore processors are less analyzable than single-core processors due to the interferences between different tasks when accessing shared hardware resources. In this paper we propose a multicore architecture with shared resources that allows the execution of applications with hard real-time and non hard real-time constraints at the same time, providing time analizability for the hard real-time tasks so that they can meet their deadlines. Moreover our architecture proposal provides high-performance for the non hard real-time tasks.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems;
C.1.3 [**Computer Systems Organization**]: Processor Architectures—*Other Architecture Styles*

General Terms

Design, Experimentation

Keywords

Multicore, real-time embedded systems, hard real-time, interconnection network, cache partitioning, WCET, analyzability

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISCA'09, June 20–24, 2009, Austin, Texas, USA.
Copyright 2009 ACM 978-1-60558-526-0/09/06 ...$5.00.

1. INTRODUCTION

The correct behavior of hard real-time embedded systems like automotive, aerospace, space and construction machinery relies on meeting execution time *deadlines*[19]. In such systems missing a deadline may involve catastrophic system failures. It is then fundamental to provide worst-case timing analysis, i.e., to compute an upper bound of the application execution time called *Worst-Case Execution Time* (WCET)[19].

Providing higher performance than current embedded processors will allow hard real-time embedded systems to increase safety, comfort, and, number and quality of services[8]. Motor injection, for example, could be optimized to reduce gas consumption and/or to reduce emissions. Also, automotive safety relevant systems, like an automatic emergency braking system triggered by collision avoidance techniques, will master more complex situations evaluating more sensor signals. Such required high performance could be achieved by designing more complex processors with longer pipelines, out of order execution or higher clock frequency. However, in embedded system design these solutions are not feasible, because complex processors suffer timing anomalies[13] due to their non deterministic run-time behaviors. In addition, the high energy requirements of such complex processors do not satisfy the low-power constraints and the severe cost limitations of common embedded systems.

Multicores offer better performance per watt than single-core processors, while maintaining a relatively simple processor design. Moreover, multicore processors ideally enable co-hosting applications with different requirements (e.g. high data processing demand or stringent time criticality). Co-hosting non-safety and safety critical applications on a common powerful multicore processor is of paramount importance in the embedded system market. It would then be possible to schedule a higher number of tasks on a single processor so that the hardware utilization is maximized, while cost, size, weight and power requirements are reduced.

Even if multicore processors may offer several benefits to embedded systems, they are much harder to analyze than single-core processors, and so far a generic solution to perform WCET analysis of multicore processors has not been proposed. Multicores are harder to analyze due to inter-

thread interferences accessing shared resources[1] (e.g. shared bus or cache). Inter-thread interferences appear when two or more threads that share a resource try to access it at the same time. To handle this kind of contention, an arbitration mechanism when accessing the shared resources is required, which may affect the execution time of running threads. Therefore, a tight WCET estimation becomes extremely difficult or even impossible because the execution time of a thread may change depending on the other threads running at the same time.

Providing a WCET much longer than the WCET running alone, would result in a safe upper bound of the execution time but it would require an over-dimensioned system to handle worst-case computational requirements that most of the time would not occur. Another obvious solution would be to fully isolate the cores providing each core with full set of resources (cache, bandwidth). This would solve the problem of thread interference but it would provide low performance due to the use of static partitions, and all the advantages previously described about multicore processors would not be valid anymore. Even if we were able to redefine the WCET as the longest execution time within all possible workloads, it would be necessary to analyze a huge amount of workloads. For example, for a set of n tasks and a target processor with k cores, we should profile all $n/(k!(n-k)!)$ possible combinations. Furthermore, any change in the workload, like a shift in the time at which each application in the workload starts, would invalidate the previous analysis resulting in an unsafe WCET estimation for the remaining threads, especially when running mixed workload applications with non real-time applications[2].

A previous work already deals with the problem of analyzability in the presence of shared resources [17]. This study effectively provides WCET analyzability of multicore processors in which each thread has its own private memory. However, the authors assume that the complete workload is known before hand, which complicates the time analysis of current and future embedded systems. For example, today, most automotive electronic systems use a set of electronic control units (ECUs) based on single processors, connected together via a network such as CAN (Controller Area Network). Different sub-suppliers are responsible for the different ECUs such as transmission control, engine management, climate control, ABS (Anti-lock Braking System), stability control, etc. Typically, a sub-supplier undertakes all of the development, testing and timing analysis of the ECUs that they supply. It is well recognized that many of the most difficult and expensive problems in the development of automotive electronics occur during system integration when all of the ECUs are brought together to form a complete system. With a move to a multiprocessor architecture, it is essential that dependencies between the different sub-systems (effectively different threads running on the multicore platform) are minimized. In particular, it is vital that individual sub-suppliers are able to perform meaningful timing analysis on their application without this analysis being dependent on the software applications supplied by other companies. Without this independence changes in one application can have unexpected effects on others, making integration, timing analysis, and maintenance extremely costly if not impossible in practice. Hence a major design goal for our multicore architecture is to make the analysis of each task independent from the other tasks it may be co-scheduled with, when the system is integrated.

In particular, the major contributions of this paper are:

1. We propose a new multicore architecture in which the maximum time a request to a shared resource from a *Hard Real-time Task* (HRT) can be delayed by any other task is bounded: our multicore processor *enforces* that a request of a HRT cannot be delayed longer than a given *Upper Bound Delay* (UBD). We also analyze what is the required structure and the behavior of shared resources in a multicore so that an UBD can be determined.

2. Next, we extend our multicore architecture, which allows determining an UBD, with a novel hardware feature called *WCET Computation Mode*. In this execution mode, each HRT is run in isolation: The processor, on each access to a shared resource, artificially introduces the maximum delay that a request from HRT can suffer because of inter-thread interference, that is the UBD. As a consequence, the resulting WCET from the profile of this execution is ensured to provide a safe upper bound of the execution of the HRT when it runs in *Standard Execution Mode* together with other tasks sharing processor resources. The advantages of *WCET Computation Mode* are:

   (a) It allows computing for each HRT a safe WCET estimation that does not depend on the other co-running threads. Thus, since every request considers the UBD, when a given HRT runs in a multicore environment its execution time is upper bounded by its WCET regardless of the other tasks it is co-scheduled with.

   (b) Our solution allows changing, after the integration phase of the system, the threads of the workload and it does not require a re-estimation of the WCET of all the threads in the task set. This reduces the cost of analyzability as only the tasks involved in the change need to be re-analyzed, as opposed of having to re-analyze the whole system.

   (c) Our proposed processor architecture can be easily analyzed by current measurement based WCET tools with *no modifications*. Hence, it is possible to perform WCET analysis of a multicore processor using the same tool chain used for single-core.

   (d) Finally, our approach also allows *Non Hard Real-time Tasks* (NHRTs) to satisfy their high data processing demands. Given a mixed workload, all shared resources not required by HRTs can be used by non real-time tasks.

3. We verify our proposal using a commercial WCET analysis tool (RapiTime[2]), a commercial compiler[18], and an industry hard real-time application provided by Honeywell, in addition to representative benchmarks. In particular, we evaluate a 4-core architecture with a shared second level cache (L2) connected by a shared bus. Our proposal includes a WCET-aware bus arbitration policy that guarantees that the execution time is not going to be longer than the computed WCET

[1] By default, the term *resources* refers to hardware resources.

[2] We use the terms *application*, *thread* and *task* interchangeably.

due to bus contention; and cache partitioning using *columnization*[7] or *bankization* techniques in order to avoid *storage interferences* between different threads.

Our results show that by only increasing the WCET of HRTs between 2% and 27% respect to their WCET estimations when they run in isolation, our multicore architecture is able to execute several HRTs and NHRTs simultaneously ensuring that HRTs meet their deadlines and providing high performance to NHRTs. We also show that the performance of NHRTs ranges between 78% to 99% with respect to their performance when they run in a multicore without HRTs.

To sum up, in this paper we propose a multicore processor architecture that allows co-hosting HRTs and NHRTs running at the same time. Our proposal provides analyzability for the HRTs and it allows NHRTs to be executed into the same chip. Moreover, all current WCET analysis tools used in single-core systems can be used in our multicore processor with no changes, being extremely beneficial for the industry.

This paper is structured as follows: Section 2 introduces time analyzability in single-core architectures. Section 3 describes our multicore architecture that provides an upper bound delay on the inter-thread interferences a thread can suffer. A full description of *WCET Computation Mode* with hardware details appears in Section 4. The experimental setup and the experiments we run are addressed in Section 5 and Section 6. In Section 7 we propose how to increase system schedulability. Section 8 covers an overview of related work. In Section 9 final conclusions are considered.

2. TIME ANALYZABILITY IN SINGLE-CORE ARCHITECTURES

One of the design goals of our architecture is that it can be easily analyzed by current measurement based WCET tools with no modifications. In this section we provide some background on real-time scheduling and the analysis tool we use in this paper.

In real-time systems, for each task, the scheduler knows three main parameters: The *period*, the *deadline* and the *Worst-Case Execution Time* (WCET). If the task is periodic, its *period* is the interval at which new instances of that task are ready for execution. The *deadline* is the time before a task instance must be complete. For simplicity, the *deadline* is often set equal to the *period*. This means that a task has to be executed before its next instance arrives into the system. The *WCET* is a safe estimation of the upper bound time required to execute any instance of the task. In single-core systems the WCET of a task is computed assuming that the task has full access to processor resources.

One of the current approaches to analyze WCET in single-core processors is *measurement-based WCET analysis*[5]. In this paper we use RapiTime[2], a commercial tool developed by Rapita Systems Ltd.[3], that estimates the WCET using a measurement-based technique. This tool is widely used in the avionics, telecommunications, space, and automotive industries. RapiTime uses on-line testing to measure the execution time of sub-paths between instrumentation points in the code. Moreover, by contrast, offline static analysis is the best way to determine the overall structure of the code and the paths through it. RapiTime therefore uses path analysis techniques to build up a precise model of the overall code structure and determine which combinations of sub-paths form complete and feasible paths through the code. Finally RapiTime combines the measurement and control flow analysis information to compute measurement based worst-case execution time estimations in a way that captures accurately the execution time variation on individual paths due to hardware effects[5].

[3] www.rapitasystems.com

3. TOWARDS AN ANALYZABLE MULTI-CORE PROCESSOR

Multicore processors have an important drawback which could make it difficult or even *impossible* to use them in real-time systems: The *inter-thread interferences*. Inter-thread interferences appear when several threads that share a resource try to access it at the same time, so an arbitration mechanism is required.

In this section we describe in detail one of the major contribution of this paper: A new multicore architecture in which the maximum time a request from a HRT can be delayed by any other task is bounded and it can be determined. Our multicore processor *enforces* that a request of a HRT cannot be delayed longer than a given *Upper Bound Delay* (UBD). This is a necessary feature to make a multicore architecture analyzable and so WCET computable.

For the purpose of this paper we consider a multicore processor in which each core, that has its private data and instruction first level cache, is connected to a second level shared cache through a shared bus. Both shared resources are the main source of inter-thread interference in our architecture. It is important to notice that although this paper focuses only on bus and cache interference, our solution can be applied to other hardware shared resources present not only in multicore processors but also in other processors such as simultaneous multithreading.

In our architecture and along this paper, we assume that the arbiter needs 1 cycle to select which request accesses the bus (labeled as A in following Figures), 2 additional cycles to send the data through the bus (labeled as B in Figures and L_{bus} in Formulas) and 4 cycles to access a bank (an access to bank n is labeled M_n in Figures and L_{bank} in Formulas).

3.1 Interference-Aware Bus Arbiter

Buses have a *latency*, that is a fixed amount of time necessary for a request/access to cross them. When one request is granted access to the bus, no other request can use it, so an arbitration policy is required if two requests try to access the bus at the same time. In this case, one thread will delay the execution of the other one until it frees the bus, producing a *bus interference*.

In Figure 1 we show an overall picture of our interference-aware bus arbiter that controls bus interferences when computing the WCET. Our proposal splits the bus arbiter into two hierarchical components: The *Inter-Core Bus Arbiter* (XCBA) that schedules among requests from different cores, and several *Intra-Core Bus Arbiters* (ICBAs), one per core, which schedules among requests from the same core. The idea behind our design is to ensure that the delay that a thread can suffer due to requests of any other thread is bounded by a fixed amount of time.

In our architecture, a thread sends a request to the bus on (1) every data cache load miss, (2) instruction cache miss and (3) store operation. These requests are handled by its

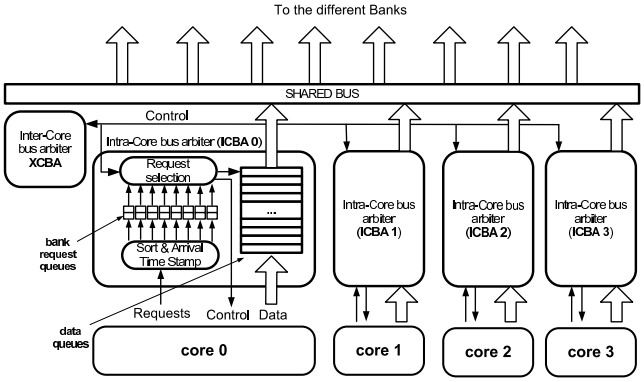


Figure 1: Our interference-aware bus arbiter

corresponding ICBA, which selects the next memory request to be sent to the XCBA. Hence, by maintaining the requests of the different cores apart, the execution time, and so the WCET of a task does not depend on the number of request from the other tasks that are ready and waiting to be granted access to the bus. The XCBA is in charge of deciding which of those requests from different cores access the bus.

In order to accomplish with our second objective of providing high performance, in each ICBA the requests to the cache are placed in different *bank request queues*. There is a bank request queue per L2 bank, which holds requests based on their target destination bank. Bank request queues contain the information of the memory request and the index to the data buffer entry that stores all the data to transfer with that request. Thus, once a core sends a request, the ICBA time-stamps it and inserts it into its corresponding bank request queue. The ICBA implements a given policy, which selects the next memory request that is forwarded to the XCBA (that sends it to the bus). In particular, the ICBA applies the following policies among requests from different bank queues. In the case of NHRTs, we allow parallel out of order execution of different cache requests that do not address the same bank in order to increase the overall performance, that is, we apply a *First Ready First Serviced policy*. In the case of HRTs, in order to prevent timing anomalies[13] we apply a FIFO policy, so the oldest request in all bank request queues is selected.

Our ICBA splits wide bus transfers into independent request so they can be sent in non-consecutive bus slots. We allow this way bus transfers wider than the bus bandwidth. A wide bus transfer is complete when the last request has been sent.

3.2 Analyzing the Effect of Different Bus Arbitration Policies

The delay a thread can suffer due to bus interferences depends on the bus arbitration policy. In this section we analyze the variation that a shared bus can introduce on the execution time of different tasks. We also show how XCBA enforces that a request from a given task cannot be delayed longer than UBD, and we determine formally UBD's value. Through this section we assume that each task has its own piece of memory, so tasks are only affected by bus interferences. In Section 3.3 we study an architecture in which tasks share both the bus and the cache.

3.2.1 Scheduling One Hard Real-Time and Several Non Hard Real-Time Threads

In a mixed workload composed of only one hard real-time thread and $N-1$ non hard real-time threads, bus interferences could be avoided by flushing the requests from the NHRT. That is, if the HRT requires the bus and it is being used by a NHRT, the requests from the NHRT can be flushed, so that the bus arbiter immediately grants access to the HRT without introducing any extra delay to its execution time. This technique is too costly in terms of power consumption since it requires re-sending the flushed request and cannot be applied if we run several HRTs simultaneously.

Analogously, if all shared resources are fully pipelined or they have a single-cycle access no interferences occurs between the HRTs and NHRTs. In fact, if a request from a NHRT arrives at the same time as a request from the HRT, the latter is prioritized. While in the case where the request from the NHRT thread arrives one cycle before than the request from the HRT, the latter can proceed as the resource is available in that cycle.

In a more realistic scenario where no flushing technique is used and the access to shared resources takes multiple cycles, the XCBA arbiter prioritizes requests from HRTs on NHRTs in order to minimize the interference of NHRTs on HRTs. Hence, if a request from the HRT and a request from a NHRT are ready at the same cycle, the arbiter prioritizes the request from the HRT. However, it may happen that the request coming from the HRT arrives just one cycle after the request from the NHRT has been already granted the access to the bus. In such a situation, the request from HRT will be delayed by the request from the NHRT (see Figure 2). In this case, the maximum delay that a request from HRT can suffer is upper bounded and can be computed by the following expression: $UBD = L_{bus} - 1$

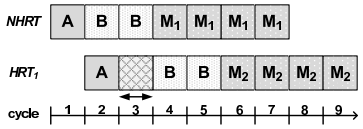


Figure 2: Example of interference between a NHRT and a HRT accessing the bus

3.2.2 Scheduling Several Hard Real-Time Threads

In a more realistic scenario, in which we have more than one HRT running at the same time inside the processor, it may happen that two or more requests from different HRTs try to access the bus at the same time. In this case, there is not always an upper bound on the time one HRT can delay the other to access the bus. The existence of such an upper bound depends on the arbitration policy. We are going to use three different XCBA arbitration policies for illustrative purposes. *Thread Prioritization* always gives priority to requests that are generated from the highest priority HRT (or a set of HRTs). *Round Robin* assigns the same priority to all the requests from HRTs. Finally, *FIFO* prioritizes the requests in arrival order to the arbiter.

When a *thread prioritization* is used the maximum delay a thread can suffer is not bounded. As shown in Figure 3, in cycle 0, a request from each HRT is ready. In cycle 1, the arbiter prioritizes requests from HRT_1, so HRT_2 is stalled until HRT_1 finishes. However, before leaving the bus an-

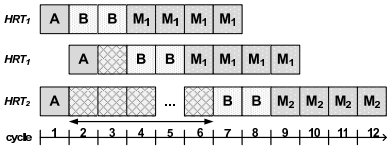


Figure 3: *Thread prioritization.* The delay that HRT_1 produces over HRT_2 is unbounded

other request from HRT_1 becomes ready in cycle 2. In this situation, the amount of time HRT_2 needs to wait to get access to the bus depends on the total time the HRT_1 has requests ready. Thus, the UBD that HRT_2 suffers due to interferences with HRT_1, depends on HRT_1. Even though this UBD can be computed knowing HRT_1 sequence of accesses, it can be too long/pessimistic to be useful.

With *round robin*, the maximum delay a request from a HRT can suffer is bounded by the total number of HRTs that can send a request at the same time. Figure 4 shows an example of such worst-case scenario that occurs when two requests from two different HRTs become ready at the same time. In this case, a given HRT, let's say HRT_2 must wait until the previous request from HRT_1 finishes. The maximum delay HRT_2 suffers, is: $UBD = (N_{HRT} - 1) \cdot L_{bus}$, N_{HRT} is the number of HRTs running at the same time in the processor, which is upper bounded by the number of cores.

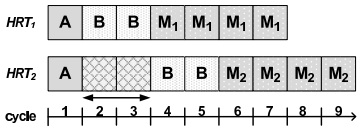


Figure 4: Worst-case scenario between two HRTs

Finally, if a *FIFO* policy is applied, the maximum delay a request from a HRT can suffer is bounded by the total number of HRTs, that can send a request at the same time, times the number of entries in the request queues each thread is allowed to have.

3.2.3 Overall Effect of Bus Arbitration

Our multicore uses a round robin bus arbitration policy between HRTs and prioritize them over NHRTs. The maximum delay is determined by the combination of the effects of the requests coming from HRTs and NHRTs:
$UBD = L_{bus} - 1 + (N_{HRT} - 1) \cdot L_{bus}$. It can be simplified as follows:

$$UBD = N_{HRT} \cdot L_{bus} - 1 \quad (1)$$

We want to highlight that the N_{HRT} is the number of HRTs running at the same time inside the processor (and not the total number of HRTs that form the system), which is upper bounded by the number of cores.

Therefore, by using *round robin* policy the maximum delay that a request will suffer due to bus inferences does not depend on previous knowledge of the workload (task set), but only on the total number of HRTs that are going to be executed simultaneously inside the multicore processor. Moreover, the WCET analysis of each thread can be performed in isolation, since by design our architecture ensures that the UBD of Formula (1) is never going to be violated. The requests to the bus from a HRT will never be delayed longer than UBD due to the interactions with the other threads, regardless of the workload.

In conclusion, to guarantee a bounded inter-thread interference delay when running in a mixed application workload, the XCBA applies the following policy to select the next memory request that will access the bus. First, the requests from HRTs have priority over requests from NHRTs. Second, between different requests from HRTs, a round robin policy is applied as well as between different requests from NHRTs. Having more than one pending request from the same thread it does not affect inter-thread interferences but it is a concern of single-core WCET analysis[5].

Although not shown in Figure 1, our bus is full-duplex and the same principle is applied for the bus arbiter that controls the requests that go from second level cache to the corresponding core, i.e., load misses and instruction misses. Hence, an ICBA for each core, as well as a global XCBA is required to control the request from L2 to cores. In this case the L2 banks insert the request into the ICBAs while the cores are the destinations of those requests. Notice that requests from cores to L2 banks do not interact with requests from L2 banks to cores and vice versa.

3.3 Analyzing the Shared Cache

In this section, we consider a more realistic multicore scenario in which threads can suffer interference delays from two shared resources: Bus and second level cache. The bus acts as the connection between cores and L2 banks. The use of shared memories in multicore systems introduces unpredictable and not analyzable worst-case behavior due two factors: *Bank access interference* and *storage interference*. In this section we will focus on addressing both problems, enabling multicore processors to become analyzable.

3.3.1 Bank Access Interference

Caches are normally partitioned into multiple banks to enable parallel operations, i.e., different memory operations can access different banks simultaneously. However, a bank can only handle one memory request at a time. When a bank is serving a memory request, it is inaccessible to any other request for an amount of cycles equal to the bank latency. So, if two memory requests try to access the same bank at the same time, the bus arbiter avoids any conflict by delaying the second access. This kind of effect, called *bank interference* or *bank conflict*, may introduce variability in the execution time of a thread.

An example of a bank conflict is shown in Figure 5. Two threads, HRT_1 and HRT_2, want to access the same memory bank (labeled as M_1) at the same time. Since we assume a memory latency of 4 cycles, HRT_2 turns out to be delayed 4 cycles because of a previous request from HRT_1.

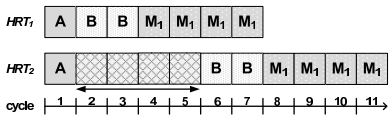


Figure 5: Bank conflict example between two HRTs

In order to control the execution time variation caused by bank interference, we apply the same principle used to avoid bus interference, i.e., determining the maximum delay a memory request can suffer because of bank interference. Assuming the same arbitration policy presented in Section 3.2, the UBD is determined combining the effects of the requests coming from HRTs and NHRTs. On the one hand, the maximum delay a request from a HRT request can suf-

fer because of NHRTs appears when the former arrives just one cycle after the latter was granted the bus (if both arrive at the same time, the request of the HRT has priority and so it does not suffer any delay). In this case the maximum expected delay is: $UBD = L_{bank} - 1$.
On the other hand, the maximum delay that a request from a HRT can suffer because of other HRTs occurs when it must wait until all other HRT requests finish. In this case the maximum expected delay is: $UBD = (N_{HRT} - 1) \cdot L_{bank}$. Hence, by combining both effects, the maximum delay results in $UBD = L_{bank} - 1 + (N_{HRT} - 1) \cdot L_{bank}$. This can be simplified as follows:

$$UBD = N_{HRT} \cdot L_{bank} - 1 \quad (2)$$

As in Formula (2), N_{HRT} is the number of HRTs running at the same time inside the processor.

Notice that, as it is commonly the case, a bank access takes longer than accesses to the bus: We consider a L_{bus} of 2 and a L_{bank} of 4 cycles. Hence, the bus latency is overlapped when accessing the bank, as shown in Figure 5 (cycles 6 and 7). For that reason, the bus latency does not appear in the formula. However, if the bank latency would be smaller than the bus latency, the bank conflict effect would be hidden because the time required to access a bank would be overlapped by the bus latency. In general Formula 2 can be expressed as $UBD = N_{HRT} \cdot max(L_{bank}, L_{bus}) - 1$

3.3.2 Storage Interferences

Storage interferences appear in shared memory schemes when one thread evicts data of another one, potentially delaying the execution time of the second thread. Such time variation makes WCET estimation harder or even infeasible.

Cache locking[16] helps to make caches more analyzable. This technique provides hardware support in order to allow the software to control which cache lines can not be modified by the replacement policy, locking the most frequently used cache lines, and reducing storage interferences. However, this technique requires knowing the whole memory footprint of a thread, being hard to implement in multicores. In such case it is necessary to consider all threads that can be co-scheduled into the processor at the same time, in order to prevent that two tasks lock the same lines at the same time. Caches using locking techniques have been shown to have similar behavior of scratchpad memories[16].

Cache partitioning is a well known technique that eliminates completely storage interferences by splitting the cache into private portions, each assigned to a different thread. Our mixed workload environment can benefit from cache partition: Storage interferences between HRTs are avoided by assigning them different partitions of the cache, while non real-time threads can share the same part of the cache. In this paper, we study two different cache partition techniques controlled via software: *Columnization* and *bankization*, and their effect on the WCET computation.

In *columnization*[7] the cache is partitioned into ways, giving to each thread a subset of the total number of ways that no other thread can use. In fact this technique only varies the replacement policy: A thread can read all the ways but evicts data only in the assigned ways. Cache partitions at level of ways can be implemented with column caching[7]: A bit vector specifies the set of columns (ways) assigned to a given thread. The replacement algorithm is modified to limit replacement to the columns specified by the bit vector.

In *bankization* the cache is partitioned into banks, giving to each thread a subset of the total number of banks that no other thread can use. In any cache access it is required to remap the destination bank of a memory request to one of the banks assigned to the thread. The additional hardware necessary to implement bankization, is based on a *Bank Remapping Unit* (BRU) that computes the target L2 bank given the thread identifier and the memory address, as shown in Figure 8 (Page 7). The information regarding the destination bank of any memory request is contained inside its address. A given range of bits of the memory address is used to select the destination L2 bank. Since bankization assigns a subset of the L2 banks to a given thread, it is necessary to remap the destination bank of a memory request to one of the bank assigned to the thread. The BRU performs this remapping, i.e., it determines the new destination bank of a memory request. The table inside the BRU (see Figure 8) is updated by the RTOS based on the subset of banks assigned to each thread. The remapping table is indexed by the thread id and the original L2 bank id of a memory request. The BRU output is the new bank id.

The main difference between columnization and bankization, in addition to their hardware requirements that will be explained in the next section, is that columnization prevents only storage conflicts since different threads can still access to the same bank. As a result, the UBD to use with columnization is the one shown in Formula (2). Meanwhile, with bankization we prevent both storage and bank access conflicts, so the UBD to use is given by Formula (1). Hence, bankization provides tighter WCET estimation than with columnization. A detailed comparison between columnization and bankization is done in Section 6.

4. COMPUTING A SAFE WCET ESTIMATION ON MULTICORE PROCESSORS

So far we have shown how our multicore architecture enforces a given UBD that a thread can suffer due to interferences with other threads accessing shared resources. In particular we have computed the UBD due to bus and cache bank interferences, ensuring that, regardless of the workload any request of a HRT will never be delayed longer than the UBD. This is a necessary feature to make a multicore architecture time analyzable.

In this section we propose a novel hardware feature: The *WCET Computation Mode*. The *WCET Computation Mode* allows computing safe WCET estimations of HRTs that are going to be executed simultaneously with other tasks on the multicore architecture. Our multicore processors has two execution modes: *WCET Computation Mode* and *Standard Execution Mode*. Our processor is set in the *WCET Computation Mode* when computing a WCET estimation for the HRT and in *Standard Execution Mode* otherwise. Next, we explain both.

4.1 The WCET Computation Mode

When analyzing a set of HRTs, the processor is set into *WCET Computation Mode* and each HRT is run in isolation. In this execution mode, the processor delays the execution of every request to a shared resource by UBD cycles. That is, once both, the request from the HRT and the shared resource (in our case the cache and the bus) are ready, the XCBA *freezes* that request by UBD cycles. By doing this, the XCBA artificially introduces the maximum delay that a request from HRT can suffer because of inter-thread in-

Inputs		Output
NHRTs	nHRTs	UBD
-	0	0
0	1	0
0	2	4
0	3	8
0	4	12
1	1	3
1	2	7
1	3	11
1	4	15

Figure 6: UBD values

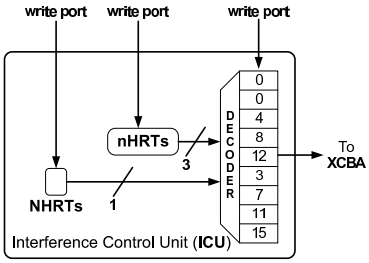


Figure 7: ICU

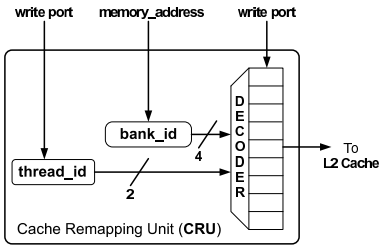


Figure 8: BRU

terferences, that is the UBD. Hence, the execution profile that results executing the HRT under the *WCET computation* mode takes into account the worst-case delay that the HRT can suffer due to inter-task interferences. This execution profile is passed to RapiTime (our WCET analysis tool) that computes a safe WCET estimation without any single change in the tool.

Once a WCET estimation has been obtained for each HRT, the processor is set back to *Standard Execution Mode*, in which no artificial delay is introduced. Our bus arbiter ensures that the execution time of a HRT that runs in a given workload formed by N HRTs running at the same time, will not be longer than its corresponding $WCET_N$. When running in *Standard Execution Mode* instructions accessing shared resources are executed before their estimated WCET, as it is not always the case that they suffer an inter-task interference. In [4, 17] it has been formally proved that executing an instruction before its estimated WCET, ensures that the WCET estimation derived running in *WCET Computation Mode* is safe. As a consequence, the WCET estimation provided by RapiTime, is a safe upper bound of the execution of the HRTs when they run in the multicore processor sharing resources with other tasks.

The UBD artificially introduced by our *WCET Computation Mode* is computed using Formula (1) or (2). The UBD depends on the total number of hard real-time threads running at the same time in the processor (N_{HRT}), that is upper bounded by the number of cores. Thus, depending on the number of HRTs the analyzed thread is going to be co-scheduled with, a different UBD value is used by the *WCET Computation Mode*, resulting in different WCET estimation values. In general, we say that a HRT that is co-scheduled at the same time with N HRTs, is analyzed using a *WCET Computation Mode* of N, which results in a WCET estimation $WCET_N$.

Our *WCET Computation Mode* allows analyzing each HRT in isolation, i.e., independently from the particular task set in which that task is going to be scheduled. For every HRT we build a WCET-matrix. The WCET-matrix has as many entries as WCET-computation modes times the number of cache partitions a thread can be assigned. In our baseline, we have a total of 25 configurations, 5 WCET-computation modes (including a configuration that disables it) times 5 cache configurations (assigning a power of 2 cache size to each HRT). This WCET-matrix can be computed in *isolation* for each HRT. This process can be easily automated, in fact we do so to run the experiments for this paper. The next step is to provide the WCET-matrix to the schedulability algorithm, which selects the best allocation of resources for the tasks in the task set. In Section 7, we elaborate more the schedulability issues of our proposal.

4.1.1 Hardware Implementation

The *WCET Computation Mode* requires extra hardware in the XCBA to store all possible UBD values (for our architecture they are shown in Figure 6) when analyzing HRT running in a *WCET Computation Mode* of N. To do so, we introduce the *Interference Control Unit* (ICU), shown in Figure 7, that contains all precomputed UBD values corresponding to each N-*WCET Computation Mode*. Moreover, in order to generate a tighter WCET estimation, we also include inside the ICU the UBDs that do not take into account the NHRTs. Hence, since N is bounded by the number of cores, the size of the ICU is limited to $2 \cdot N_{cores}$.

To sum up, the *WCET Computation Mode* works as follows. According to the number of HRTs (*nHRTs*) and whether there are NHRTs, the corresponding UBD is forwarded to XCBA. Then, the XCBA inserts such value into a down-counter that is reset every time a new request is ready. When the counter reaches zero, the request is sent through the bus, effectively delaying each request by UBD cycles. To disable the *WCET Computation Mode* and run the processor in *Standard Execution Mode*, it is necessary to set to zero the *nHRTs* register. An example of an ICU in a 4-core architecture using columnization and a bank latency of 4 cycles is shown in Figure 7. For example, when analyzing a HRT that is going to be co-scheduled with 2 more HRTs and one NHRT, the UBD required is: $UBD = N_{HRT} \cdot L_{bank} - 1$ because there is a NHRT. Thus, being $N_{HRT} = 3$ and $L_{bank} = 4$ this results in a UBD of 11 cycles. ICU can be set either by the RTOS or even by the processor vendor since it depends on the architecture.

5. EXPERIMENTAL SETUP

5.1 Architecture Simulator

All experiments presented were carried out in an in-house cycle-accurate, execution-driven simulator compatible with Tricore ISA[11]. Tricore ISA, designed by Infineon Technologies, is widely used in automotive hard real-time applications because it combines RISC, general-purpose and signal processing instructions within a single instruction set. The simulator was derived from CarCore[20]. We paid special attention to the simulator correctness, extensively validating it through a wide range of tests.

With our simulator we model a multicore architecture composed of 4-cores, a shared second level cache (L2) and a full duplex bus as interconnection network between the cache and the cores. Two interference-aware bus arbiters, described in Section 3, control the access to the bus: One handles the requests from the cores to L2 and the other handles the requests from L2 to the cores. L2 is organized in banks and in order to avoid storage interferences (see Section 3.3) is partitioned using either *bankization* or *colum-*

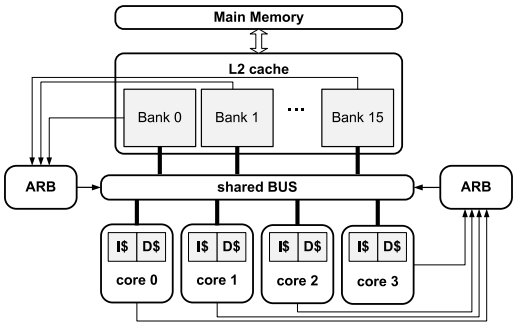


Figure 9: Our processor architecture

nization. Even though the *WCET Computation Mode* can be applied to any type of shared resource, in this paper we have focused on the bus and L2 shared cache. In this paper we have focused on on-chip shared resources. We have assumed that threads do not suffer inter-task interaction accessing the main memory. This can be achieved with an existing solution like Predator [3], i.e., a memory controller that provides a guaranteed minimum bandwidth and a maximum latency bound to the IPs. As future work we plan to implement our proposal accessing off-chip resources (like the memory) and also to integrate the solution provided in [3].

Each core implements an in-order dual-issue pipeline inspired by CarCore[20]. Each core has support for floating point operations, and a private first level cache with separated data and instruction caches. Stores do not block the pipeline and they access directly the L2 cache through the bus, unless the write buffer is full. Each core has the following characteristics: 12-stages pipeline, no branch prediction, a fetch bandwidth of 8 instructions and pre-issue bandwidth of 4 instructions. The size of the instruction buffer is 16, while the instruction window size is 8. The total number of registers is 64: 32 registers for the data and 32 for the address pipeline. The Instruction and Data L1 cache are private per each core and 8KB each (4-way, 1-bank, 8-byte per line, 1 cycle access, write-through write-not-allocate policy). The L2 cache is shared among all cores and it is 128KB (16-way, 16-banks, 32-byte per line, 4 cycles access, write-back write-allocate policy). The total number of cycles for L1 miss and L2 hit is 9.

5.2 WCET Analysis Tool

In our experiments, we used RapiTime[2] to estimate the WCET, a measurement-based tool described in Section 2. One of the main advantages of our *WCET Estimation Mode* is that it allows existing analysis tools already in use in single-core systems to be used in multicore systems without *any change*. As a matter of example, we used RapiTime (the original version unchanged) to provide WCET estimation of our HRTs running in our multicore processor.

RapiTime automatically instruments the source code (.c) of the program by inserting RapiTime_IDPoint, which is a number that identifies each basic block. Next, the program is compiled generating the executable file (.elf). When a RapiTime_IDPoint is found, the timing simulator creates a new trace line in the Rapita Trace file (.rpz) with the RapiTime_IDPoint and the timestamp of the current cycle time. The complete RapiTime trace file (.rpz) is then passed to the RapiTime tool to estimate the WCET of the task.

5.3 Benchmarks

The main goal of our research is to design a multicore architecture that allows running mixed application workloads, i.e., applications with and without real-time constraints. Hence, our proposal has been evaluated using representative benchmarks of both domains (HRTs and NHRTs). All benchmarks have been compiled with Tasking[18], an embedded commercial compiler from Altium Corporation, using maximum optimization levels.

As HRTs we use EEMBC Autobench[15], a well-known benchmark suite formed by fourteen applications widely used in both industry and academia, that reflects the current real-world demands of embedded systems. However, in order to be representative of future hard-real time applications requirements[1], we have increased their memory requirements without modifying any instruction inside the source code. Moreover, we also use a collision avoidance application as HRT provided by Honeywell Corporation. This application is based on an algorithm for 3D path planning used in autonomous-driven vehicles to process the frames captured by on-board cameras and to build the path to reach the target points avoiding the obstacles. It requires high-performance with high-data rate throughput and it is strictly hard real-time. As NHRTs we use benchmarks from MediaBench II[4],
SPEC CPU 2006[5] and MiBench Automotive[6] benchmark sui- tes, which are going to compete against HRTs for shared resources. From MediaBench II, which is a representative suite of the multimedia applications, we use *mpeg2* coder and decoder. From SPEC CPU 2006, that is an industry standardized CPU-intensive benchmark suite, we use *bzip2*. From MiBench Automotive, which includes benchmarks representative of embedded control systems, we select *susan_corners* and *qsort*.

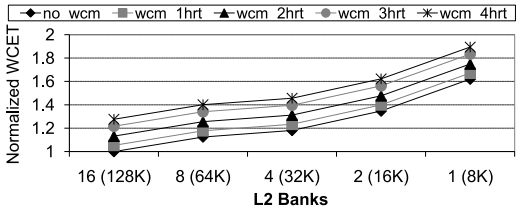
5.4 Workload Composition and Metrics

The main metrics for our experiments is the capability to provide analizability to the HRTs when running in a multicore processor: That is a safe upper-bound of the execution time of the HRTs while running together with NHRTs. The second metric is optimizing the performance of the NHRTs. In particular, we choose to optimize the total throughput as it provides a measurement of the performance per resource we can get from the architecture.

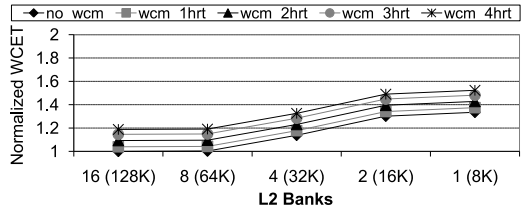
We evaluate the impact of the *WCET Computation Mode* on the WCET estimation for each HRT. We also analyze the effect of the size of the partition of the cache assigned to the given HRT. In each case, we breakdown the results according to the demand of shared resources of HRTs, so that we created 3 groups called High, Medium, Low *shared resources demanding*. We classified the EEMBC benchmarks into 3 groups as follows: High (*aifftr01*,*aiifft01*,*cacheb01*), medium (*aifirf01*, *iirflt01*, *matrix01*, *pntrch01*), low (*a2time01*, *basefp01*, *bitmnp01*, *canrdr01*, *idctrn01*, *puwmod01*, *rspeed01*, *tblook01*, *ttsprk01*).

We also run several mixed workloads in order to show that hard real-time constraints of HRTs are satisfied while providing high performance to NHRTs. To determine representative workloads we run the non hard real-time in isolation on our processor and we identified their memory utilization.

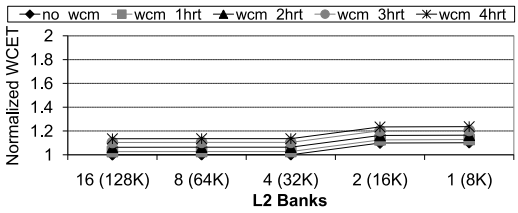
[4] http://euler.slu.edu/ fritts/mediabench/
[5] www.spec.org/cpu2006/
[6] www.eecs.umich.edu/mibench



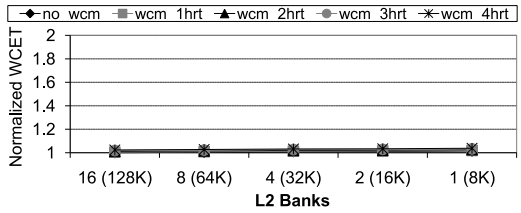
(a) High demanding HRTs (bankization)



(b) Medium demanding HRTs (bankization)



(c) Collision Avoidance App. (bankization)



(d) Low demanding HRTs (bankization)

Figure 10: WCET estimation of different HRTs (using bankization)

Our goal was to select memory-hungry NHRTs so that they can interfere as much as possible with the HRTs. For this paper, the workloads we built are composed of 2 HRTs running on 2 cores and 2 NHRTs running on the other 2.

6. RESULTS

This section evaluates the proposed *WCET Computation Mode* and compares the two cache partitioning techniques: Bankization and columnization. All WCET estimation values have been obtained using RapiTime and normalized to the WCET estimation when the hard real-time task runs in isolation with all the hardware resources in the multicore architecture and the *WCET Computation Mode* disabled.

6.1 WCET Evaluation

For each benchmark we compute a WCET estimation varying the *WCET Computation Mode* and the amount of cache assigned to each of them. Bankization is used as a cache partitioning technique. In Figure 10 we show the normalized WCET average between all the benchmarks belonging to the same *high*, *medium* and *low* demanding group.

Notice that in all cases, the WCET estimation increment when varying the *WCET Computation Mode* from 1 to 4 is almost the same regardless of the cache size given to the task. This is because the bus is accessed before the cache access, so the introduced delay is independent of the cache configuration used.

High demanding benchmarks, shown in Figure 10(a), are very sensitive to both *WCET Computation Mode* variation and cache partition size reduction. Varying the *WCET Computation Mode* from 1 to 4 and fixing a cache size, the WCET estimation increases from 5% to 27%. When reducing the cache size from 64KB to 8KB the WCET estimation increases with respect to using the whole cache (128KB) from 12% to 62%. Hence, running with 4-*WCET Computation Mode* and 8KB cache size, the WCET estimation increases 89% in comparison of not using this mode and having the whole cache. Such increment comes from the contribution of *WCET Computation Mode* (27%) and the cache size (62%).

Medium demanding benchmarks, shown in Figure 10(b), have a smoother behavior with respect to *high* demanding ones. The *WCET Computation Mode* increases the WCET estimation from 3% to 18% when varying it from 1 to 4 respectively. The cache size increases the WCET estimation from 13% to 34% when reducing it from 32KB to 8KB respectively (notice that there is no degradation in performance when reducing the cache size from 128KB to 64KB). Hence, when running with 4-*WCET Computation Mode* and 8KB cache size, the WCET estimation increases 52% in comparison of not using *WCET Computation Mode* and having the whole cache.

Low demanding benchmarks, Figure 10(c), increases the WCET estimation 2% when running with up to 4 *WCET Computation Mode*, and it has not effect on WCET estimation when reducing the cache size.

The collision avoidance algorithm provided by Honeywell, Figure 10(d), resembles a *medium* demanding benchmark. The *WCET Computation Mode* increases the WCET estimation up to 14% when running in 4 *WCET Computation Mode*, and the cache size reduction increases the WCET estimation up to 10% when having a 8KB cache. This results in an overall WCET estimation increment of 24% in comparison of not using this mode and having the entire cache.

Unlike single-core scheduling techniques where only one WCET estimation value per cache size is required, a set of WCET estimations are computed using our *WCET Computation Mode* technique, resulting in the *WCET-matrix* presented in Section 4. As explained, this matrix is given to the scheduling algorithm that looks the best scheduling. The values presented in Figure 10 results in a WCET-matrix of 25 entries, five cache sizes and four *WCET Computation Mode*. Moreover, since each WCET estimation is independent of the workload, any change in a WCET-matrix of a task does not affect any matrix of other tasks.

6.2 Bankization vs Columnization

The previous subsection uses bankization as cache partition technique to evaluate the WCET increment when using our *WCET Computation Mode*. In this section we compare bankization and columnization in terms of WCET estimation variation and implementation complexity.

Figure 11 compares bankization (labeled with *B*-) and columnization (labeled with *C*-) in terms of WCET estimation increment for *high*, *medium* and *low* demanding tasks and the collision avoidance algorithm, when varying the

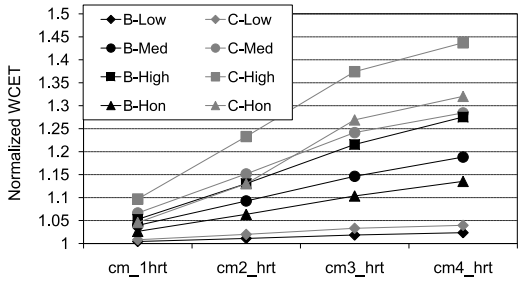


Figure 11: Bankization vs Columnization

WCET Computation Mode from 1 to 4. All the values (in case of bankization and columnization) are normalized to the same WCET estimation obtained running the HRTs with the whole cache (i.e., there is no difference between bankization and columnization) and *WCET estimation mode* disabled.

Columnization prevents only storage conflicts since different threads can still access to the same bank. As a result, the UBD to use with columnization is the one shown in Formula (2). Meanwhile, with bankization we prevent both storage and bank access conflicts, so the UBD to use is given by Formula (1). Hence, bankization provides tighter WCET estimation than with columnization. For *High* demanding tasks (squares in Figure 11) the use of columnization increases their WCET estimation from 4% to 16% when varying the *WCET Computation Mode* from 1 to 4 respectively. In case of *medium* demanding tasks (circles) such increment varies from 2% to 9%. Finally, for *low* demanding tasks (diamonds) the WCET estimation increment is up to 1.6% when running with 4-*WCET Computation Mode*. Collision avoidance algorithm (labeled as Hon), has an increment from 2% to 18% when varying the *WCET Computation Mode* from 1 to 4 respectively.

Even though it is clear that columnization involves a bigger WCET estimation than bankization, bankization has an important drawback: It requires bigger cache area and additional hardware in comparison to columnization. Such increment comes from two sides. First, a BRU (see Figure 8) is required to remap the destination bank. Second, since the number of banks assigned to a given thread is not fixed, the number of bits that form the memory address tag cannot be fixed. It is then required to reserve space for the maximum tag size, that is when only one bank is assigned to a thread. In our architecture the cache area is increased by a 3% in comparison to columnization.

Depending on the allowed WCET estimation increment and the amount the hardware available a designer of an embedded, hard real-time system can choose one of the two alternatives discussed in this section

6.3 Mixed-Application Workload Evaluation

The first objective of our architecture is to ensure that HRTs finish before their deadlines, which can produce a performance degradation of NHRTs. For example, high resource demanding HRTs require reserving a significant part of the cache. As a consequence, when NHRTs are co-scheduled with high-demanding HRTs, NHRTs will be allowed to use less resources than when they run with low demanding HRTs. In this section we analyze the performance, IPC throughput, we obtain for the NHRTs when they run with other HRTs.

We composed 4-thread workloads with 2 HRTs and 2

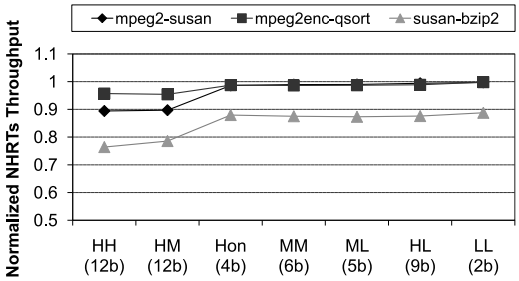


Figure 12: Normalized Throughput of NHRTs. The numbers in parenthesis are the number of L2 banks required by the HRTs

NHRTs. We use HRTs with different resource demands: *high*, *medium* and *low* demanding groups (labeled as H, M and L respectively). As NHRTs we use benchmarks from MediaBench, MiBench and SPEC CPU 2006 grouping them as: *mpeg2dec - susan*, *mpeg2enc - qsort* and *susan - bzip2*.

Figure 12 shows the throughput of the NHRTs when they run with other HRTs. The throughput is normalized to the case when the NHRTs run with no other HRTs at the same time. In these experiments we have assumed that each HRT has an utilization of 20%, that is that its deadline is only 20% higher than its WCET estimation when it runs in isolation ($d_i/WCET_i = 1.2$). In order to accomplish with this time requirement in the x-axis of Figure 12 we show the number of L2 banks that the HRTs must reserve. The number of banks given to the NHRTs is 16 minus the number of banks given to the HRTs, where 16 are the number of banks L2 has in our baseline architecture. In all our experiments, the HRTs finished before their deadlines. In all workloads we observe that, obviously, when the HRTs require less L2 banks to reach their deadline, Low demanding type of HRTs, the NHRTs run faster. In the case of *mpeg2dec-susan* and *mpeg2enc-qsort* benchmarks, they reach an increment of 10% and 5% respectively between LL and HH HRTs, obtaining a throughput of 1 in case of LL (i.e., the maximum we can achieve, because it means we are achieving the same performance of when there are not HRTs in the workload). These results show that our multicore can execute HRTs meeting deadlines and provide high performance to NHRTs. In the case of *susan-bzip2* the variation in the throughput is also 10% but in case of LL the normalized throughput is 0.88, i.e., they do not reach 1. This is because the resources used by HRTs slow down the NHRTs but still they achieve high performance. In general, the performance that NHRTs achieve depend on their cache utilization. If they are high-demanding they will be more affected by the use of cache the HRTs do and vice versa.

Unlike the cache, the bus is not reserved for the HRTs. The bus arbiter just prioritizes the requests from the HRT over the requests of the NHRTs. When the bus is not used by HRTs, NHRTs can use it. The use of the bus done by the HRTs, depends on the particular HRTs.

7. INCREASING SCHEDULABILITY

The *WCET Computation Mode* allows analyzing each HRT in isolation, i.e., independently from the particular task set in which that task is going to be scheduled. Let's define $WCET_k^{taski}$ the WCET estimation for $task_i$ when it runs in *WCET Computation Mode* k. So far we have assumed that all HRTs have the same priority. That is, in the bus schedul-

Table 1: WCET estimation for some EEMBC benchmarks

task	resource demand	WCET isolation (processor cycles)	Normalized WCET in WCET Computation Mode n			
			$WCET_1$	$WCET_2$	$WCET_3$	$WCET_4$
aifftr01	High	9.09×10^8	1.06	1.15	1.26	1.33
a2time01	Low	6.66×10^8	1.001	1.001	1.002	1.003
tblock01	Low	6.40×10^8	1.001	1.001	1.002	1.004

ing we use a round robin policy among HRTs, and they have priority over NHRTs. In this scenario, if we schedule several NHRTs and M HRTs at the same time, the execution time of each HRT is upper bounded by a WCET estimation, $WCET_M^{task_i}$. However, as shown in Figure 10, the WCET of tasks with high resource demands is sensitive to the *WCET Computation Mode* in which the task runs. That is, the WCET estimation rapidly increases as we increase the *WCET Computation Mode* in which we run the task. Meanwhile, tasks with low shared resource demands are almost insensitive to the *WCET Computation Mode* used (less than 2% in the worst-case). To maximize the utilization of the processor, we propose to divide the HRTs into groups. The bus arbiter applies a round robin priority among groups. Inside each group the arbiter also applies a round robin policy. In general, we create g groups of HRTs and in a given group we place n tasks. In this scenario, each task in such group use a WCET estimation $WCET_{g \cdot n}^{task_i}$. That is, a thread in a group with n tasks, has to use the *WCET Computation Mode* k, where k equals the total number of groups times the number of HRTs in its group. Hence, tasks in populated groups use a higher *WCET Computation Mode* than tasks in smaller groups.

Let's assume we want to schedule several NHRTs with the following EEMBC HRTs: *aifftr01*, *a2time01* and *tblock01*. As shown in Table 1, *aifftr01* is a high demanding benchmark, e.g. its WCET is 33% higher when run in *WCET Computation Mode* 4 with respect to its WCET in isolation. Meanwhile, *a2time01* and *tblock01* are two benchmarks with low resource demands.

If we use a round robin policy, we have to use $WCET_3$ for each task, which is high for *aifft01*, 26%. However, if we put *aifftr01* alone in a group and *a2time01* and *tblock01* in another group, each request from *aifftr01* may suffer at most a delay accessing the bus and cache equal to the delay when running in *WCET Computation Mode* 2, which leads to a $WCET_2^{aifftr01} = 1.15(15\%)$, meanwhile *a2time01* and *tblock01* has to run in *WCET Computation Mode* 4, which leads to an increment of their WCET less than 0.5% ($WCET_4^{a2time01} = 1.003$ and $WCET_4^{a2time01} = 1.004$). In this way, the scheduling algorithm can take full advantage of using grouping by considering the resource demands of each HRT, which can be determined analyzing the WCET-matrix of each HRT. Tasks with high resource demands can reduce its WCET estimation by placing them into different small groups, while putting all threads with low demand of shared resources into a single group.

The grouping technique requires small hardware modification to XCBA: Instead of applying a round robin policy between the HRTs, the round robin is applied among different groups and among threads inside each group. Thus, XCBA requires additional information: The total number of groups and the group associated to each thread. For sake of space we do not provide such hardware description.

The scheduler has also to take into account the cache requirements of each task, ensuring that all the tasks that run at the time in the system have enough space to meet their deadline. Notice that for each HRT we have a WCET-matrix that has as many entries as the number of *WCET Computation Mode*s times the number of cache configurations. The computation of the WCET-matrix is done in *isolation* for each benchmark and this process is independent of the particular task set each task is going to run in. When doing the analyzability test, we have to ensure that each task is run in a *WCET Computation Mode* and with enough cache space to meet its deadline. Work on the analyzability test of a system implementing our *WCET Computation Mode* is part of our future work.

8. RELATED WORK

Some works already deal with the problem of analyzability in the presence of some shared resources[3, 14, 9, 17]. In [17] Rosen et al. described a solution to implement predictable real-time applications on multiprocessors. They propose a bus scheduling policy based on TDMA (Time Division Multiple Access) based on a previously statically defined scheduling policy. Different time-slots to access the bus are allocated to different processors by static scheduling, i.e., stored in a memory directly connected to the bus arbiter. This technique needs to know the workload a priori, which is the whole set of tasks that run on the system at any given time, in order to avoid situations where the bus contention increases the memory access latency. This solution prevents any deadline miss due to bus conflicts. The architecture, which is described in [12] for a real-time biomedical monitoring and analysis system, is a multicore processor where each core has its own private memory, connecting all of them with a bus.

The Real-Time Virtual Multiprocessor (RVMP) architecture [9] virtualizes a single in-order super-scalar processor into multiple interference-free different-sized virtual processors. The configuration of the virtual processors can be changed at run-time according to the timing requirements, providing a timing analyzable architecture together with the flexibility of SMT processors. The processor partitioning is determined statically by the real-time scheduling framework that preserves the possibility of analyzing the WCET as in single-cores. The architecture does not include any cache to reduce the level of non-determinism. In this work, the SMT is assumed to have fully-pipelined functional units so that every clock cycle a new access to a shared resource can be performed without any interference.

In [10] the authors propose a real-time multithreading framework, that can be applied on a switch-on-event single-core multithreaded processor. According to the same authors ([9]), the solution is limited to scalar pipelines with only one of the hardware threads selected for execution on the pipeline at the time. The main characteristic of this processor is that a thread cannot overlap its computation and its access to memory. Instead the memory access of one thread can be overlapped with the computation of other threads. This model cannot be applied in our multicore approach, in

which multiple threads run in parallel. In [9, 17] it is further assumed that each task has a private piece of memory on chip, either a cache or a scratchpad. In this paper, instead of an interference-free architecture we focus on an architecture in which threads can compete for the hardware shared resources, providing in this way high performance. Moreover, for the HRTs we provide a WCET estimation that is independent on the task set in which that HRT is executed.

In [14, 6] the authors describe different aspects of accessing shared resources taking into account cache memories. In [14] they deal with interferences at the bus level between cache accesses and I/O peripheral transactions, concluding that these kinds of inferences cause unpredictable behaviors. They present a theoretical framework able to model the interaction between the CPU and the peripherals accessing the front side bus. In [6] they address the cache partitioning problem (to avoid interference between different cores) as an optimization problem. The solution found by the optimization algorithm identifies the optimal size of each cache partition such that the system worst-case utilization is minimized and real-time schedulability is increased.

9. CONCLUSIONS

In this paper we have proposed a new multicore architecture in which the maximum time that a request from a HRT accessing a shared resource can be delayed by any other task is bounded. That is, our multicore processor enforces that a request of a HRT cannot be delayed longer than a given *Upper Bound Delay* (UBD). This is a necessary feature to make a multicore architecture analyzable.

We extend our multicore architecture, which allows determining an *UBD*, with a novel hardware feature called *WCET Computation Mode* that allows estimating safe WCET of HRTs running into our multicore architecture. HRTs are run in isolation with the *WCET Computation Mode* enabled. In this execution mode, the processor artificially delays each HRT request by the worst-case delay that every HRT request can suffer due to the interaction with other tasks when run inside a workload. As a result, the computed WCET estimation is a safe upper bound of the execution of the HRT when it runs in *Standard Execution Mode* together with other tasks in the multicore processor.

Hence, with our solution the WCET analysis of each hard real-time thread can be performed in isolation as done in single-core processors. This is a vital characteristic that future embedded systems should have. Moreover, our proposal can use current WCET analysis tools without requiring any modification, so whatever analysis tool is used in single-core systems can be applied to our multicore architecture.

We evaluate our proposal using a real WCET analysis tool and a real hard real-time application. In particular we evaluate a 4-core architecture with a shared L2 cache connected by a shared bus. On average, for all EEMBC and the application provided by Honeywell, the *WCET Computation Mode* using bankization increases the WCET estimation of HRTs between 2% and 27% respect to the WCET estimated running it alone. Moreover our multicore architecture is able to execute several HRTs and NHRTs at the same time ensuring that HRTs meet their deadlines and providing high performance to NHRTs. We show that the performance of NHRTs ranges between 78% to 99% with respect to when they run in the multicore processor without HRTs.

10. ACKNOWLEDGMENTS

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN-2007-60625, by the HiPEAC European Network of Excellence and by the MERASA STREP-FP7 European Project under the grant agreement number 216415. Marco Paolieri is supported by the Catalan Ministry for Innovation, Universities and Enterprise of the Catalan Government and European Social Funds. Authors would also like to thank Professor Gurindar S. Sohi, Rob Davis and the conference reviewers for their helpful comments. We also thank Honeywell for providing us the Collision Avoidance application, Jörg Mische, Professor Theo Ungerer at University of Augsburg for their help in the integration of CarCore emulator into our simulation environment and the members of Industrial Advisory Board of MERASA[1] for their precious advices.

11. REFERENCES

[1] *MERASA EU-FP7 Project: www.merasa.org*, 2007.

[2] *RapiTime: Worst-case execution time analysis. User Guide. Rapita Systems. Ltd.*, 2007.

[3] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *CODES+ISSS*, Salzburg, Austria, 2007.

[4] A. Andrei, P. Eles, Z. Peng, and J. Rosen. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In *VLSID*, Hyderabad, India, 2008.

[5] G. Bernat, A. Colin, and S. Petters. WCET analysis of probabilistic hard real-time systems. In *RTSS*, USA, 2002.

[6] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. *RTCSA*, Kaohsiung, Taiwan, 2008.

[7] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *DAC*, Los Angeles, CA, USA, 2000.

[8] K. De Bosschere, W. Luk, X. Martorell, N. Navarro, M. O'boyle, D. Pnevmatikatos, A. Ramirez, P. Sainrat, A. Seznec, P. Stenstrom, and O. Temam. High-performance embedded architecture and compilation roadmap. 2007.

[9] A. El-Haj-Mahmoud, A. S. AL-Zawawi, A. Anantaraman, and E. Rotenberg. Virtual multiprocessor: an analyzable, high-performance architecture for real-time computing. In *CASES*, San Francisco, CA, USA, 2005.

[10] A. El-Haj-Mahmoud and E. Rotenberg. Safely exploiting multithreaded processors to tolerate memory latency in real-time systems. In *CASES*, Washington DC, USA, 2004.

[11] Infineon. *Tricore 1. 32-bit Unified Processor Core v1.3*, 2005.

[12] I. A. Khatib, F. Poletti, D. Bertozzi, L. Benini, M. Bechara, H. Khalifeh, A. Jantsch, and R. Nabiev. A multiprocessor system-on-chip for real-time biomedical monitoring and analysis: architectural design space exploration. In *DAC*, San Francisco, CA, 2006.

[13] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *RTSS*, Phoenix, AZ, USA, 1999.

[14] R. Pellizzoni and M. Caccamo. Toward the predictable integration of real-time COTS based systems. In *RTSS*, Tucson, Arizona, USA, 2007.

[15] J. Poovey. *Characterization of the EEMBC Benchmark Suite.* North Carolina State University, 2007.

[16] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a qualitative and quantitative comparison. Technical report, IRISA, Paris, France, 2006.

[17] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*, Tucson, Arizona, USA, 2007.

[18] Tasking. *Tricore v2.2 C Compiler, Assembler, Linker Reference Manual*, 2005.

[19] L. Thiele and R. Wilhelm. Design for time-predictability. In *Design of Systems with Predictable Behaviour*, 2004.

[20] S. Uhrig, S. Maier, and T. Ungerer. Toward a processor core for real-time capable autonomic systems. In *Proc. ISSPIT*, Athens, Greece, 2005.