A Software-Pipelined Approach to Multicore Execution of Timing Predictable Multi-Threaded Hard Real-Time Tasks

Marco Paolieri[1,3], Eduardo Quiñones[1], Francisco J. Cazorla[1,2],
Julian Wolf[4], Theo Ungerer[4], Sascha Uhrig[5], Zlatko Petrov[6]
[1] Barcelona Supercomputing Center (BSC), Spain
[2] Spanish National Research Council (IIIA-CSIC), Spain
[3] Universitat Politecnica de Catalunya, Spain
[4] University of Augsburg, Germany
[5] Technical University Dortmund, Germany
[6] Honeywell International s.r.o., Czech Republic

***Abstract*—Multicore processors can deliver higher performance than single-core processors by exploiting *thread level parallelism* (TLP): applications are split into independent threads, each of which is mapped into a different core, reducing the execution time and potentially its worst-case execution time (WCET). Unfortunately, inter-thread interferences generated by simultaneous accesses to shared resources from different threads may completely destroy the performance benefits brought by TLP.**

This paper proposes a software/hardware *cache partitioning* approach that reduces the inter-thread memory interferences generated in hard real-time software-pipelined parallel applications. Our results show that our approach effectively reduces memory interferences, while still guaranteeing a predictable timing behaviour, achieving a WCET estimation reduction of 28% for a software pipelined version of the LU decomposition application with respect to the single-threaded version.

Keywords-hard real-time systems, multicore processors, cache partition, software pipelined parallel programming model

I. Introduction

Multicore processors are increasingly being considered as an effective solution to cope with the performance requirements of current and future hard real-time embedded systems [2]. Multicore processors ideally enable co-hosting applications with different criticality levels, so higher number of tasks can be scheduled, maximizing the hardware utilization, while reducing cost, size, weight and power requirements. Moreover, they offer a better performance per watt ratio than single-core processors, while maintaining a relatively simple core design.

Unfortunately, multicore processors can increase the execution time of single-threaded applications, and hence the worst-case execution time (WCET), due to inter-thread interferences accessing shared hardware resources. Inter-thread interferences appear when two or more threads try to access a shared resource at the same time, so an arbitration mechanism is required, affecting the execution time of the running threads. To handle this type of contention, WCET analysis techniques have to consider the impact of such contention on the execution time of applications, potentially introducing an increment on the WCET estimations with respect to their corresponding WCET estimations when running in stand-alone mode, i.e. without suffering inter-thread interferences [8].

High performance can be achieved using multicore architectures if applications are made *multi-threaded* by exploiting thread-level parallelism (TLP): applications are split into threads that run in parallel on different cores and synchronize whenever they need to communicate. This paper focuses on the software-pipelined parallel programming technique, i.e. a particular implementation of the producer-consumer programming model. In the software-pipelined approach, a stream of data is passed through a sequence of pipeline stages and each stage performs a step of the overall computation. These stages can be implemented as individual threads. Since the different stages that compose the application access different portions of data, the stage threads can be executed in parallel on different cores of a multicore processor.

However, each thread may also suffer from inter-thread interferences like accesses to shared resources, which can potentially reduce the performance benefits brought by TLP. The shared resource with the highest impact on the WCET is the main memory [9]. Therefore, the inter-thread interferences generated by simultaneously executed threads when accessing the memory may destroy the advantage of TLP. That is also the case of the software-pipelined approach. In fact, the data that has been processed by stage n is stored in memory, so it can be accessed by the next stage $n+1$. Concurrently, stage n starts to process a new portion of data that has been stored also in memory by stage $n-1$, originating memory contention between the two stages.

Shared caches have traditionally been used to alleviate the impact of accessing the main memory in multicore processors. However, their use in hard real-time scenarios complicates considerably the WCET analysis or make it even infeasible, as shared caches generate storage interferences: a thread evicts data of another one, originating additional misses and it potentially delays the execution time of the second thread, and so the benefits of caches are lost. Partitioned caches [8] have been proposed to increase the predictability of shared caches in multicore processors by assigning private portions of the cache to each thread and so eliminating the storage interferences. However, cache partitioning techniques are not suitable for the software-pipelined pattern, due to the fact that moving the data among the different stages of the application,

would require to copy such data among the different cache partitions in order to have the proper data stored in the correct cache partition.

This paper proposes a software/hardware *cache partitioning* approach that exploits the benefits of TLP by reducing the memory contention in hard real-time pipelined parallel applications guaranteeing a predictable timing behaviour. The hardware technique, called *bankization* [8], which allows to dynamically assign private cache banks to cores at run-time, has been extended with a software interface which ensures a correct behaviour of the cache partition remapping mechanism and provides the foundation for developing multi-threaded applications using the bankization technique. By doing so, the data generated by one stage and stored into its corresponding cache partition can be easily accessed by another stage executed in another core, without requiring the copy/movement of the entire chunk of data among the different cache partitions.

Our results show that inter-thread memory interferences increase the WCET estimation of the pipelined parallel version of a LU decomposition application by 30% with respect to the single-threaded version. Instead, when using the bankization technique, interferences are drastically reduced, achieving a WCET estimation reduction of 28% with respect to the single-threaded version. Therefore, our technique effectively reduces the WCET estimation of pipelined parallel applications when comparing to the single-threaded version, bringing back the benefits of parallel execution.

II. Background

A. MERASA Multicore Architecture

This paper focuses on the predictable MERASA multicore processor described in [14]. Concretely, we consider a two-core processor in which each core has a private dynamic instruction scratchpad (D-ISP) [7], which loads the complete code of an activated function dynamically on call and return, and a private data scratchpad (DSP), which holds the private stack. The cores are connected to a *dynamically shared partitioned cache*, which holds the heap of the application, and a JEDEC-compliant DDR2 SDRAM memory system through a shared bus. The partitioned cache is loaded from the memory system. Figure 1 shows the complete architecture.

The MERASA architecture ensures, by design, that the maximum delay a request to a shared hardware resource can suffer due to inter-thread interferences[1] is bounded by a pre-computed *Upper Bound Delay* (UBD) and such UBD depends on the number of co-running Hard Real-Time Tasks (HRTs). In other words, the architecture ensures that no request will be delayed longer than UBD cycles, regardless of the behaviour of the other HRTs running simultaneously on different cores [8], [9]. In the MERASA architecture, two sources of inter-thread interferences can increase the WCET estimation of HRTs: the *shared bus* and the *main memory*.

In order to make the WCET analysis of a HRT independent from the other co-running HRTs, the arbitration policy of

[1]Inter-thread interferences appear when two or more requests from different cores attempt to access a shared hardware resource at the same time

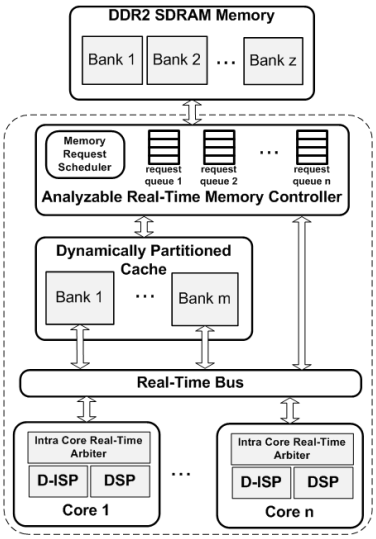

Fig. 1. MERASA Multicore Architecture

both resources, i.e. the bus arbiter and the memory controller arbiter, implements a *round robin* policy with a private request queue per core (see Figure 1). Hence, for each shared resource, the UBD a request can suffer from because of inter-thread interferences is determined by the worst-case access latency and the number of simultaneously executed HRTs (No_{HRT}), i.e. the maximum number of round robin slots. According to this, the UBD can be defined as a function of No_{HRT}, i.e. $UBD(No_{HRT})$. For example, in case of the bus, $UBD_{bus} = (No_{HRT} - 1) \cdot L_{bus}$, where L_{bus} is the bus transmission latency: In the worst case scenario, a request must wait until all other HRTs have been served. Similarly, in case of the memory controller, the $UBD_{MC} = (No_{HRT} - 1) \cdot t_{IDWORST}$, $t_{IDWORST}$ being the longest possible delay that a memory request can take because of another memory request that was previously issued by the memory controller; such delay depends on the specific timing constraints determined by the DDR2-SDRAM memory device used [4].

Then, in order to consider the $UBD(No_{HRT})$ in the computation of the WCET, every request that accesses the bus and the memory controller is delayed by UBD_{bus} and UBD_{MC} cycles respectively, so the HRT considers the worst-case delay of interferences. Hence, the WCET estimation obtained considering $UBD(No_{HRT})$ is the worst-case scenario that could occur [8], [5]. It is important to remark that different $UBD(No_{HRT})$ (varying the No_{HRT}) lead to different scenarios, and so different WCET estimations, making them independent of the specific tasks inside the task set.

Finally, the cache shared among the different cores may also suffer from two different types of inter-thread interferences: *bank* and *storage interferences*. Instead of bounding the effect of such interferences, the MERASA multicore processor completely avoids them by introducing *bankization* [8], a *Dynamically Partitioned Cache* that assigns a private subset of the cache banks to each core so that no other core can access them. The assignment of banks to cores is controlled by the system software. Section IV-B explains bankization in more detail.

B. MERASA System-Level Software

The MERASA system-level software [18] contains functionalities of a real-time operating system (RTOS) that provides the foundation for applications running on the MERASA processor. The system software guarantees an isolation of multiple HRTs on memory and I/O resources. When isolation cannot be achieved, the system software ensures a time-bounded access to avoid mutual and possibly unpredictable interferences. The intention of this isolation and bounding is also to enable an effective WCET analysis of the application's code. The resulting system software allows to execute multiple simultaneous HRTs on different cores.

The memory management of the MERASA system-level software minimizes interferences of different threads by providing a flexible two-layered memory management. The first layer allows pre-allocation of memory regions to threads while the second layer is in charge of memory management inside each thread. The pre-allocation of the heap regions is used to guarantee the isolation of different threads' memory regions [6]. After the pre-allocation of memory regions to the threads, the dynamically partitioned cache is configured accordingly.

III. Our Proposal

In this section we propose a new technique to propagate data through a software-pipeline using a multicore processor with dynamic cache partitioning. Besides a performance increase compared to a single-threaded version of the same application, the pipelined version is still suitable for hard real-time systems.

A. The Software-Pipelined Parallel Programming Model

The software-pipelined parallel programming technique, i.e. a particular implementation of the producer-consumer programming model, is a very popular programming model in which a stream of data is passed through a sequence of pipeline stages, where each of them performs a part of the overall computation. With the exception of the first stage of the pipeline, which is fed with the original input data, the execution of each following stage requires the output of the preceding one. Hence, the data moves through the pipeline in discrete clocking steps. The length of such a clocking step and herewith the performance of the pipeline, is defined by the longest pipeline stage i.e., by the pipeline stage with the highest WCET in case of a hard real-time system. Improving the execution speed of the longest pipeline stage also improves the overall performance of the pipeline while improving any other pipeline stage is useless.

As an example, we consider the two-staged pipeline shown in Figure 2. In the case of a single-core, the processor has to execute the two stages sequentially and also multiple times to process a stream of incoming data portions. On a multicore system, the two pipeline stages can be executed simultaneously working on different iterations of the incoming data stream. This means that the second iteration of the pipeline stage 0 starts processing the second data portion of the input stream while, in parallel, stage 1 performs the second processing part of the first data portion.

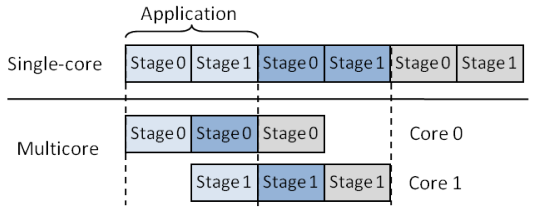

Fig. 2. The original application is split into two stages. Each iteration of the application is represented in a different color (light-blue, blue and grey). In a single-core processor, each stage runs one after the other. Instead, in a multi-core processor, each stage can run in a different core.

By doing so, it would be reasonable to expect that, without considering the first and the last stages of the execution, the execution time of the application can be reduced by a factor of two. However, because of inter-thread interferences accessing the main memory, the performance benefits of parallel pipelining can be significantly reduced. That is, the data processed by stage 0 is stored in main memory, so stage 1 can access it. At the same time, stage 0 starts to process a new stream of data also stored in main memory, generating memory contention between the two stages.

Traditionally, caches have been used to alleviate the impact of memory interferences. If we consider the example above, the data processed by the stage 0 in core 0 should be available to the stage 1 in core 1 without requiring to access the main memory, resulting in a reduced number of accesses to main memory and hence memory interferences. However, concurrently to the execution of stage 1, stage 0 is also executed for the next iteration, generating *storage interferences* among the two stages.

These storage interferences lead to an unpredictable cache state at any point in time of the execution. Hence, a tight WCET estimation is infeasible as the cache must be assumed to produce cache misses only. As a result, the performance of the hard real-time software-pipeline is very low in case of using a shared cache because it directly depends on the WCET estimation.

B. Bankization: A Dynamic Partitioned Cache

Cache partitioning have been proposed to completely eliminate storage interferences by splitting the cache into private portions, each assigned to a different thread. Unfortunately, this makes each partition private to each thread and so not visible to the rest of the threads. Thus, if we consider the example of Figure 2, and we assign a different cache partition to each stage, the storage interferences would be eliminated, but the data generated by stage 0 would be invisible to stage 1. A naive solution is to copy the data from the cache partition assigned to stage 0 to the cache partition assigned to stage 1. However, the overhead introduced could reduce the performance benefits of the pipelining.

The dynamically cache partitioning technique called *bankization* that is integrated in the MERASA architecture (see Figure 1) allows to assign a subset of the total number of cache banks to each thread so that no other thread can access it (see [8] for more details).

To do so, bankization introduces the *Bank Remapping Unit* (BRU) that determines the bank assigned to a given thread

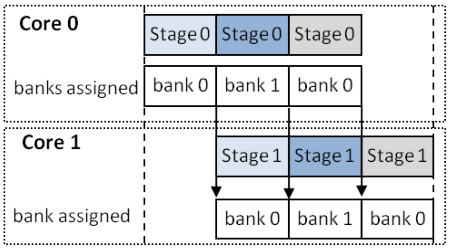

Fig. 3. A two-stage pipeline application running in a two-core processor. Bankization allows to reassign the bank used by stage 0 to stage 1.

based on the thread identifier and the accessed memory address (the destination bank of any memory request is contained inside its address). By changing the bank assignment defined inside the BRU, bankization allows to remap the banks assigned to the different threads at runtime such that the data visible to one thread can be accessed by another one without requiring the movement of data among banks.

Let's consider the pipelined example shown in Figure 3. The first stage of the pipeline, stage 0 (light-blue box) is assigned to an empty bank: bank 0, such that no other thread can access it. Then, once stage 0 has finished, bank 0 is re-assigned to stage 1 (light-blue box), such that it can access the data stored by stage 0. At the same time, a new empty bank (bank 1) is assigned to a new execution of stage 0 (blue box) such that it can compute a new stream of data without interfering with bank 0. Once stage 1 (light-blue box) has finished its computation, bank 0 can be re-assigned to a new execution of stage 0 (grey box) as an empty bank, and so all the content will be invalidated.

C. System Software Bankization Interface

In parallel programming models, such as the software pipelined model considered in this paper, multiple threads of the application need access to a shared heap memory region for data exchange. Using the pure bankization technique would not allow to access shared data because memory regions are visible only to a single thread.

However, in order to guarantee the correct functional behaviour of bankization as well as to provide time predictability we introduced two types of heap regions: *private heap* and *shared heap*. Both types of regions only allow accesses of one thread at a time (during one clocking step) but the shared regions can be exchanged between pipeline stages. To reach that, it is required to: (1) isolate the shared heap regions from the private heap regions such that accesses to the shared heap do not collide with private heap accesses inside the same bank, and (2) synchronize the accesses to the shared data. We enhanced the MERASA system-level software to address both requirements.

First, we provide a mechanism to split the heap memory region of the application into two independent regions: the *private heap region* and the *shared heap region*. The private region uses a real-time capable two-level allocation mechanism [6] to maximize the isolation of different threads' memory regions. Instead, in the shared heap region, which allows the access of multiple threads, the two-level allocation is reduced to one real-time capable allocation level. It is important to remark that the shared allocation during the application's execution must be performed in a mutually exclusive way to keep the state of memory consistent. Nevertheless, we are able to guarantee timing predictability, as the number of potentially waiting threads, i.e. the list of HRTs is bounded by the number of cores in our architecture.

Moreover, in order to avoid storage interferences among different accesses to the private and shared heap regions, we extended the BRU such that both regions are mapped into different banks. To do so, it is required to consider only few most significant bits of the address defined by the linker, that distinguish the two regions. The information about the heap region type is set using the *set_privateheap_size* and *set_sharedheap_size* functions that control the BRU based on the given cache partition size of each region.

Second, we guarantee that a bank used by one thread is not remapped until this thread has finished. However, such a synchronization is not done at a single stage basis but at the application basis. In other words, the bank remapping has to be done once all stages that run simultaneously have finished. Such a constraint is fundamental to allow the computation of the WCET estimation, as the WCET of each parallel phase can be expressed as the maximum WCET of all different stages. Section IV analyses in detail the WCET estimation of a pipelined parallel application. We have integrated the following functionalities into the MERASA system-software, which will ensure the correct behaviour of the cache by synchronizing the bank remapping at the end of all threads:

- *init_pipeline*: This function sets the number of stages and it assigns to the first stage/thread of the pipelined parallel application its corresponding set of banks based on the cache size given to each stage. The size is forced to be a multiple of the size of a cache bank. Moreover, it initializes two barriers for synchronization which are used in the *clock_edge* function. This functions uses the *set_sharedheap_size* to identify the set of banks as shared heap region.
- *start_pipeline*: It stalls each thread until the data coming from the previous thread is available. This function is composed of a loop controlled by the number of stages that have not being executed yet for the first time. Each iteration, the *clock_edge* function is called.
- *stop_pipeline*: This function does the same as start_pipeline but at the end of the execution. Thus, it stalls each thread until all the subsequent threads finish.
- *clock_edge*: It is the core of the software-pipelining technique and ensures the correct setting of the bankization parameters and the synchronous start of all pipeline stages (threads). Two barriers are required for this purpose. The first barrier ensures that all stages have finished their computation. When all stages have reached that barrier, the remapping of the banks is performed. After that, the second barrier ensures that the new bank assignment has finished and all threads start execution of the next iteration. The first pipeline stage is the one in charge of generating the banks' reassignment.

Moreover, the MERASA system software is equipped with

```
1  void main() {
2    init_pipeline(n, 4096);
3    stage 0 in core 0;
4    stage 1 in core 1;
5      ...
6    stage n in core n;
7  }
8  stage n {
9    start_pipeline(n);
10   while (true) {
11     do_stage n;
12     clock_edge(n);
13   }
14   stop_pipeline(n);
15 }
```

Fig. 4. An n-stage pipelined parallel application using the bankization interface

time predictable synchronization primitives, which implement the barriers required for the bankization interface [10].

Figure 4 shows an example of using the software pipelining interface in a n-stage pipelined parallel application. The pipelining mechanism is initialized using *init_pipeline* (line 2). At the beginning of each stage, e.g. stage n, the function *start_pipeline* (line 9) stalls it until its previous stage, stage $n-1$, finishes. This function call is also responsible of reassigning the new bank when the computation starts with the data produced by stage $n-1$. Once the stage has performed its computation (line 11), it is stalled again in the *clock_edge* (line 12) waiting all stages to finish in order to perform the remapping. Finally, when the pipeline finishes, *stop_pipeline* (line 14) stalls all stages until the last stage finishes.

IV. WCET Analysis of a Software-Pipelined Parallel Application

A. WCET Estimation Without Cache

In the WCET analysis of a pipelined application it is required to consider the longest execution time of all the stages that compose the application, considering the impact that inter-thread interferences have on the WCET due to the parallel execution of multiple stages. However, the impact that interferences have along the execution is different depending on the execution phase in which they are: the initialization phase (*prologue*), finalization phase (*epilogue*) and the central phase of the execution (*kernel*).

Figure 5 shows a four-stage pipelined application that is executed five times (each color represents a different iteration). Let's assume that $WCET_n$ is the computed WCET estimation of the stage n. However, this WCET is not the same along the execution of the application, because the number of stages that run simultaneously changes in the prologue and epilogue phase. Therefore, it is required to consider the WCET in the different phases. Let's assume that the $WCET_n(m)$ is the WCET of stage n when running with m simultaneous stages. In other words, $WCET_n(m)$ considers the impact that inter-thread interferences have on the WCET estimation stage n introduced by the other stages. Thus, the WCET estimation of each stage can be represented as:

- During the prologue phase, stage 0 runs alone and simultaneously with one and two more stages (stage 1 and 2).

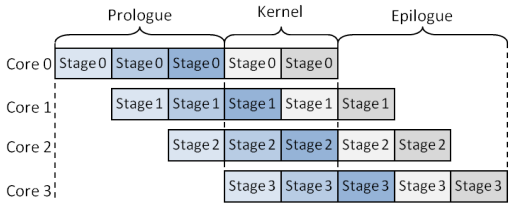

Fig. 5. The prologue, kernel and epilogue phase of a 4-stage pipelined application that runs five times, each represented with a different color (light-blue, blue, dark-blue, light-grey and grey)

Hence, the WCET of the prologue ($WCET_{pro}$) can be expressed as:

$$WCET_{pro} = WCET_0(0) + max\{WCET_0(1), WCET_1(1)\} + max\{WCET_0(2), WCET_1(2), WCET_2(2)\} \quad (1)$$

- During the epilogue phase, stage 3 runs alone and simultaneously with one and two more stages (stage 1 and 2). Hence, the WCET of the epilogue ($WCET_{epi}$) can be expressed as:

$$WCET_{epi} = WCET_3(0) + max\{WCET_2(1), WCET_3(1)\} + max\{WCET_1(2), WCET_2(2), WCET_3(2)\} \quad (2)$$

- Finally, during the kernel phase, all stages run simultaneously. Hence, the WCET of the kernel ($WCET_{kernel}$) can be expressed as:

$$WCET_{kernel} = 2 \times max\{WCET_0(3), WCET_1(3), WCET_2(3), WCET_3(3)\} \quad (3)$$

were 2 is the number of iterations executed during the kernel phase.

Therefore, the overall WCET of a pipelined application can be expressed as:

$$WCET_{pipe} = WCET_{pro} + WCET_{epi} + WCET_{kernel} \quad (4)$$

B. Considering The Cache into the WCET Estimation

However, when introducing the bankization technique in the computation of the WCET of each stage, it is required to consider the state of the cache at the end of the previous stage. That is, because each stage consumes the bank used by the previous one, a WCET analysis chain is created, in which the analysis of each state is used to feed the analysis of the next stage. It is worth noting that the analysis of a given stage does not require to know the state of the cache in all previous stages but just the previous one. Hence, we can define $WCET_n(m, cache_{n-1})$ as the WCET estimation of stage n, being n any stage except stage 0, running with m simultaneous stages and considering the cache state at the end of stage $n-1$. In case of stage 0, the WCET remains as $WCET_0(0)$, because it starts the computation of the streaming with the cache empty.

Therefore, the computation of the WCET estimation in every pipeline phase results in:

- The prologue phase:

$$WCET_{pro\_b} = WCET_0(0) + max\{WCET_0(1), WCET_1(1,0)\} + max\{WCET_0(2), WCET_1(2,0), WCET_2(2,1)\} \quad (5)$$

- The epilogue phase:

$$WCET_{epi\_b} = WCET_3(0,2) + max\{WCET_2(1,1), WCET_3(1,2)\} + max\{WCET_1(2,0), WCET_2(2,1), WCET_3(2,2)\} \quad (6)$$

- The kernel phase:

$$WCET_{kernel\_b} = max\{WCET_0(3), WCET_1(3,0), WCET_2(3,1), WCET_3(3,2)\} \quad (7)$$

Therefore, the overall WCET of a pipelined application when using bankization can be expressed as:

$$WCET_{pipe\_b} = WCET_{pro\_b} + WCET_{epi\_b} + num_{iterations} * WCET_{kernel\_b} \quad (8)$$

It is important to remark that the bankization interface ensures that the end of the stage n and the beginning of stage $n+1$ are synchronized. By doing so, we can ensure that the WCET estimation of multiple stages running in parallel is equal to the highest WCET estimation among the different stages. Note that, in case of not using bankization, such synchronization among the stages is also required.

V. Experimental Setup

The experiments presented in section VI have been obtained using an in-house cycle-accurate, execution-driven simulator compatible with Tricore ISA and derived from CarCore[13] that models the MERASA multicore processors described in Section II-A. Each core implements an in-order dual-issue pipeline with a fetch bandwidth of 64 bits. No branch prediction is used. The Dynamic Instruction Scratchpad [7] and the Data Scratchpad has an latency of 1 cycle.

The simulator also models the dynamically partition cache, considering a cache size of 128KB (16-way, 16-banks, 32-byte per line, 9 cycles access, write-back write-allocate policy, LRU replacement policy). The memory system is modelled using the DRAMsim [15], which has been integrated inside our simulation framework. The memory system considered is a close-page/interleaved-bank memory controller that is interfaced with a JEDEC-compliant DDR2-800C 256Mb x16 DDR2 SDRAM device composed by a single DIMM, single rank and a single 4-banks memory device [4]. We assume a CPU frequency of 800MHz, being a CPU-SDRAM clock ratio of 2.

WCET estimations has been obtained using RapiTime, a commercial timing analysis tool [1]. In order to consider the UBD of both, the shared bus and the memory controller, on the WCET estimation, the simulator introduces the *WCET computation mode* [8]. When computing the WCET estimation of a HRT, the processor is set to that mode and the HRT under analysis is run in isolation. Under this mode, the bus arbiter and the memory controller artificially delay every request by UBD cycles. Once the WCET analysis of all the HRTs is completed the processor is set to *Standard Execution Mode* in which no artificial delay is introduced. Hence, the resulting WCET estimation considers the worst possible inter-thread interference scenario. Finally, the MERASA system software implements the interface described in Section III-C.

A. Benchmarks

We used pipelined versions of two applications for the evaluation: LU decomposition and a stereo navigation application.

1) LU Decomposition: In linear algebra, the LU decomposition (also called LU factorization) of the square matrix A is defined as $A = L \times U$, where L and U are lower and upper triangular matrices of the same size respectively, in which L has only zeros above the diagonal and U has only zeros below the diagonal. This decomposition is used in numerical analysis to solve systems of linear equations or to calculate the determinant. We have split the applications into two stages: Given a $n \times n$ square matrix A, the *stage 0* replaces A by the LU decomposition of a row wise permutation of itself. The resultant matrix A' is arranged using the Crout's method. Based on the stage 0's output, A', *stage 1* solves the set of n linear equations $A'X = B$ computing the solution vector X, being B the right-hand side vector.

2) Stereo Navigation: The stereo navigation application is intended for an aircraft localization in case that the GNSS (Global Navigation Satellite System) used in aircraft is temporarily unavailable and the plane has to localize itself for some period of time. In the latter situation the lack of stereo navigation application responsiveness may have a catastrophic outcome and therefore it is typically considered as a hard real-time application. The application is built on the idea that from two independent images derived from cameras looking in approximately the same direction features can be extracted (dominant entities in the image invariant to rotation and translation). Using two cameras recording images at the same time allows for localization of the features in a 3D-space. Furthermore, from two adjacent image snaps taken in two subsequent time moments t1 and t2 and the change of position of the features in the images, absolute translation $(dT/(t2-t1))$ and absolute rotation $(dR/(t2-t1))$ can be inferred. The stereo navigation computation is composed by the 10 phases (rectify, tile, sort, match_lr, match_t1t2, circular_check, reproj, ransac_loop, refin, finit_state). We have split the application in two stages: the two first phases compose the first stage; the rest composes the second stage.

VI. Results

This section presents the impact of the bankization technique on the WCET estimation of the two-stage pipelined parallel version of the two applications. We consider the WCET estimation of each application under three different scenarios. For each scenario, we define n as the number of

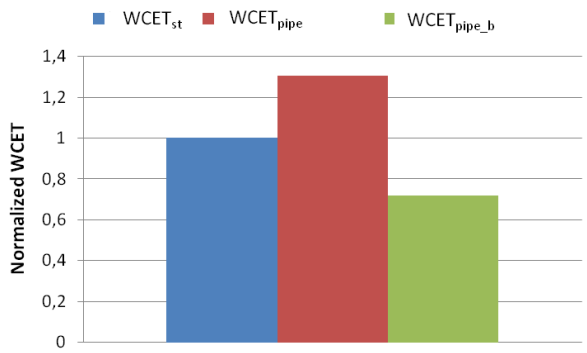

Fig. 6. WCET estimation of the two-stage LU decomposition application using no shared cache ($WCET_{pipe}$) and bankization ($WCET_{pipe\_b}$). Values are normalized to the WCET estimation of the single-threaded version ($WCET_{st}$).

iterations of each application ($n = 5$ for LU Decomposition and $n = 4$ for stereo navigation).

1) *Single-thread version* ($WCET_{st}$): The two stages run one after the other into the same core. We assume that no other threads run concurrently, so the benchmark does not suffer from any inter-thread interferences from other applications. A private cache partition of 32 KB is assigned to this core. The WCET expression used is: $WCET_{st} = n \times [WCET_0(0) + WCET_1(0,0)]$
2) *Pipelined parallel version without cache* ($WCET_{pipe}$, see equation 4): Each stage runs in a separate core. Again, no other threads run in parallel, so each stage may suffer only from inter-thread interferences from the other stage. The cache has been disconnected, so all memory requests go directly to the memory controller through the bus.
3) *Pipelined parallel version with bankization* ($WCET_{pipe\_b}$, see equation 8): Same as above but with cache and bankization technique enabled. So the cache partition used by stage 0 can be consumed by stage 1. A private cache partition of 16 KB has been assigned to each core.

A. LU Decomposition

Figure 6 shows the WCET estimation when running the two-stage LU decomposition application into two different scenarios: Using the bankization technique ($WCET_{pipe\_b}$) and not using a shared cache ($WCET_{pipe}$). Values are normalized to the WCET estimation of the single-threaded version of the LU decomposition application ($WCET_{st}$).

As expected, without a cache ($WCET_{pipe}$) the memory access interferences destroy completely the benefits brought by the pipelined model and increase the WCET estimation by up to 30% with respect to the single-threaded version. Instead, if using the bankization technique, the memory interferences are considerably reduced. As a consequence the WCET estimation is reduced by 28% compared to the single-threaded version, taking full advantage of the parallel execution.

To better understand the benefits of bankization, Table 1 shows the WCET estimation increment of each stage of the LU decomposition (stage 0 and 1) during the three phases of

TABLE I
WCET ESTIMATION INCREMENTS OF THE LU DECOMPOSITION DURING THE THREE PHASES USING NO CACHE ($WCET_{pipe}$) AND BANKIZATION ($WCET_{pipe\_b}$). VALUES ARE NORMALIZED TO THE WCET ESTIMATION OF THE SINGLE-THREADED VERSION USING CACHE.

	$WCET_{pipe}$	$WCET_{pipe\_b}$
stage 0 (*prologue*)	24.06%	0.10%
stage 1 (*epilogue*)	30,07%	0,50%
max{stage 0, stage 1} (*kernel*)	22,73%	-33,43%

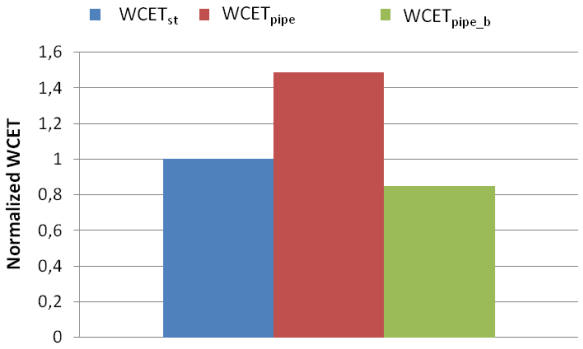

Fig. 7. WCET estimation of the two-stage stereo navigation application using no shared cache ($WCET_{pipe}$) and bankization ($WCET_{pipe\_b}$). Values are normalized to the WCET estimation of the single-threaded version ($WCET_{st}$).

the application (prologue, epilogue and kernel) if the shared cache is not used ($WCET_{pipe}$, see equations 1, 2 and 3) and if cache and bankization is enabled ($WCET_{pipe\_b}$, see equations 5, 6 and 7). Values are normalized to the WCET estimation of the corresponding stage in the single-threaded version: stage 0 for the prologue phase, stage 1 for the epilogue phase and the sum of both stages for kernel phase (note that in this case, stages do not suffer inter-thread interferences). Not using bankization, impacts negatively on the WCET estimation of each stage, increasing the WCET estimation in all phases, including the kernel phase whose WCET estimation is higher that the sum of the WCET estimation of the two stages in the single-threaded version. However, in case of using bankization, although the WCET estimation increases in the prologue and epilogue phase by 0.1% and 0.5% due to having a smaller cache (16KB) with respect to the single-threaded version (32KB). The benefit comes from the kernel phase that reduces the WCET estimation by 33% due to executing both stages in parallel.

B. Stereo Navigation

Figure 7 shows the WCET estimation of the two-stage stereo navigation application in the two different scenarios: Using the bankization technique ($WCET_{pipe\_b}$) and not using a shared cache ($WCET_{pipe}$)). Values are normalized to the WCET estimation of the single-threaded version of the stereo navigation application ($WCET_{st}$).

Similarly to the LU decomposition case study, the interferences introduced by main memory when not using a cache ($WCET_{pipe}$) destroy completely all the benefits brought by the pipelined parallel model and increases the WCET estimation by up to 48% with respect to the single-threaded version. Instead, when using the bankization technique, the

memory interferences are considerably reduced, taking full advantage of the parallel execution and so allowing the WCET estimation to be reduced by 15% compared to the single-threaded version.

VII. Related Work

In the field of embedded real-time systems, there are different related approaches for predictable architectures. Schoeberl provides a time-predictable computer architecture for real-time systems that supports worst-case execution time (WCET) analysis and evaluates it using the Java optimized processor (JOP), which is a real-time Java processor [11]. For performance enhancements, Schoeberl proposes a multicore system that features time-sliced arbitration of the main memory access to enable analyzability.

In [12] Schoeberl and Puschner additionally discuss a single-path programming style combined with multicore systems. The single-path approach includes a conversion, which eliminates all input dependent control flow decisions in order to get straight-line predicated code. In that case, the WCET can be measured in an easy way. On the other hand, this programming paradigm is quite uncommon and restrictive.

To find a trade-off between predictability and processor throughput, Whitham introduced the MCGREP architecture [16], which enhances a sequential processor with dynamically reconfigurable logic. By this, a predictable processor increases performance through the use of application specific microprograms directing the operation of the reconfigurable logic.

Edwards and Lee developed the concept of precision timed (PRET) machines [3] applied to predictable simultaneous thread execution. This approach uses a thread-interleaved pipeline to guarantee a precise timing without a loss of throughput, while providing a simplified WCET analysis.

Predator is another European Commission project reconciling efficiency with predictability [17]. Its architectural approach uses analyzable caches, a compiler-controlled memory management, and simple cores with analyzable behaviour. It uses predictable kernel mechanisms to cover the main challenge for a real-time operating system, which is the variability in task execution times caused by interference.

VIII. Conclusions

This paper proposes a software/hardware cache partitioning approach that exploits the benefits of the software pipelined parallel programming model to effectively reduce inter-thread interferences when accessing the main memory, and so the WCET estimation with respect to the single-threaded programming model. Our approach extends bankization [8], a dynamically cache partitioning technique that allows to assign at run-time a private set of cache banks to a thread that no other can use, by providing an interface to guarantee the correct functional behaviour of bankization as well as to provide the time predictability required by hard real-time applications. The programming model splits the heap memory region into two isolated regions: the *shared heap region* and the *private heap region*, such that accesses to the shared heap do not collide with private heap accesses. The used system-level software provides a set of synchronization primitives that ensures that a bank used by one thread is not remapped until this thread has finished. Hence, we can compute the WCET estimation of a software-pipelined parallel application.

IX. Acknowledgment

This work has been mainly funded by MERASA STREP-FP7 European Project under the grant agreement number 216415. M. Paolieri is partially supported by the Catalan Ministry for Innovation, Universities and Enterprise of the Catalan Government, and European Social Funds. E. Quinones is partially funded by the Spanish Ministry of Science and Innovation under the grant Juan de la Cierva JCI2009-05455

References

[1] *RapiTime: WCET analysis. www.rapitasystems.com*.
[2] *MERASA EU-FP7 Project: www.merasa.org*, 2007.
[3] S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *DAC '07*, pages 264–265, New York, NY, USA, 2007. ACM.
[4] JEDEC Solid State Techn. Assoc. *JEDEC DDR2 SDRAM Specification JEDEC Standard No. JESD79-2E*, April 2008.
[5] J. W. Lee and K. Asanovic'. Meterg: Measurement-based end-to-end performance estimation technique in qos-capable multiprocessors. In *Proc. RTAS*, pages 135–147, 2006.
[6] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A New Dynamic Memory Allocator for Real-Time Systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS '04)*, pages 79–86, Washington, DC, USA, 2004. IEEE Computer Society.
[7] S. Metzlaff, S. Uhrig, J. Mische, and T. Ungerer. Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. In *Proceedings of the 9th MEDEA workshop*, MEDEA '08, pages 38–45, New York, NY, USA, 2008. ACM.
[8] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems. In *ISCA '09*, pages 57–68, New York, NY, USA, 2009. ACM.
[9] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero. *An Analyzable Memory Controller for Hard Real-Time CMPs* . Embedded System Letters (ESL), 2009.
[10] C. Rochange, A. Bonenfant, P. Sainrat, M. Gerdes, J. Wolf, T. Ungerer, Z. Petrov, and F. Mikulu. Wcet analysis of a parallel 3d multigrid solver executed on the merasa multi-core. In *Workshop on Worst-Case Execution-Time Analysis in conjunction with ECRTS*, 2010.
[11] M. Schoeberl. Time-Predictable Computer Architecture. *EURASIP J. Embedded Syst.*, 2009:2:1–2:17, January 2009.
[12] M. Schoeberl, P. Puschner, and R. Kirner. A Single-Path Chip-Multiprocessor System. In *Proceedings of the Seventh IFIP Workshop on SEUS 2009*, number LNCS 5860, pages 47–57. Springer, 2009.
[13] S. Uhrig, S. Maier, and T. Ungerer. Toward a processor core for real-time capable autonomic systems. In *ISSPIT*, 2005.
[14] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, and J. Wolf. Merasa: Multi-core execution of hard real-time applications supporting analysability. *IEEE Micro*, 99(PrePrints), 2010.
[15] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. Dramsim: a memory system simulator. *SIGARCH Comput. Archit. News*, 2005.
[16] J. Whitham and N. Audsley. MCGREP – A Predictable Architecture for Embedded Real-Time Systems. In *Proc. of RTSS*, pages 13–24, Washington, DC, USA, 2006. IEEE Computer Society.
[17] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966 –978, 2009.
[18] J. Wolf, M. Gerdes, F. Kluge, S. Uhrig, J. Mische, S. Metzlaff, C. Rochange, H. Cassé, P. Sainrat, and T. Ungerer. RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-Core Processor. In *Proceedings of the 13th IEEE ISORC 2010*, pages 193–201. IEEE Computer Society, May 2010.