WCET ANALYSIS OF A PARALLEL 3D MULTIGRID SOLVER EXECUTED ON THE MERASA MULTI-CORE [1]

Christine Rochange[2], Armelle Bonenfant[2], Pascal Sainrat[2], Mike Gerdes[3], Julian Wolf[3], Theo Ungerer[3], Zlatko Petrov[4], František Mikulu[4]

Abstract
To meet performance requirements as well as constraints on cost and power consumption, future embedded systems will be designed with multi-core processors. However, the question of timing analysability is raised with these architectures. In the MERASA project, a WCET-aware multi-core processor has been designed with the appropriate system software. They both guarantee that the WCET of tasks running on different cores can be safely analyzed since their possible interactions can be bounded. Nevertheless, computing the WCET of a parallel application is still not straightforward and a high-level preliminary analysis of the communication and synchronization patterns must be performed. In this paper, we report on our experience in evaluating the WCET of a parallel 3D multigrid solver code and we propose lines for further research on this topic.

1. Introduction

The demand for computing power in embedded systems is ever growing as is the demand for new functionalities that will improve safety, comfort, number and quality of services, greenness, etc. Multi-core processors are now being considered as first-rate candidates to achieve high performance with limited chip costs and low power consumption. However, off-the-shelves components do not exhibit enough timing predictability to be used to design hard real-time (HRT) systems since the estimation of worst-case execution times (WCETs) would be infeasible or extremely pessimistic.

The MERASA project focuses on multi-core processors and system-level software for HRT embedded systems. The main issue is to guarantee the analyzability and predictability of WCETs. The proposed MERASA architecture features 2 to 16 cores, each one with simultaneous multithreading (SMT) facilities designed to support one HRT thread and up to three non real-time (NRT) threads. Private instruction and data scratchpad memories favor local memory accesses for HRTs and the shared memory hierarchy, including the common bus, features time-predictable arbitration policies. Target workloads are multiprogrammed, mixed-critical tasks, as well as multithreaded, including control and data parallelism. However, computing the WCET of a parallel application is not a straightforward process even when the hardware features timing predictability. As far as we know, this issue has not

[1]**M**ulti-Core **E**xecution of Hard **R**eal-Time **A**pplications **S**upporting **A**nalysability. This research is partially funded by the European Community's Seventh Framework Programme under Grant Agreement No. 216415
[2]IRIT - Université Paul Sabatier, Toulouse, France - {rochange,bonenfant,sainrat}@irit.fr
[3]University of Augsburg, Germany - {wolf,gerdes,ungerer}@informatik.uni-augsburg.de
[4]Honeywell International s.r.o., Czech Republic - {Zlatko.Petrov,Frantisek.Mikulu}@Honeywell.com

been addressed so far and the work that is reported in this paper is to be seen as a first attempt from which we will get feedback to guide future research.

Within the MERASA project, a collision avoidance application, developed by Honeywell International and based on the Laplace'equation for a 3D path planning, has been selected as example of an industrial-relevant application to support investigations on timing analyzability and predictability. In this paper, we study the core component of this application, a 3D multigrid solver. The parallelized version of this code splits the 3D domain into 3D compartments and creates one thread to process each compartment. We assume a number of threads equal to or lower than the number of cores in the target MERASA processor so that all the threads can run in parallel, each on one core. While the hardware guarantees isolation between concurrent HRT threads so that their respective WCETs could be analyzed without any difficulty as if they were independent, software-related dependencies due to shared data and synchronizations make the WCET analysis of the whole application challenging. In this paper, we show how we have estimated the WCET of the parallel 3D multigrid solver.

The paper is organized as follows. Section 2 introduces the multi-core processor and system software designed within the MERASA project. The parallel 3D multigrid solver is presented in Section 3. How the WCET of this application can be analyzed is discussed in Section 4 and experimental results are reported in Section 5. Section 6 provides feedback from this study and Section 8 concludes the paper.

2. The MERASA multi-core architecture and system software

The MERASA processor architecture was developed as a timing predictable and WCET-analyzable embedded multi-core architecture. Cores must execute HRT threads in isolation, and the hardware must guarantee a time-bounded access to shared resources by e.g. applying techniques for a real-time capable bus and memory controller [7]. In order to allow mixed application execution of HRT and NRT threads each MERASA core is an in-order SMT-core providing the possibility to run a HRT thread simultaneously in concert with additional NRT threads. The cores of our multi-core processor feature two pipelines, an address and a data pipeline, a first level of hardware scheduling to the thread slots, a real-time aware intra-core arbiter, a data scratchpad (DSP) and a dynamic instruction scratchpad (D-ISP) [5] on core level. The instruction scratchpad is automatically filled with target functions on call/return instructions so that (a) the load time only depends on the function size and can be accounted for when computing the WCET, and (b) every instruction fetch within a function always hit in the D-ISP. The real-time aware intra-core arbiters are connected to the real-time bus for accessing the main memory. The real-time bus features different real-time bus policies inspired from Round-Robin and priority-based schemes [6] for dispatching requests to the memory. Thus, we are able to execute multi-programmed and multithreaded workloads on our multi-core processor. A detailed description of the MERASA multi-core and its implementations as low-level (SystemC) and high-level simulators and also as an FPGA prototype are available at the website of the MERASA project (www.merasa.org).

The MERASA system software [11] represents an abstraction layer between application software and embedded hardware. On top of the MERASA multi-core processor it provides the basic functionalities of a real-time operating system (RTOS) like synchronization functions and memory management facilities to be a fundament for application software. Similar demands as for the processor architecture arise for the RTOS: the challenge is to guarantee an isolation of memory and I/O resource accesses of various HRT threads running on different cores to avoid mutual and possibly unpredictable interferences between HRT threads and therefore also enable WCET analyzability. If common resources are accessed, a time-bounded handling must be guaranteed. The resulting system software executes HRT threads in parallel on different cores of the MERASA multi-core processor. To yield thread isolation, we decided to apply a second level of hardware-based real-time scheduling and devised a thread control block (TCB) interface for the system software. The TCB interface is used to schedule by hardware a fixed number of HRT threads (fixed by the number of cores, one per core) and an arbitrary number of NRT threads into the available hardware thread slots. The commonly used POSIX-compliant mechanisms for thread synchronization, like mutex, conditional and barrier variables are implemented in a time-bounded fashion and a fixed WCET of each function was computed [11]. For the dynamic memory management we chose a flexible two-layered mechanism with memory pre-allocation in the first and the real-time capable TLSF [4] memory allocator in the second layer.

3. The multigrid solver application

3.1. General overview

The software considered in this study is part of a larger application for airbone collision avoidance. Moreover, it is the basic building block that plans a path between the current vehicle position and the current goal position, using the Laplace's equation. The Laplacian algorithm constructs paths through a 3D domain by assigning a potential value of $v(r) = 0$ for r on any boundaries or obstacles, and a potential value of $v(r) = -1$ for r on the goal region. Then Laplace's equation is solved in the interior of the 3D region, guaranteeing no local minima in the interior of the domain, leaving a global minimum of $v(r) = -1$ for r on the goal region, and maxima of $v(r) = 0$ for r on any boundaries or obstacle. A path from any initial point $r(0)$ to the goal is constructed by following the negative gradient of the potential v.

Numerical solutions of Laplace's equation are obtained by partitioning the domain, then iteratively setting the potential at each interior point equal to the average of its nearest neighbors. By varying the grid size (halving the voxel[2] size at each step) from the crudest that still leaves paths between obstacles, to the finest that is required for smooth paths, the iteration converges in a time proportional to the number of voxels in the finest grid. The solution on crude grids is cheap, and is used to initialize the solution on finer grids. This multigrid technique is described in [10].

In the version of the code we considered, the multigrid solver function includes five phases, each of them breaks down into an interpolation step and an iteration step, shown in Table 1. There are no data races in the interpolation code, while those are in the iteration step code.

3.2. Parallel version

This code has been parallelized by breaking the 3D domain down into 3D compartments, as illustrated in Figure 1. The main thread creates as many child threads as compartments and each of them performs the computations for one compartment. The main thread orchestrates the phases and steps and enforces the synchronization of the child threads after each step. During interpolation steps, the child threads run independently from each other. However a synchronization at the end is required to ensure that the whole 3D matrix has been processed before starting the iteration step. In the iteration

[2]A voxel is a volume element, representing a value on a regular grid in a 3D space. It is analogous to a pixel in a 2D space.

Table 1. Steps in the multigrid solver algorithm

Interpolation	Iteration
`for (x=0; x<NX; x++)`	`for (i=0; i<NUM_ITE; i++)`
`  for (y=0; y<NY; y++)`	`  for (x=0; x<NX; x++)`
`    for (z=0; z<NZ; z++)`	`    for (y=0; y<NY; y++)`
`      v[x][y][z]`	`      for (z=0; z<NZ; z++)`
`        = compInterpolate(old_v);`	`        v[x][y][z]`
	`          = compIterate(v);`

steps, each thread has to synchronize with the threads that produce the data it depends on and with the threads that consume the produced data.

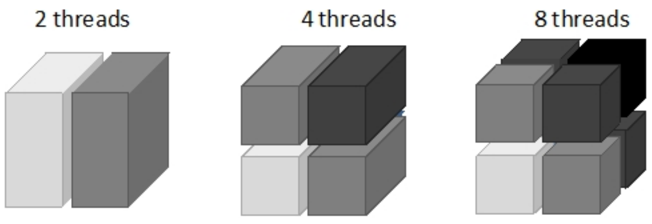


Figure 1. 3D domain splitting into compartments

4. WCET analysis of the parallel multigrid solver

4.1. General overview

Analyzing the WCET of a parallel application requires two steps. First, it is necessary to analyze the general structure of the code, and to determine which parts are executed in parallel. The result of this step is a slicing of the code into units and the specification of how units are scheduled with respect to each other. This specification allows deriving the list of code units for which a WCET must be computed and a formula to determine the WCET of the whole application from the WCETs of code units. Second, the synchronizations and communications between threads must be carefully analyzed to be able to compute upper bounds for waiting times due to synchronizations.
In the following, we will illustrate these two steps considering the multigrid solver application introduced in Section 3.

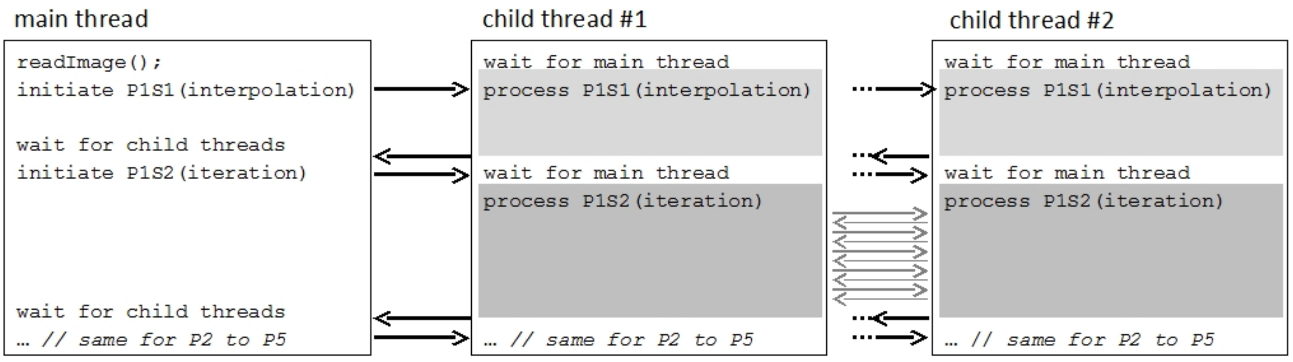
4.2. Analysis of the application structure

Figure 2 shows the structure of the parallel code, the synchronization points between the main thread and the child threads, as well as the synchronizations between child threads (in this figure, `PiSj` stands for "Phase `i`, step `j`"). From this structure, it is possible to derive a first breakdown of the overall WCET, as shown in Figure 3. This diagram combines the WCETs of code parts, some of them are executed by the main thread and the other ones by the child threads. The question of the synchronizations will be addressed in Section 4.3 but what is suggested here is that the WCET of a code part executed by the main *(resp. a child)* thread does not include the waiting time for other threads: the waiting time is represented by the WCET of child threads *(resp. of the main thread)*.

According to the diagram, the WCET can be computed as:

$$WCET_{global} = WCET(main) + \sum_{i=1}^{5}\sum_{j=1}^{2} WCET(P_i S_j)$$

where $WCET(main)$ is computed without accounting for the waiting times.



```
readImage();
initiate P1S1(interpolation)

wait for child threads
initiate P1S2(iteration)




wait for child threads
… // same for P2 to P5
```

```
wait for main thread
process P1S1(interpolation)


wait for main thread
process P1S2(iteration)




… // same for P2 to P5
```

```
wait for main thread
process P1S1(interpolation)


wait for main thread
process P1S2(iteration)




… // same for P2 to P5
```

Figure 2. Structure of the parallel multigrid solver code

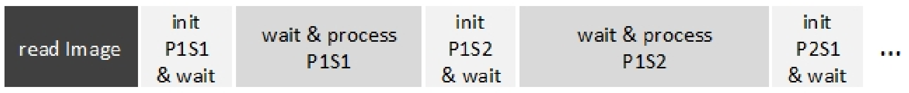


Figure 3. First-level breakdown of the WCET

Now, this application requires a second level of analysis due to the synchronizations between child threads in the iteration steps. As explained in Section 3.2, the sequential algorithm enforces data dependencies between successive iterations of the loop nest since the potential of a voxel is computed from the potential of its neighbors (some of them have been updated before it, other ones will be updated after it). Once the 3D domain is split into compartments, each one has dependencies with the borders of its neighbor compartments. Moreover, the grid is processed several times in a loop.

Fortunately, data sharing patterns are regular and it is possible to compute the WCET of a whole interpolation step as illustrated in Figure 4. The interesting point here is the absence of data races requiring synchronizations between the first compartments and the last ones which allows the first threads starting a new iteration of the outer loop before the last threads have finished the current iteration. This is demonstrated in Figure 4 by the overlapping of i0 threads with i1 threads in the 4-, respectively 8-threaded cases. The examples show that a computation speed-up of at most 2 in the 4-threaded and 4 in the 8-threaded cases can be achieved (not taking the main thread and the synchronization overheads into account).

As a result, if the WCET of each computation part is noted W_i and NUM_ITE is the number of iterations, the WCET of an iteration step can be computed as:

$$WCET(P_i S_2) = \begin{cases} 2.W_i \times \texttt{NUM\_ITE} & \textit{with 2 threads} \\ 3.W_i + 2.W_i \times (\texttt{NUM\_ITE} - 1) & \textit{with 4 threads} \\ 4.W_i + 2.W_i \times (\texttt{NUM\_ITE} - 1) & \textit{with 8 threads} \end{cases}$$

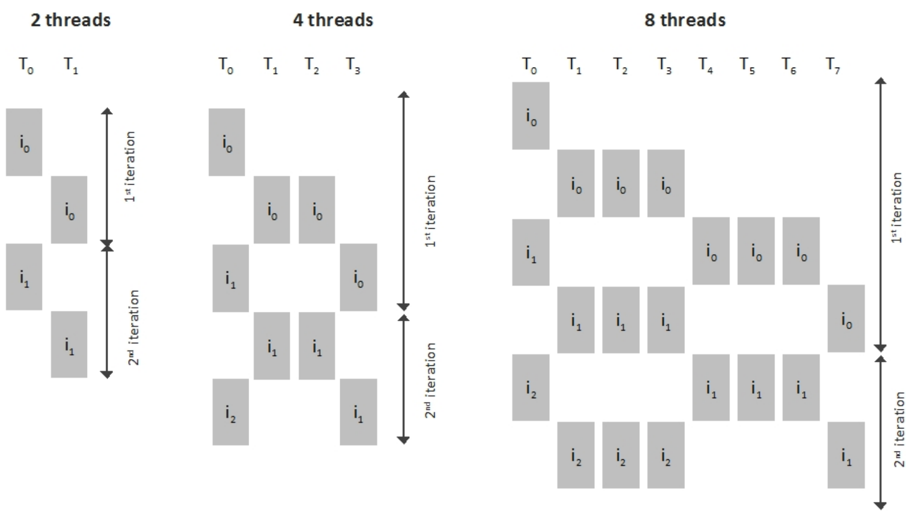


Figure 4. Second-level breakdown of the WCET

4.3. Analysis of the synchronizations

In the application under analysis, the inter-thread synchronizations are implemented using POSIX-compliant mutex and conditional variables. In this Section, we focus on the mutex variable acquisition (`mutex_lock`) but the approach that we propose to analyze the waiting time within this function is also valid for other synchronization primitives.
A simplified version of the `mutex_lock()` function is shown in Figure 5. There are four different cases for a thread trying to acquire the mutex lock. If the mutex lock is free, the thread will acquire the guard lock (1), the mutex lock, and release the guard lock (4). If the mutex lock is hold by another thread, the thread will, after acquiring the guard lock (1), suspend on the mutex lock (2). If the other thread releases the mutex lock, all suspended threads will try to get the mutex lock (3). Now, there are two possibilities: either a thread acquires the mutex lock (4) or it is suspended again (2) if an other thread acquired the mutex lock successfully. The worst-case waiting time for the guard lock is then the maximum WCET of all these four paths and of the similar paths in the other primitives that manipulate the guard lock, multiplied by the number of active threads.
In general, the WCET of a synchronization primitive can be broken down into four terms:

Term		depends on...	
		# threads	application
T_e	Execution time when the synchronization variable is free	no	no
T_{w1}	Overhead execution time when the thread has to wait for the variable	no	no
T_{w2}	Waiting time related to system-level variables	yes	no
T_{w3}	Waiting time related to application-level variables	yes	yes

As far as `mutex_lock()` is concerned, these times can be computed as follows:

- T_e is the WCET computed with flow facts indicating that the loops (the one that implements the wait for the mutex lock, but also the one that stands for the wait for the guard lock in `spinlock_lock()`) have zero iteration, which corresponds to the situation where both the guard lock and the mutex lock are free. This is a constant time.

- T_{w1} is the overhead time of waits (both for the guard lock and the mutex lock), i.e. the time to enter and then leave the loop, considering the lock is released right after the unsuccessful try. In practice, this time is the difference between T_e and the WCET computed considering each loop iterates once (ignoring the waiting time). This time is also constant.

- T_{w2} stands for the waiting times that relate to variables that are manipulated by system-level code only. This is the case for the guard lock. The knowledge of the system software code makes it possible to determine all the possible executions where a thread holds the guard lock. The time T_{w2} then depends on the number of threads but not on the application code, i.e. once the system software has been analyzed, this time is known for any application.

- Finally, T_{w3} includes the waiting times that relate to variables used at the application level. Here, the mutex lock is concerned. How long a thread will have to wait to get the mutex lock depends on the application code and more specifically on possible paths from any call to `mutex_lock()` to any call to `mutex_unlock()`. This time also depends on the number of threads that are likely to use the variable.

```
     int mutex_lock(mutex_t mutex) {
(1)    spinlock_lock(&mutex->guard);
     ...
        while (spinlock_trylock(&mutex->the_lock)) {
         ... // insert thread into queue
(2)      spinlock_unlock_and_set_suspended(&mutex->guard);
(3)      spinlock_lock(&mutex->guard); // on wakeup
       }
       ...
(4)    spinlock_unlock(&mutex->guard);
}
```

Figure 5. Code of the `mutex_lock` primitive

Three kind of synchronizations have to be considered when analyzing the WCET of the parallel multigrid solver. Synchronizations from the main thread to the child threads rely on a shared variable that indicates the next step to execute. It includes a mutex lock (to protect accesses to the shared variable) and a condition to wait on when the variable is not in the expected state. With N child threads, the number of threads that manipulate the lock and the condition is $N + 1$. The paths on which the lock can be hold can be identified in the main thread and child thread codes: the maximum WCET of these paths, multiplied by $N + 1$, makes the T_{w3} term for the lock. Synchronizations from the child threads to the main thread (it must wait that they all have performed the current step before initiating the next one) are similar except for each thread has its own state variable (which is shared only with the main thread). Synchronizations between two child threads (when one produces data used by the other one, on compartment borders) are implemented through state variables related to each child thread. As above, state variables include mutex lock and conditions. Each state variable is shared with the producers and consumers of its owner.

5. Experimental results

In this Section, we provide experimental results obtained considering the following configuration for the MERASA multi-core processor: 2 to 8 cores available for computation threads (an additional core is considered for the main thread), perfect ISP (all the instructions can be fetched from the instruction scratchpad memory), DSP (scratchpad for stack data), round-robin bus, 5-cycle DRAM latency. Due to the round-robin policy and to intra-core arbitration between real-time and non-real-time threads, the *worst-case* latency of an access to the main memory is $5 \cdot n + 12$ for an n-core configuration. In addition, we have considered a single-core configuration to calculate the WCET of a single-threaded version of the application. The application was compiled to the TriCore ISA [13].

WCETs have been computed using OTAWA, a toolset based on static WCET analysis techniques [2]. A specific model of the MERASA architecture has been implemented within this framework. To control the WCET analysis, we have written a script that initiates all the required computations. First, it gets all the WCET terms needed for system-level synchronization functions; then, it requests the WCET analysis of application-level code parts (main function, computation steps); finally, it combines all these times to compute the overall WCET.

Table 2. Estimated WCETs (# cycles)

	1 thread	**2+1 threads**	**4+1 threads**	**8+1 threads**
1 core	53,990,765	-	-	-
3 cores	58,297,375	66,849,369	-	-
5 cores	62,603,985	73,372,109	40,389,428	-
9 cores	71,217,205	86,417,589	47,678,968	28,105,761

Table 2 shows our WCET estimates for several configurations (from 1 to 9 cores) and for several versions of the application (from 1 to 8+1 threads). We considered small grid sizes so that only three phases (instead of five) are executed. As expected, the WCET of a t-threaded version of the application increases with the number of cores: this is due to the round-robin bus policy under which, in the worst case, a given core has to wait for all the other cores to be served before being served itself. Then the worst-case memory latency increases linearly with the number of cores.

On the other hand, the WCET is noticeably improved when the application is parallelized to 4 threads and more. We define the *WCET-speedup* as the WCET of the single-threaded code executed on one core over the WCET of the n-threaded code executed on n cores. On a 9-core architecture, the WCET-speedup is 1.13 with 4 computation threads and 1.9 with 8 computation threads.

Table 3 indicates how the WCET breaks down into ”active” execution time (ignoring all waits), waiting time due to synchronizations and waiting time for producing threads (in iteration steps, a computation thread has to wait until all the previous compartments have been updated by other threads before starting to update its own compartment). The provided numbers are for a 9-threaded version of the code and a 9-core configuration, but other combinations exhibit similar breakdowns. It appears that the part of the execution time due to the main thread is small (6%): it mainly includes the time to read the grid. Moreover the synchronizations have little impact on the overall WCET: they account for less than 2%. This is due to limited interactions between concurrent threads. The largest part of the total WCET (more than 92%) comes from the computation threads. About one half of this time is spent in performing the computations assigned to a thread (three interpolation steps and three iteration steps for the map size selected for this study) while the other half is spent in waiting for producing threads within iteration steps. This is coherent with Figure 4 that shows that the length of loop body in an iteration step should be twice the length of computing one compartment. Predominance of the iteration steps duration in the total WCET explains why the dual-threaded version of the code does not improve the WCET but instead degrades it: since parallelism is not exploited in the iteration steps (both compartments must be processed in sequence), the gain due to parallelized interpolation steps is not sufficient to counterbalance the cost of synchronizations.

Table 3. WCET breakdown (8 computation threads, 9 cores)

WCET for the main thread without waits	6.0%
WCET for a computation thread without waits	45.0%
Time to wait for producers in iteration steps	47.2%
Total waiting time for synchronizations	1.8%

6. Lessons learnt from this case study

While research on WCET computation of sequential programs has received much attention the last fifteen years, resulting in a set of techniques that can handle not too complex software and hardware, the arrival of multi-cores on the embedded systems market raises the need of investigating strategies to analyze the WCET of parallel applications. In this paper, we have reported a study that has been carried out as part of the MERASA project. This first experience in the domain has inspired new lines of research that we will outline in this Section.

The first need for the WCET analysis of a parallel program that appeared during this study is to get an overview of the structure of the application. It is necessary to determine which parts of the code execute in parallel and where the dependencies are. This knowledge is required to build the first-level breakdown of the overall WCET and to schedule the analysis of each piece of code that must be taken into account. Automatically extracting this kind of information from the source or executable code seems infeasible and it is likely that the user/programmer will be asked to provide them. However, this might be a source of errors and it may be desirable to favor well-known parallelization patterns. In addition, an appropriate formalism to express the software parallel architecture would be helpful.

The second need concerns synchronizations: to compute worst-case waiting times, it is necessary to know where synchronizations occur and which threads share synchronization variables. Moreover, the paths on which locks are hold by threads must be carefully identified. Again, specific support should be provided (e.g. in the form of an annotation language) so that the user can specify these information.

In addition, the considered WCET analysis tool must be enhanced to be able to process the specifications of the parallelized code (structure and synchronizations). In particular, we have identified as a must functionality the ability to compute the WCET of a given path (from address a to address b) taking into account its possible contexts of execution. For example, the cache behavior should be preliminary analyzed considering the whole program instead of performing the cache analysis on the path only. So far, the OTAWA toolset does not support global context analysis for path WCET estimation. This is the reason why we have considered a perfect instruction scratchpad in this paper.

Finally, we would like to point out that WCET analysis might not be feasible for any parallel code. As future work, we plan to provide a set of parallelization recommendations that will serve as guidelines to split the computations into parallel threads so that both the code structure and the communication and synchronization patterns match the requirements for timing analyzability.

7. Related work

As we have mentioned it, we are not aware of any paper dealing with the WCET analysis of data-parallel applications and in particular with the analysis of synchronization delays. However, several recent works have studied the impact of interactions between concurrent threads on the WCET.

Some papers focus on the delays introduced by the other threads in the interconnection structure (mainly at the bus level). Solutions reside in predictable arbitration schemes like Round-Robin (considered in the MERASA architecture) [6] or TDMA schemes [1]. A different approach integrates task- and system-level analyses to derive upper bounds for memory latencies [8].

Other contributions concern the analysis of interactions in the level-2 shared cache [12] or solutions to make this analysis more accurate [3][9]. They all focus on spatial conflicts (when two concurrent threads may access the same cache line in a shared cache), not on temporal interferences (i.e. their cache analysis does not take into account the times at which conflicting accesses to the cache will occur but considers instead the worst case).

8. Conclusion

Multi-core processors seem to be key components for the design of future embedded systems because of their ability to provide high performance with low cost and low power consumption. However, existing designs are often not compatible with worst-case execution time analysis since the dynamic sharing of resources (bus, memory hierarchy, etc.) makes it complex if not impossible to determine upper bounds on the delays experienced by a thread running on one of the cores that are due to other threads running on other cores (or on the same core if simultaneous multithreading is supported). The MERASA project fills in this gap by developing a hard-real-time-aware multi-core processor and the appropriate system software. Both levels (hardware and software) have been designed keeping in mind the need of being able to determine upper bounds on pipeline, bus and memory latencies as well as on delays related to dynamic memory management and inter-thread synchronization. As a result, the MERASA multi-core can be used to run mixed multiprogrammed workloads with critical and non critical tasks as well as parallel programs subject to timing constraints.

However, even with WCET-friendly hardware, the WCET analysis of a parallel application is still challenging because it requires having a clear view on the code parallel structure and on the communication and synchronization patterns. We have investigated this field and this paper reports our first experiment in computing the WCET of a parallel 3D multigrid solver code. The process described here is completely guided by the user who must provide insight into how the complete code breaks down into units that can be analyzed separately before their respective WCET estimates are combined to produce the total WCET. As future work, we plan to work on automating the analysis as much as possible.

References

[1] ANDREI A., ELES P., PENG Z., ROSEN J.: Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In: 21st International Conference on VLSI Design, 2008.

[2] CASSE H., SAINRAT P.: OTAWA, a Framework for Experimenting WCET Computations. In: 3rd European Congress on Embedded Real-Time Software, 2006.

[3] HARDY D., PIQUET T., PUAUT I.: Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In: IEEE Real-Time Systems Symposium (RTSS), 2009.

[4] MASMANO M., RIPOLL I., CRESPO A., REAL J.: TLSF: A New Dynamic Memory Allocator for Real-Time Systems. In: 16th Euromicro Conf. on Real-Time Systems (ECRTS), 2004.

[5] METZLAFF S., UHRIG S., MISCHE J., UNGERER T.: Predictable Dynamic Instruction Scratchpad for Simultaneous Multithreaded Processors. In: 9th workshop on MEmory performance (MEDEA), 2008.

[6] PAOLIERI M., QUIÑONES E., CAZORLA F., VALERO M.: Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems. In: 36th Int'l Symp. on Computer Architecture (ISCA), 2009.

[7] PAOLIERI M., QUIÑONES E., CAZORLA F., VALERO M.: An Analyzable Memory Controller for Hard Real-Time CMPs. In: IEEE Embedded Systems Letters, 1(4), 2009.

[8] STACHULAT J., SCHLIECKER S., IVERS M., ERNST R.: Analysis of Memory Latencies in Multi-Processor Systems. In: 5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, 2007.

[9] SUHENDRA V., MITRA T.: Exploring Locking and Partitioning for Predictable Shared Caches on Multi-cores. In: 45th Conf. on Design Automation (DAC), 2008.

[10] VALAVINIS K.P., HERBERT T., KOLLURA R.,TSOURVELOUDIS N.: Mobile Robot Navigation in 2-D Dynamic Environments Using an Electrostatic Potential Field. In: IEEE Trans. on Systems, Man, and Cybernetics-PART A: Systems and Humans, 30(2), 2000.

[11] WOLF J., GERDES M., KLUGE F., UHRIG S., MISCHE J., METZLAFF S., ROCHANGE C., CASSE H., SAINRAT P., UNGERER T.: RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-Core Processor. In: 13th IEEE Int'l Symp. on Object/component/service-oriented Real-time Distributed Computing (ISORC), 2010.

[12] YAN J., ZHANG W.: WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2008.

[13] Infineon Technologies AG, TriCore 1 User's Manual (v1.3.8), Jan., 2008.