Integrated Worst-Case Execution Time Estimation of Multicore Applications*

Dumitru Potop-Butucaru[1] and Isabelle Puaut[2]

1 **INRIA, Paris-Rocquencourt,** `dumitru.potop@inria.fr`
2 **University of Rennes 1/IRISA, Rennes,** `isabelle.puaut@irisa.fr`

Abstract

Worst-case execution time (WCET) analysis has reached a high level of precision in the analysis of sequential programs executing on single-cores. In this paper we extend a state-of-the-art WCET analysis technique to compute *tight* WCETs estimates of parallel applications running on multi-cores. The proposed technique is termed *integrated* because it considers jointly the sequential code regions running on the cores and the communications between them. This allows to capture the hardware effects across code regions assigned to the same core, which significantly improves analysis precision. We demonstrate that our analysis produces tighter execution time bounds than classical techniques which first determine the WCET of sequential code regions and then compute the global response time by integrating communication costs. Comparison is done on two embedded control applications, where the gain is of 21% on average.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases WCET estimation, multicore architectures, parallel programming

Digital Object Identifier 10.4230/OASIcs.WCET.2013.21

1 Introduction

Multi-core systems are becoming prevalent in both general purpose and embedded systems. Their adoption is driven by scalable performance arguments, but this scalability comes at the price of increased software complexity. Indeed, multi-core systems run parallel software involving potentially complex synchronizations between the sequential programs executed on the various cores. In the current state of the art of validation of real-time multi-task software, temporal validation is achieved by computing the worst-case response time (WCRT) of every task, defined as an upper bound for the duration between the task arrival and its termination. Two main classes of techniques, usually applied sequentially, are used: (i) Worst-case execution time (WCET) estimation, which works on sequential programs, and (ii) WCRT estimation, that computes response times thanks to WCET values as inputs.

WCET analysis emphasizes the importance of hardware micro-architecture. Indeed, in its double quest for execution speed and programming simplicity, modern hardware architectures include user-transparent performance enhancing features (e.g., pipelining, caching). The presence of these elements complicates WCET estimation. Limiting generality to sequential code running on single-cores and selecting moderately complex hardware allows the preservation of computational tractability, while the hardware micro-architecture is precisely modeled. This allows the computation of tight execution time bounds.

* This work was partially supported by EU COST Action IC1202 Timing Analysis at Code-Level (TACLe)

13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013).
Editor: Claire Maiza; pp. 21–31

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

WCRT analysis emphasizes the system-level complexity, by taking into account aspects such as inter-task task communication/synchronization, and interaction with the environment. The objective here is usually to provide execution time bounds for execution flows involving several tasks, possibly running on multiple processors, and their communications and synchronizations. To limit computational complexity of WCRT analysis, hardware and software are usually represented in much less detail than in WCET estimation techniques. Typical objects at this level are sequential tasks characterized by functional and non-functional properties, such as: inputs and outputs, WCET, period, execution conditions, etc.

When dealing with parallel applications running on multicore architectures, the classical separation between WCET and WCRT analysis has to be revisited, since an application, even when considered in isolation from the others, includes parallelism. In this paper, we address the issue of determining the WCET of an isolated parallel application where each core is statically allocated one sequential thread. Since the threads synchronize with each other, our *integrated* WCET estimation technique must address issues that are usually dealt with by WCRT estimation techniques (integration of synchronization and communication costs). On the other hand, as a WCET estimation technique, our proposal calculates execution times of a parallel application considered in isolation from the other activities that run concurrently on the multi-core architecture.

Contribution. Classical WCRT-based timing analysis techniques for parallel code isolate micro-architecture analysis from the analysis of synchronizations between cores by performing them in two separate analysis phases (WCET and WCRT analysis). This isolation has its advantages, such as a reduction of the complexity of each analysis phase, and a separation of concerns that facilitates the development of analysis tools. But isolation also has a major drawback: a loss in precision which can be significant. To consider only one aspect, to be safe the WCET analysis of each synchronization-free sequential code region has to consider an undetermined micro-architecture state. This may result in overestimated WCETs, and consequently on pessimistic execution time bounds for the whole parallel application. The contribution of this paper is an *integrated* WCET analysis approach that considers at the same time micro-architectural information and the synchronizations between cores. This is achieved by extending a state-of-the-art WCET estimation technique and tool to manage synchronizations and communications between the sequential threads running on the different cores. The benefits of the proposed method are twofold. On the one hand, the micro-architectural state is not lost between synchronization-free code regions running on the same core, which results in tighter execution time estimates. On the other hand, only one tool is required for the temporal validation of the parallel application, which reduces the complexity of the timing validation toolchain.

Such a holistic approach is made possible by the use of deterministic and composable software and hardware architectures (homogeneous multi-cores without cache sharing, static assignment of the code regions on the cores) as detailed later in this paper. We demonstrate the interest of the approach using an adaptive differential pulse-code modulation (*adpcm*) encoder where the integrated WCET approach provides significantly tighter response time estimations than the more classical WCRT approaches.

Outline. The rest of this paper is organized as follows. Section 2 presents the application model and defines more formally what is meant by worst-case execution time of a parallel application. Section 3 details and motivates the class of multi-core architectures considered in this study. Section 4 defines our WCET estimation method. Experimental results are given in Section 5. Our proposal is briefly compared to related work in Section 6. Finally, we conclude and discuss future work in Section 7.

```
void core1() {
  int tqmf[24]; long xa, xb, el;
  int xin1, xin2, decis_levl;
  for(;;) { //Infinite loop
    // Computation phase 1
    xa = 0; xb = 0;
    for (i=0;i<12;i++) { // 12 iterations
      xa += (long) tqmf[2*i]*h[2*i];
      xb += (long) tqmf[2*i+1]*h[2*i+1];
    }
    // Send the results to core 2
    send(channel1,(int)((xa+xb)>>15)); ------
    // Read inputs
    xin1=read_input(); xin2=read_input();
    // Computation phase 2
    for(i=23;i>=2;i--) { // 22 iterations
      tqmf[i]=tqmf[i-2];
    }
    tqmf[1] = xin1; tqmf[0] = xin2;
    // Receive data from core2 and output it
    decis_levl = receive(channel2) ; <------
    write_output(decis_levl) ;
  }
}
```

```
const int decis_levl [30];
int core2() {
  int q,el;

  for(;;) {//Infinite loop




    // Receive data from core1
--> el = receive(channel1);
    // Computation phase 1
    el = (el>=0)?el:(-el);
    for (q = 0; q < 30; q++) {
      // 30 iterations
      if (el <= decis_levl[q])
        break;
    }
    // Send result to core1
--- send(channel2,decis_levl) ;

  }
}
```

Figure 1 Toy example: parallel version of *adpcm*, from the Mälardalen WCET benchmark suite [8].

2 Application model and problem formulation

The simplest embedded control systems running on *mono-processor* architectures follow a so-called *simple control loop* paradigm. In such systems, the software is simply a loop whose body is the sequence of calls to the various input sampling, processing, and actuation functions. The sequence of calls is fixed off-line. In this paper, we consider the multi-core equivalent of simple control loops, where each core executes a simple control loop (the practical importance of such a code structure mainly derives from the use of automatic mapping techniques [7], which often generate such code). We shall denote with task τ_i the program that forms the body of the loop executed by core $CPU_i, 1 \leq i \leq n$. Each task τ_i satisfies the classical requirements allowing WCET analysis (all loops it contains except the main loop have statically bounded numbers of iterations).

Tasks $\tau_i, 1 \leq i \leq n$ can communicate with each other through a set of logical message-passing channels $\mathcal{C} = \{c_1, \ldots, c_m\}$ which are bounded FIFO buffers that do not lose, duplicate, or corrupt messages. Communication is done using *send* and *receive* primitives that can be invoked at any statically known position in task τ_i. The two primitives are blocking (*send* on full channel, *receive* on empty channel), which means that channels can be used for synchronization. For the scope of this paper, we make the following assumptions concerning the channels: (i) each channel connects exactly two processors (one sender and one receiver); (ii) Each channel allows the storage of only one message. No assumption is made on how the logical message passing channels are implemented on the execution platform. We also assume that inter-task communications are free of deadlocks by construction.

An illustrating application, that will be used all along the paper, is given in Figure 1. The application is a portion of a bi-processor parallel version of *adpcm* (adaptive pulse code modulation) from the Mälardalen WCET benchmark suite [8]. We emphasized with arrows the two `send`/`receive` pairs associated with `channel1` and `channel2` respectively.

Assuming the previously-defined application model, the worst-case execution time analysis problem we solve in this paper is to compute the worst-case duration of a fixed number of iterations of the application, considered in isolation from the other activities running concurrently on the multi-core architecture.

3 Execution platform

As noted by Puschner *et al.* [14], obtaining *precise* and *composable* timing information in a multiprocessor system is only possible if we can ensure spatial or temporal separation between concurrent accesses to shared resources. The shared resources we consider in our work are the memory subsystem, the on-chip buses and networks (including I/O), DMA controllers, and the synchronization subsystem. We shall make on all of them hypotheses that allow both a precise timing characterization of complying architectures, and the modeling of complex, real-life architectures. Multi-processor systems with shared caches, although amenable to WCET estimation, may yield pessimistic WCET estimates, because the state of these caches becomes difficult to approximate in the presence of concurrent requests. Our choice is to consider architectures where each processor has its private cache subsystem, independent from the ones of other processors. It is also assumed that each core has *separate* instruction and data caches. We consider such architectures because separate caches are analyzable more precisely than unified caches by WCET estimation techniques. All caches have a Least Recently Used (LRU) replacement policy. LRU is selected because it was shown to be the most predictable cache replacement policy [15]. Finally, cores are homogeneous (all cores have the same micro-architecture).

Another significant source of WCET estimation imprecision is the presence of shared memory banks and shared communication busses. Our choice here is to consider architectures where the duration of all memory accesses and data transmissions can be precisely determined. The timing precision can be ensured: (i) fully by hardware mechanisms, for instance through the use of time division (TDM) memory controllers or on-chip buses [5], (ii) or through a mix of software and hardware mechanisms. In these cases, software and/or hardware synchronization mechanisms (semaphores, locks) are used to guarantee the absence of contentions due to access to RAM banks or communication buses. In this paper we consider architectures of the second type, as this case covers classical distributed bus-based architectures, shared memory architectures featuring multiple RAM banks, but also mixes of the two, such as the Network-on-Chip (NoC) based architectures proposed by various vendors [17, 12]. Our NoC-based experimentation platform that will be detailed in Section 5 falls in this last case. An upper bound of the communication latency for every *send*/*receive* pair is assumed known, which is realistic on all the previously-mentioned architectures. The determination of communications latencies is considered outside the scope of the paper.

4 WCET computation

Our approach to WCET computation for parallel applications (§ 4.2) consists in extending a state-of-the-art WCET estimation method (§ 4.1) to compute WCETs of parallel applications.

4.1 Existing state-of-the-art WCET estimation technique

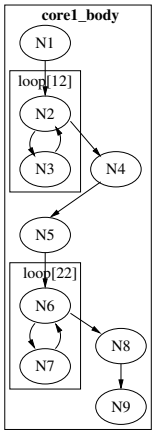
Static WCET estimation techniques are commonly organized in three phases performing different analyses [4]: *Control-flow analysis*, *Hardware-level analysis* and *WCET calculation*.

Control-flow analysis. This phase extracts information about possible execution paths from the program source or binary. The output of this phase is a data structure representing the possible flows. For the scope of this paper, this phase produces Control Flow Graphs (CFG), extracted from the program binary. The CFGs are annotated with additional flow

information such as maximum number of loop iterations. The CFG of the loop body of the task running on *core1* (from our sample application) is given in Fig. 2(a).

Hardware-level analysis. This step, also called *low-level analysis*, estimates the worst-case execution times of basic blocks. The difficulty during this phase is to take into account micro-architectural components of the target processor (caches, pipelines, branch predictors). In the presence of such components, the execution time of a statement is dependent on the context it is called in. The overall typical outcome of hardware-level analysis is a maximum execution time per basic block in two different contexts to cope with cache effects: the first execution of the basic block in a loop, denoted t^f and its subsequent executions, denoted t^n, as more formally defined in [9]; (negative) execution times may also be associated to edges to account for pipeline effects between basic blocks.

WCET calculation. The purpose of this final phase is to determine an estimate for the WCET, based on the flow and timing information derived in the previous phases. The most widespread calculation method, that will be adopted in this paper, is called *implicit-path enumeration* (IPET). In IPET, program flow and basic-block execution time bounds are combined into sets of arithmetic constraints. Each entity (basic block or program flow edge) in the code is assigned two values: a time coefficient, denoted t_{entity}, which expresses the upper bound of the contribution of that entity to the total execution time every time it is executed, and count variable (x_{entity}), corresponding to the number of times the entity is executed. The program's WCET of the program is determined by maximizing the sum of products of the execution counts and times ($\sum_{i \in entities} x_i * t_i$), where the execution count variables are subject to constraints reflecting the structure of the code and possible flows. The result of an IPET calculation is an upper timing bound and a worst-case count for each execution count variable.



(a) CFG of the loop body of `core1`

// Start constraint
$x_1 = 1$
// Structural constraints
$x_1 = x_{1,2}$
$x_2 = x_{1,2} + x_{2,3} = x_{2,3} + x_{2,4}$
$x_3 = x_{2,3} = x_{3,2}$
$x_4 = x_{2,4} = x_{4,5}$, ...
// Loop bound constraints
$x_3 \leq 12$
$x_7 \leq 22$
// Cache-induced constraints
$x_1 = x_1^f + x_1^n$
$x_1^f \leq 1$, ...
// WCET expression
maximize($x_1^f * 10 + x_1^n * 10$
$+x_2^f * 120 + x_2^n * 14 + ...$
$+x_{1,2} * -1 + ...$);

(b) Constraints for WCET calculation of `core1_body`

Figure 2 State-of-the-art WCET calculation on the loop body of `core1` in our illustrating example.

Fig. 2(b) illustrates the constraints and formulas generated by an IPET-based bound calculation method on the loop body of `core1`, assuming it is the program entry point. The start constraint states that the code is executed once. The structural constraints reflect the possible program flows, meaning that each basic block must be entered the same number of times as it is exited. The loop bounds constrain the number of executions of basic blocks inside loops. Cache induced constraints express that basic blocks have different execution times, one for their first execution, another for the next ones. In the WCET expression to be maximized, there are two execution durations per basic block to model cache effects. For instance, in the formula, $t_2^f = 120$ (first execution, cold cache), whereas $t_2^n = 14$ (subsequent executions, warmed-up cache).

4.2 WCET computation of parallel applications

Starting from the WCET estimation method sketched above, we build our WCET estimation technique for parallel applications by performing a per-core hardware-level analysis and then adding new edges in the CFG to model synchronization/communications between code regions. The modified analyzer runs as follows, with the analysis phases presented in their invocation order:

1. The original **control flow analysis** extracts the CFG of the tasks to be run on the cores, from the application binary.
2. The **hardware-level analysis** runs *unmodified* on each task (control loop running on each core). During this step, each task is analyzed as if it was not communicating with the other tasks executed concurrently on the other cores.
3. A new step dubbed **modeling of communications**, is invoked. This step adds new edges between the control flow graphs of these tasks, the result being a single CFG. A new edge is added for each communication between code regions; it is associated with a duration to model its execution time (message transmission time for communications).
4. The **WCET computation** step is executed unmodified, even though the CFG corresponds to a parallel application and has slightly different topological properties. This is due to the fact that the analysis works by finding the critical path in a directed acyclic graph. The fact that the graph represents purely sequential behaviors, or parallel ones (including the new edges that model communications) is not important.

The method was integrated into the Heptane static WCET estimation tool [1] through the addition of a new pass, corresponding to phase 3, interposed between hardware-level analysis and WCET calculation. Communications between code regions are detected through annotations in the source code of the analyzed application, specifying at each communication point the recipient of the message and the communication latency. The code of the new pass represents around 200 lines of C++ code.

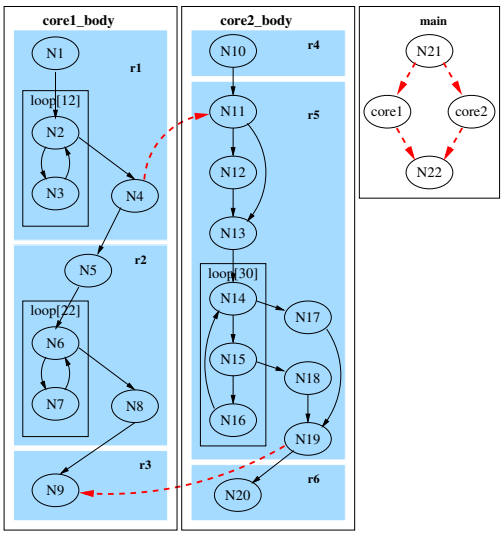


Figure 3 WCET computation of parallel application.

The method is illustrated step by step in Fig. 3 on our toy application of Fig. 1. The shaded areas labeled **r1**–**r6** correspond to the code regions, which are by definition the portions of the two tasks that are separated by communications. For instance, node N4 of the `core1_body` CFG is the basic block containing the `send` call on `channel1`. After the hardware-level analysis phase runs unmodified on the tasks of the parallel program, the modeling of communications adds new edges in the application CFG to model communications (bold arrows in the figure). These new edges correspond to: message passing between code regions (edges $N4 \rightarrow N11$ and $N19 \rightarrow N9$) and parallel launching of code regions on the different cores (edges to and from nodes $N21$ and $N22$ in the application entry point *main*).

During the hardware-level analysis phase, our WCET analysis method applies instruction cache, data cache, and pipeline analysis on the two CFGs `core1_body` and `core2_body`. This allows to benefit from the tightness of hardware-level analysis on each task. For instance, in task `core1_body`, it allows to detect that array *tqmf* is still in the data cache after calling primitive *send*. This would not have been possible if a *decoupled* approach was used (WCET estimation of regions followed by an aggregation of individual WCETs to compute the global WCET). If a decoupled method was used, conservative assumptions would have been taken for the analysis safety (assuming the worst-case hardware state, i.e. empty cache at WCET analysis start). Using an integrated approach, the hardware-level analysis is able to capture hardware effects between regions (instruction caches, data caches, pipeline) naturally.

Finally, the WCET computation step is applied unmodified. Thanks to the introduction of the new edges, new constraints are automatically added in the WCET calculation equations, and communication delays are automatically taken into account. The new or modified formulas of the WCET calculation equations are illustrated below for our running example, with the modified parts in bold face. Communication/synchronization edges are taken into account in the new structural constraints (e.g. number of executions of communication/synchronization edges, $x_{4,11}$). Data transmission latencies are considered as well (e.g. 250 time units to communicate data from node $N4$ to node $N11$ according to the amount of data to be transmitted between the two nodes).

// New or modified structural constraints (non exhaustive)
$x_4 = x_{2,4} = x_{4,5} + \mathbf{x_{4,11}}$
$x_{11} = x_{10,11} + \mathbf{x_{4,11}} = x_{11,12} + x_{11,13}$
// New WCET expression
$\text{maximize}(x_1^f * 10 + x_1^n * 10 + x_2^f * 120 + x_2^n * 14 + \mathbf{x_{4,11}} * \mathbf{250} + x_{1,2} * -1 + ...);$

5 Experimental evaluation

5.1 Experimental setup

Multi-core architecture. Given that our claims mainly concern the precision of the timing analysis, we considered an evaluation platform allowing us to perform cycle-accurate estimations and measurements of execution time, in both single-processor and multi-processor cases. We achieved this by using the SoCLib library [16] for virtual prototyping of multi-processor systems-on-chips (MPSoC). The hardware components we use are of cycle-accurate, bit accurate type, written in SystemC.

The precise architecture we worked on using SoCLib is a scaled-down version of that of [2]. While the original platform scales up to 4096 cores, we have only used for the presented experiments single-, double-, and quad-core configurations. Each core has separate L1 instruction and data caches, both implementing a Least Recently Used (LRU) cache replacement policy. Both caches feature 32 sets, 4 ways, and 32 bytes per cache line. All CPU cores are of the same type, using the MIPS32 instruction set. Each core is part of a computing tile containing a multi-bank RAM (to accommodate non-interferent concurrent accesses to program text and data by the CPU cores), a DMA unit, and a hardware lock unit. The local interconnect of each tile is a full crossbar. The tiles are inter-connected through a 2D mesh network-on-chip. The overall structure of our architecture is very similar to that of commercial many-core architectures [12].

Studied applications. The proposed WCET estimation method was experimented on two small signal processing applications. The first one is a parallel version of the *adaptive differential pulse-code modulation (adpcm)* from the Mälardalen WCET benchmark suite [8]. The global dataflow of the code executed at each iteration of the modulation application is depicted in Fig. 4, where boxes represent code regions and arrows communications between them. Fig. 4 also depicts mapping of regions to cores when the application is parallelized for a 2-core architecture. Only the arrows crossing CPU boundaries are coded as communications; the sequencing of QMF, low-band encoder and multiplexer is implemented simply by calling successively the three codes in the main loop of CPU0. When parallelized for a 4-core architecture, every region is assigned to a different core, and software pipelining is used to allow more parallelism. The communication latency of every inter-core communication was determined by an analysis of the hardware platform as a formula dependent on the volume of data to be transferred.

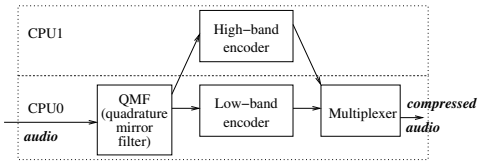


Figure 4 *adpcm* application: dataflow and mapping on a 2-core architecture.

The second application, named *filter*, is a simple load balancing example, where two processors are needed to improve the throughput of a simple image filter. In this bi-processor application, processor 0 successively receives image lines in a buffer. The buffer content must be stored elsewhere to allow a new line to arrive, and this new line will be sent to processor 1, cyclically.

Application code was compiled using a standard GNU MIPS compilation toolchain with no optimization. For the scope of this performance evaluation, application code was parallelized manually. Automatic code parallelization software like [6] or offline real-time scheduling tools like [7] that generate efficient parallel code could have been used instead.

5.2 Experimental results

Experimental results are given: (i) to evaluate the accuracy of the hardware model used in the base timing analysis tool; (ii) to compare WCETs obtained using our *integrated* approach against those obtained using a *decoupled* estimation method; (iii) to evaluate the pessimism of our integrated method. We do not give numbers on the run-time of the analysis, simply because modifying the application's CFG turned out to take negligible time compared to hardware-level analysis and WCET calculation.

Accuracy of hardware model. To show the accuracy of the hardware model used in the analysis, we have validated Heptane's hardware model against the SoCLib simulator on single-path code. Experiments were conducted on randomly generated single-path code, starting with known contents of the instruction and data caches. After a careful and extensive comparison of the analyzer and simulator cycle counts, both tools returned exactly the same number of cycles for all considered code.

Comparison with baseline decoupled WCET estimation method. To evaluate the tightness of WCET estimates, we have compared them with a baseline decoupled approach that first estimates WCETs of code regions and then computes the overall WCET through a composition of the regions WCETs. The baseline method operates as follows. It first computes computes WCETs of all regions; to be safe, the worst-case hardware state is assumed by the

Table 1 Experimental results: computed WCET bounds using our integrated approach and a base-line decoupled approach, and measured execution time. Improvement over the baseline is defined as $\frac{Decoupled-Integrated}{Integrated} * 100$. Analysis pessimism is defined as $\frac{Integrated-Measured}{Measured} * 100$.

Name	Integrated	Decoupled	Gain (%)	Measured	Pessimism (%)
adpcm – 2 cores	73563	101431	**36.5%**	64944	**13.3%**
adpcm – 4 cores	44568	55919	**25.5%**	41468	**7.5%**
filter – 2 cores	110825	112543	**1.55%**	108296	**2.3%**

static analyzer at the start of every region. Then, the application overall WCET is computed in an *ad hoc* manner according to the synchronization pattern between code regions. This turned out to be very easy for the considered applications, that have simple and regular communications, never more complex that the ones illustrated in Figure 1.

The estimated WCETs are given in Table 1, for 10 iterations of the main control loop on each core. The WCETs produced by our integrated approach are always tighter than using the decoupled method (21% in average on the three case studies). The gain varies depending on the amount of reuse between successive regions assigned to the same core. When the amount of reuse is high, like in application *adpcm*, that features intensive code and data reuse between code regions, the gain is significant. When the amount of reuse is smaller like in *filter* (no reuse of data, modest reuse of code between regions), the gain is much smaller.

Comparison with observed execution times. The pessimism of our WCET evaluation method is evaluated by comparing estimated WCETs with observed execution times, obtained using the SoCLib simulation software. Regarding simulation results, due to time constraints, we made no attempt to identify the worst-case input data and execute the code with typical input data, not necessarily representative of the worst-case situation. The estimated pessimism is thus an upper bound of the method pessimism. Results are reported in Table 1. The numbers show that even without executing the code using its worst-case input data, the results are encouraging: estimated and measured execution times are close to each other (of 7.7% in average). Further experiments need to be conducted to identify the actual overestimation and not only an upper bound of the overestimation.

6 Related work

Much research effort has been spent in the past in estimating WCETs of sequential code and WCRTs of multi-task applications. Research on WCET estimation has mainly targeted software running on single-core architectures (see [4] for a survey). A lot of effort has been put on *hardware-level* analysis, allowing architectures with caches and in-order pipelines to be analyzed precisely. The research presented in this paper is *not* a new WCET estimation technique, but rather takes benefit of state-of-the-art low level analysis to produce tight WCET estimates of parallel applications.

Many WCRT estimation methods compute end-to-end response times of distributed applications communicating using message passing, or multiprocessor systems (e.g. [11]). To our best knowledge, all these methods can be qualified as *decoupled*, in the sense that they use as input WCET estimations of code regions computed before the WCRT analysis. By comparison, we have shown that an *integrated* analysis allows to produce tighter WCRTs than a decoupled approach, because it allows hardware effects between code regions to be captured accurately.

The research we found to be closer to our approach is described in [3, 13, 10]. Paper [3] is devoted to WCET estimation of a parallel application running on a predictable multi-core architecture. Similarly to our work, emphasis is put on predictability of the hardware and software architectures. However, in contrast to [3] that provides formulas to combine WCETs of code snippets to obtain the WCET of the parallel application, in our work the application running on each core is analyzed as a whole. As a consequence, we are able to exploit knowledge of the hardware state between code snippets and thus can provide tighter estimates, especially for fine-grain parallelism.

In [13], a method to determine residual cache states after the execution of sequential code on a mono-core platform is provided. The method allows to obtain tighter WCETs in case of repetitive executions of the analyzed code. Using our method, we obtain the same benefits, but without needing a specific analysis and tool. This benefit comes as a side product of our method because the WCET computation of the parallel application is integrated into a WCET estimation tool that originally was analyzing sequential code.

Paper [10] proposes an ILP formulation for WCRT computation of task graphs running on multi-core systems. The method computes the application WCRT given a task-to-core mapping, architecture and scheduling policy, with contentions when accessing shared resources. Unlike [10], we currently rule out resource contentions, that is left for future work. However, contrary to [10], our analysis of the hardware is expected to be tighter because of our *integrated* approach.

7 Conclusion

We have presented in this paper a method to compute the WCETs of parallel applications running on multicore platforms. Thanks to small modifications of a WCET computation method, the parallel application can be analyzed as a whole, such that hardware effects across code regions of the application are dealt with naturally. We have demonstrated that our approach produces WCETs that are tighter than using a classical method by 21% in average. Preliminary experiments show that the WCET over-approximation is below 7.7% in average. We believe that our method can be integrated easily in other WCET estimation tools using the implicit path enumeration techniques to the extent that the analysis framework is sufficiently modular (hardware-level analysis and WCET computation are clearly separated). The proximity of our test architecture to existing commercial many-core architectures also suggests that our results are easily transposable to them.

In this paper, assumptions have been made regarding the software structure in order to demonstrate the validity of our approach on simple but yet realistic setting. In future work, our first objective will be to relax as much as possible these assumptions to broaden the scope of application of the approach. Another area for future research will be to use obtained WCETs to refine the structure of the parallel application (mapping of code regions on the cores, execution order). Finally, scalability to a larger number of cores is another area for future work.

References

1 A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *ECRTS*, pages 37 –44, July 2001.

2 M. Djemal, F. Pêcheux, D. Potop-Butucaru, R. de Simone, F. Wajsbürt, and Z. Zhang. Programmable routers for efficient mapping of applications onto NoC-based MPSoCs. In *DASIP*, 2012.

3 C. Rochange et al. WCET analysis of a parallel 3D multigrid solver executed on the MERASA multi-core. In *WCET workshop*, 2010.

4 R. Wilhelm et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM TECS*, 7(3):36:1–36:53, May 2008.

5 K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5):414–421, 2005.

6 M.I. Gordon, W. Thies, M. Karczmarek, J. Lin, A.S. Meli, A.A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S.P. Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS-X*, 2002.

7 T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives. In *MEMOCODE*, 2003.

8 J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In *WCET workshop*, 2010.

9 D. Hardy and I. Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*, 2008.

10 J. Kim, H. Oh, H. Ha, S. Kang, J. Choi, and S. Ha. An ILP-based worst-case performance analysis technique for distributed real-time embedded systems. In *RTSS*, 2012.

11 M. Kuo, R. Sinha, and P. Roop. Efficient WCRT analysis of synchronous programs using reachability. In *Proceedings DAC'11*, San Diego, CA, USA, 2011.

12 The MPPA256 many-core architecture. Online `http://www.kalray.eu/products/mppa-manycore/mppa-256/`, 2012.

13 F. Nemer, H. Cassé, P. Sainrat, and J.P. Bahsoun. Inter-task WCET computation for a-way instruction caches. In *SIES*, 2008.

14 P. Puschner, R. Kirner, and R. Pettit. Towards composable timing for real-time programs. In *STFSSD'09*, 2009.

15 J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *RTSJ*, 37(2), 2007.

16 SoCLib: an open platform for virtual prototyping of multi-processors system on chip, 2011. Online at: http://www.soclib.fr.

17 The TilePro64 many-core architecture. Online `http://www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_PB019_v4.pdf`, 2008.