Minimizing the Cost of Synchronisations in the WCET of Real-Time Parallel Programs

Haluk Ozaktas, Christine Rochange, Pascal Sainrat

Minimizing the Cost of Synchronisations in the WCET of Real-Time Parallel Programs

Haluk Ozaktas, Christine Rochange and Pascal Sainrat
IRIT
University of Toulouse, France
{ozaktas,rochange,sainrat}@irit.fr

ABSTRACT

Designing time-predictable architectures to support the requirements of hard real-time systems is the goal of several research projects. In this paper we assume that such platforms exist and we focus on the timing analysis of parallel real-time applications. One of the main challenges is to determine how much the delays induced by software constructs such as synchronisations can impact the worst-case execution times (WCETs) of parallel threads. In this paper, we refine state-of-the-art analysis: first, we derive more accurate estimations of stalls at critical sections; second, we introduce new locking primitives that minimise stall times on the worst-case path. Experimental results show noticeable improvements on the WCETs of benchmarks.

Keywords

Real-time, WCET, parallel programming, lock

1. INTRODUCTION

In the next years, real-time embedded systems will be built on multicore platforms which provide better energy efficiency than current single-core architectures. But such multicore platforms challenge timing analysis techniques because the sharing of hardware resources among cores generates conflicts that impair the timing predictability [13, 22].

This issue is tackled by several research projects, e.g. PREDATOR [4], parMERASA [27] and T-CREST, that aim at designing time-predictable multicore architectures. Promising solutions have already been proposed, e.g. in [19, 18]. We assume that such time-predictable platforms are available and we look forward to the next step which will be the parallelisation of applications to improve the tasks response time and not only the task throughput. This will be needed to achieve the high performance requirements that are foreseen to get support for better safety, lower emissions and improved comfort for passengers in aircrafts or cars.

Parallel threads experience delays due to the use of synchronisation primitives, which challenges the timing analysis. For example, a thread reaching a barrier might be stalled until other threads reach the same barrier. Similarly, a thread trying to acquire a lock before a critical section might be stalled until concurrent threads get the lock themselves, execute the critical section and then release the lock.

In [17], we proposed an approach to determine the WCET of a parallel application in which threads synchronise through the shared memory, using POSIX-like primitives. This approach first computes the worst-case stall times (denoted by $WCST$) generated by synchronisations, then integrates them in the global WCET analysis. In Section 2.1, we will review this algorithm to estimate worst-case delays due to inter-thread conflicts. We will then report, in Section 2.2, some experimental results which show that critical sections, already known to limit the average speedup of parallel software, degrade even more noticeably the worst-case performance. This is partly due to considering pessimistic assumptions (maximum contention) in order to obtain safe upper bounds. We will discuss this pessimism in Section 2.3.

Our contribution in this paper is twofold:

a) We propose a refined analysis of the impact of stall times at critical sections on the WCET when considering a sequence of critical sections (either distinct critical sections executed in a row, in Section 3.1, or the same section executed in a loop, in Section 3.2). Such sequences are found in various common parallel programming patterns such as dynamic work assignment, producer-consumer scheme or iteration until convergence. Experimental results (Section 3.3) show that sensibly tighter WCET estimates can be achieved.

b) We introduce new locking primitives that enforce a constant order on threads crossing repeatedly the same critical section (Section 4.1). They collectively implement our Constant- Order Lock Granting (CO-LoG) policy. Using this policy drastically reduces the impact of synchronisation stalls on the WCET of parallel programs, as shown in Section 4.2.

Related work on static WCET analysis, time-predictable synchronisations and the timing analysis of parallel real-time programs is surveyed in Section 5. We conclude the paper in Section 6.

2. MOTIVATION

2.1 Background

We have proposed an approach to the timing analysis of real-time shared-memory parallel programs in [17]. Considering a time-predictable multicore architecture, our solution aimed at estimating the worst-case stall times (WCSTs) of concurrent threads at synchronisation points, i.e. where threads call a `barrier_wait()` or a `mutex_lock()` primitive.

As far as critical sections (mutexes) are concerned, this approach is conservative: it assumes that a thread trying to acquire the lock is stalled, in the worst case, by any other thread that also requests the lock at some point in the program. To allow the stall time to be upper bounded, we consider time-predictable mutexes, as proposed in [9], which guarantee that threads are granted the lock on a first-come first-served basis. With this assumption, the WCST at a critical section is estimated as:

$$WCST \leq (T-1) \cdot C$$

where T is the total number of threads using the lock and C is the worst-case execution time spent in the critical section. If threads do not execute all the same code in the critical section, the longest path is considered.

In other words, the cost of any critical section is multiplied by the number of concurrent threads because a pessimistic situation, with full contention, is assumed. The impact on the WCET of the parallel application can be sensible if either the number of threads is large or the critical section is long or the critical section is executed several times in a loop. This is confirmed by experimental results reported in Section 2.2. In Section 2.3, we explain why this estimation of WCETs is pessimistic and can be refined.

2.2 Impact of Critical Sections on Worst-Case Execution Times

Critical sections are known to be a performance bottleneck in parallel programs: since a single thread at a time can execute in a critical section, it may happen that other threads be stalled until the critical section is free. Depending on the respective sizes of parallel parts of code and critical sections, such conflicts and sequentialisation might severely degrade performance. In this section, we report experimental results that show how much the worst-case performance is impacted by stall times at critical sections.

2.2.1 Methodology

All the experimental results reported in this paper have been collected considering the following architectural model and benchmark applications.

Multicore Model.
In order to isolate the effects of synchronisation stall times, we assume a 64-core[1] architecture built up on very simple cores. Each core benefits from private 32-KB separate instruction and data caches, and executes any instruction that hits in the cache in a single cycle. The latency to the main memory is 649 cycles, which includes the impact of maximum possible contention among cores, assuming a time-predictable interconnection network and memory system, such as the one proposed in [18].

[1]In practise, so many cores in a shared-memory architecture may raise side issues which are out of the scope of this paper.

Hardware-based data coherency management is challenging for timing analysis because its behaviour is strongly related to the actual interleaving of accesses to shared data by the different threads. Some solutions towards time-predictable data-coherent caches have been proposed, e.g. the ODC^2 cache in [21], but they introduce extra memory accesses that impact the WCET. To focus on synchronisation costs, we assume that *shared data* are *not cached* and reside in the main memory only.

Benchmark Programs.
We consider three parallel applications. Each of them is composed of a main thread that creates child threads. All threads (including the main one) execute the same computation code. Critical sections are implemented using time-predictable lock primitives, e.g. *ticket locks*, that guarantee that threads are served in a first-come first-served basis. This way, a thread competing with $T-1$ other threads cannot be stalled more than once by each other thread when it requests access to a critical section.

`jacobi` (see Figure 1) solves a system of linear equations using the iterative Jacobi method on a 2D matrix. Each thread computes part of the points, independently of the other threads (as in the so-called *embarrassingly parallel* computation model). The process is repeated *until convergence is reached*, i.e. until the average variation of results from one iteration to the other one is lower than a specified threshold. Each thread computes its local variation (*line 9*) then adds it to the global variation in a critical section (*line 12*). A synchronisation barrier is used to ensure that threads do not start their computation before the global variation is initialised (*line 5*) and do no check the convergence condition before the global variation has been computed (*line 14*). The last barrier in the loop (*line 18*) ensures that decision on convergence is made before global variation is reset to zero. Since the implementation of the `barrier_wait` primitive includes a critical section, the synchronisation pattern of this application repeats the sequence shown in Figure 2: between two progress synchronisations, threads compete for two critical sections in a row.

```
 1: function JACOBI
 2:     done ← 0
 3:     while !done do
 4:         var ← 0
 5:         BARRIER_WAIT(&bar, N)
 6:         local_var ← 0
 7:         for each local point do
 8:             compute new value
 9:             add variation to local_var
10:         end for
11:         MUTEX_LOCK(&mutex)
12:         var ← var + local_var
13:         MUTEX_UNLOCK(&mutex)
14:         BARRIER_WAIT(&bar, N)
15:         if var ≤ threshold then
16:             done ← 1
17:         end if
18:         BARRIER_WAIT(&bar, N)
19:     end while
20: end function
```

Figure 1: Pseudo-code of the `jacobi` benchmark

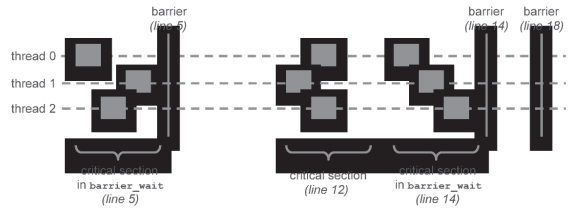

Figure 2: Synchronisation pattern in the `jacobi` benchmark

```
 1: function DTA
 2:     while element pool is not empty do
 3:         MUTEX_LOCK(&mutex)
 4:         get next element to compute
 5:         MUTEX_UNLOCK(&mutex)
 6:         if got an element to compute then
 7:             compute element
 8:         end if
 9:     end while
10: end function
```

Figure 3: Pseudo-code of the `dta` benchmark

`dta` (see Figure 3) implements *dynamic task allocation*, a technique that aims at balancing the load among threads. Each thread fetches a new element from the pool (this is done in a critical section, *line 4*) and computes it. Thus threads compete repeatedly for the same critical section within the loop.

`hist` (shown in Figure 4) generates the histogram of an image. Rows of the image are equally distributed among threads. To limit contention on the global histogram, threads keep a local histogram for each row, which they accumulate into the global histogram when the row has been completely processed. This way, the global histogram is updated regularly, and this reflects the behaviour of applications that need to share partial or temporary common results. Naturally, updating the global history is protected by a critical section (*line 8*). The synchronisation pattern is then similar to that of `dta`, i.e. a critical section repeated in a loop.

For each benchmark, we also considered a sequential version used as a reference. The speedup of a parallel version is defined as the ratio between the respective WCETs of the sequential and parallel versions:

$$speedup = \frac{WCET_{sequential}}{WCET_{parallel}}$$

Timing Analysis.
All the results presented in the remainder of the paper are WCET estimates obtained using OTAWA, a toolset dedicated to static WCET analysis [2]. The individual WCETs of basic blocks are augmented with the stall times at synchronisations, as suggested in [17].

2.2.2 Experimental Results

Table 1 shows the speedups of 2-thread to 64-thread versions of the three benchmarks. Threads are statically mapped and scheduled on cores, one per core. For 2- to 32-thread

```
 1: function HIST
 2:     for each row do
 3:         initialise local histogram
 4:         for each pixel in the row do
 5:             update local histogram
 6:         end for
 7:         MUTEX_LOCK(&mutex)
 8:         update global histogram from local histogram
 9:         MUTEX_UNLOCK(&mutex)
10:     end for
11: end function
```

Figure 4: Pseudo-code of the `hist` benchmark

versions, the timing analysis assumes that independent applications run concurrently on the cores that are not used by the benchmark. Then contention on the interconnection network and in the memory system is considered maximum for all the experiments, which leads to safe WCET estimates.

Table 1: Speedups

# of threads	`jacobi`	`dta`	`hist`
2	1.74	1.94	1.70
4	3.44	3.76	3.13
8	6.56	7.12	5.40
16	11.07	12.84	8.49
32	13.70	21.46	11.89
64	10.89	32.31	14.87

The three benchmarks, particularly `jacobi` and `dta`, exhibit high speedups up to 16 threads. This is due to threads performing independent computations, even if they have to synchronise at some points to take common decision or to share work. However, speedups are moderate with more than 32 threads (having 64 threads for `jacobi` even achieves a smaller speedup than having 16 threads). The per-thread efficiency, i.e. the ratio of the speedup over the number of threads, noticeably decreases when the number of threads increases. This suggests that the cost of stall times at synchronisations takes an important share of the WCET for medium-scale parallel versions. This is confirmed by Table 2 that shows the part of estimated WCETs that is due to stall times. For the 16-thread version of `hist`, more than half of the WCET is spent on waiting at critical sections. For 64-thread versions of `jacobi` and `hist`, the waiting time represents 3/4 of the total WCET. This shows how important it is to reduce the impact of stall times at critical sections on the WCET, either by restructuring the application or by improving the analysis.

Table 2: Impact of critical sections on the WCET

# of threads	`jacobi`	`dta`	`hist`
2	0.20%	1.48%	4.27%
4	1.18%	4.31%	11.80%
8	5.26%	9.52%	23.79%
16	19.05%	18.40%	40.08%
32	48.70%	31.79%	58.00%
64	78.63%	48.66%	73.69%

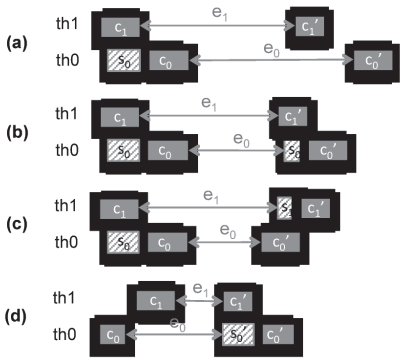

Figure 5: Possible contention among two threads executing two critical sections in a row

2.3 Pessimism of the Existing Approach

The results presented above demonstrate the importance of estimating the worst-case impact of stall times at synchronisations as tightly as possible. In [17], the execution time (t) of any piece of code breaks down into pure running time (r) and stall time (s): $t = r + s$. The WCET of this piece of code is then computed as:

$$WCET = WCRT + WCST$$

where $WCRT = max(r)$ and $WCST = max(s)$. In contrast, we compute the WCET as $WCET = max(r + t)$, considering that the maximum stall time experienced by one thread depends on the running times of all the concurrent threads. Now, as seen in Section 2.2, the approach to estimate the WCET of a parallel application extends the method used for sequential programs by adding stall times to the individual WCETs of sub-paths in the code. To remain consistent with this approach, we define the *stall time in the worst case* as:

$$ST_{WC} = max(r + t) - max(r)$$

We then have $WCET = WCRT + ST_{WC}$.

Besides, our intuition is that consistently assuming maximum contention for the longest thread is overwhelmingly pessimistic. This intuition is based on the following observations.

Sequence of Critical Sections.
Let us consider two threads, th0 and th1, that execute the same code, which contains two critical sections in a row, CS and CS' (protected by two different locks). Figure 5 shows four possible situations where th0 is stalled or not by th1 before entering a critical section. In this picture, s_i and s'_i are the effective stall times of th*i* at the critical sections, c_i and c'_i are its execution times within the critical sections, and e_i is its execution time between the critical sections. The worst-case execution times in the critical sections are denoted by C and C', respectively, the worst-case execution time between the critical sections is denoted by E.

Let us assume that th0 is the longest one, that leads the WCET of the whole parallel program. The execution time of the sequence for th0 is computed as

$$t_0 = s_0 + c_0 + e_0 + s'_0 + c'_0$$

and its WCET is determined considering the maximum value of t_0.
In situations (a) and (c), th0 is not stalled by th1 at CS' ($s'_0 = 0$). In addition, $s_0 \leq c_1$. The maximum value of t_0 is then:

$$WCET_{(a,c)} = max(c_1 + c_0 + e_0 + c'_0) = 2 \cdot C + E + C'$$

In situation (b), th0 is stalled by th1 and we get:

$$c_0 + e_0 + s'_0 = e_1 + c'_1$$

We also have $s_0 \leq c_1$. Then:

$$WCET_{(b)} = max(c_1 + e_1 + c'_1 + c'_0) = C + E + 2 \cdot C'$$

In situation (d) we have $s_0 = 0$ and $s'_0 \leq c'_1$. Then:

$$WCET_{(d)} = max(c_0 + e_0 + c'_1 + c'_0) = C + E + 2 \cdot C'$$

To summarise, the WCET on this sequence for th0 is:

$$WCET = max((2 \cdot C + E + C'), (C + E + 2 \cdot C'))$$

or

$$WCET = C + E + C' + max(C, C')$$

and

$$ST_{WC} = max(C, C')$$

which is lower than the value computed according to [17] ($C + C'$).

In Section 3.1, we will show how this result can be extended to a larger number of competing threads.

Critical Section in a Loop.
Let us now consider two threads that execute the same code with a loop containing a critical section. If we consider two successive iterations of the loop for each of the threads (e.g. iterations i and $i + 1$ for th0 and iterations j and $j + 1$ for th1, we get the same behaviour as above (except for we have the same critical section repeated instead of two different critical sections). The same reasoning can be used to show that the maximum stall time (for th0) *over two successive iterations of each thread* is only C (WCET in the critical section) instead of $2 \cdot C$. However, it can happen that one thread executes the loop body faster than the other one and that threads be in conflict in their iterations, e.g. (i and j) and ($i+1$ and $j+2$). In other words, the threads cannot be in conflict in *all* the loop iterations. This is illustrated with the experiment reported below.

We have considered an example code where three threads execute (in parallel) the same loop that has at most eight iterations. The body of the loop breaks down into a computation part (executed in at most two time units) and a critical section (executed in at most one time unit). Figure 6 shows the worst-case scenario which we have found by an exhaustive search. In this scenario, the maximum stall time for the longest thread (th0) is only $ST_{WC} = 15t.u.$ while the formula proposed in [17] would have given:

$$WCST = N \cdot (T - 1) \cdot C = 8 \times 3 \times 1 = 24\ t.u.$$

In next section, we will suggest and prove new formulas that estimate the stall times accurately.

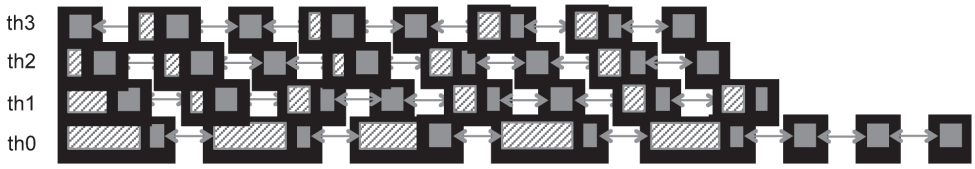

Figure 6: Worst-case scenario for an example code. Dark-grey blocks show the time spent in the critical section and stripped blocks show stall times. Arrows show the time spent in computation parts.

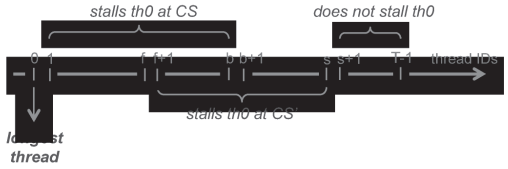

Figure 7: Stalling threads in a sequence of two critical sections in a row.

3. TIGHT ANALYSIS OF STALL TIMES AT CRITICAL SECTIONS

3.1 Sequence of Critical Sections

We first focus on a sequence of two critical sections guarded by distinct locks and executed by several concurrent threads, running simultaneously each on a different core. Thread th0 is the one that determines the WCET of the parallel application (the longest thread) and we want to compute the maximum time it can spend on this sequence of code.

Figure 7 shows that the $(T-1)$ threads that compete with th0 can be divided into four sets: those that never stall it (IDs from $s+1$ to $T-1$), those that stall it at both critical sections (IDs between $f+1$ and b) and those that stall it at the first or second critical section only.

The execution time of th0 on this code can be computed as shown in Section 2.3:

$$t_0 = s_0 + c_0 + e_0 + s'_0 + c'_0$$

and its WCET is:

$$WCET = max(s_0 + s'_0) + C + E + C'$$

We can write: $s_0 = s_{0(f)} + s_{0(b)}$ where $s_{0(f)}$ (resp. $s_{0(b)}$) denotes the stall time at CS due to threads that stall th0 at CS only (resp. both at CS and CS').
Similarly, $s'_0 = s'_{0(b)} + s'_{0(s)}$.

We have:

$$s_{0(f)} \leq \sum_{i=1}^{f} c_i \leq f \cdot C$$

and

$$s'_{0(s)} \leq \sum_{i=b+1}^{s} c_i \leq (s-b) \cdot C'$$

From Section 2.3, we know that the impact on the WCET of a thread that stalls th0 at both critical sections is limited to $max(C, C')$. Then

$$s_{0(b)} + s'_{0(b)} \leq \sum_{i=f+1}^{b} c_i \leq (b-f) \cdot max(C, C')$$

This leads to:

$$ST_{WC} = max(s_0 + s'_0) = f \cdot C + (s-b) \cdot C' + (b-f) \cdot max(C, C')$$

and

$$ST_{WC} \leq s \cdot max(C, C')$$

Since $s \leq (T-1)$, we can write:

$$ST_{WC} \leq (T-1) \cdot max(C, C') \quad (1)$$

3.2 Critical Section in a Loop

In this section, we focus on the following situation that is frequently encountered in parallel programs: T threads run the same code including a loop that iterates at most N times, and the loop body includes some computations and a critical section (and no progress synchronisation, such as barriers). In the following, E and C denote the worst-case execution time of the computation part and the critical section, respectively.

According to the formulas derived in [17], the total stall time (over all the loop iterations) for this critical section (which is to be taken into account when computing the WCET of the program) would be upper bounded by:

$$WCST = N \cdot (T-1) \cdot C \quad (2)$$

This amounts to considering that the longest thread is stalled by every other thread each time it reaches the critical section.

As suggested in Section 2.3, this is a pessimistic upper bound. We are going to show that it can be lowered to:

$$ST_{WC} \leq \left\lceil \frac{N}{2} \right\rceil \cdot (T-1) \cdot C \quad (3)$$

when $E \geq (T-1) \cdot C$, or to:

$$ST_{WC} \leq (T-1) \cdot C + \left\lceil \frac{(N-1) \cdot (T-1)^2 \cdot C^2}{(T-1) \cdot C + E} \right\rceil \quad (4)$$

when $E < (T-1) \cdot C$.

When $E \ll (T-1) \cdot C$, this bound is close to the pessimistic one (Equation 2) and no improvement is achieved. But when the computation part increases, the bound on ST_{WC} decreases. When E is equal to $(T-1) \cdot C$, the bound is as low as $\lceil \frac{N+1}{2} \cdot (T-1) \cdot C \rceil$, i.e. half of the pessimistic bound. Proofs are provided in the remainder of this section.

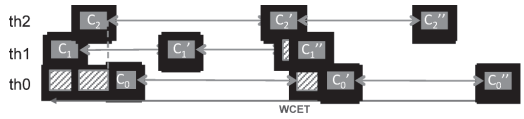

Figure 8: Critical section in a loop

Let us first consider the situation where $E \geq (T-1) \cdot C$. Let us assume that thread th$x$ stalls th0 while they are executing iteration i_x and i_0, respectively, of the loop. When th0 executes iteration $i_0 + 1$, it cannot be stalled by th$x$ executing iteration $i_x + 1$. More precisely, the stall time over these two iterations cannot exceed C, as shown in the previous section. However, th0 in iteration $i_0 + 1$ could be stalled by th$x$ in iteration $i_x + 2$ (or in a later iteration). But then th$x$ has (at least) one iteration less to execute than th0. This is illustrated in Figure 8. The result is that any stall of th0 by th$x$ removes the risk for one other possible conflict. Finally, th$x$ can stall th0 at only half of the loop iterations. □

Let us now consider the case where $E < (T-1) \cdot C$. The execution of the longest thread can be broken down into two phases, which can be observed in Figure 6.
In the *first phase*, the longest thread (th0) runs concurrently to $(T-1)$ other threads that execute all their critical sections (i.e. at most $N \cdot (T-1)$ critical sections). In the same time, the longest thread executes its first n critical sections. In this phase, there must be one thread in the critical section at each cycle because each thread finishes its computation part before all the other threads could enter the critical section (this is because we consider $E \leq (T-1) \cdot C$). The length of the first phase is:

$$WCET_1 = N \cdot (T-1) \cdot C + n \cdot C$$

In the *second phase*, only the longest thread (th0) has some critical sections left to execute. The length of this phase is given by:

$$WCET_2 = (N-n) \cdot C + (N-n+1) \cdot E$$

The total WCET of the sequence (equal to the WCET of the longest thread) is then $WCET = WCET_1 + WCET_2$ or

$$WCET = N \cdot T \cdot C + (N-n+1) \cdot E$$

The stall time in the worst case is:

$$ST_{WC} = WCET - N \cdot (C+E)$$

where $N \cdot (C+E)$ is the total execution time without stalls. Then:

$$ST_{WC} = N \cdot (T-1) \cdot C + (1-n) \cdot E \quad (5)$$

This stall time is observed during the first phase and is distributed among the n executions of the critical section by the longest thread in that phase. Since we consider a fair locking primitive, we can state that each stall cannot exceed $(T-1) \cdot C$. Then we can also write :

$$ST_{WC} \leq n \cdot (T-1).C \quad (6)$$

Combining Equations 5 and 6, we get:

$$N \cdot (T-1) \cdot C + (1-n) \cdot E \leq n \cdot (T-1) \cdot C$$

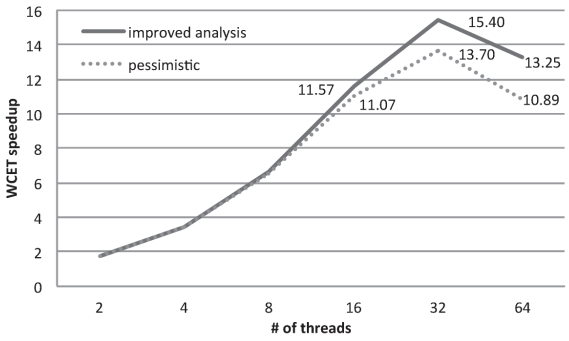

Figure 9: Speedup of `jacobi`

then

$$N \cdot (T-1) \cdot C + E \leq n \cdot ((T-1) \cdot C + E)$$

As a result,

$$n \geq \frac{N \cdot (T-1) \cdot C + E}{(T-1) \cdot C + E}$$

By considering this lower bound on n in Equation 5, we get Equation 4. □

3.3 Experimental Results

Sequence of Two Critical Sections.
We analysed the benefits or our improved analysis considering the `jacobi` benchmark (see Figures 1 and 2). This program contains a sequence of two critical sections surrounded by two barriers, and repeated in a loop, as well as a third barrier at the end of the loop body. The second critical section is part of the `barrier_wait` primitive and has a worst-case execution time (denoted by C_2) that is about 9% longer than the WCET of the first critical section (C_1) used to protect the shared variable. Then, the stall time in the worst case for each iteration of the loop is given by $ST_{WC} = (T-1).C_2$ (cf. Equation 1).

The reduction on ST_{WC} for these two critical sections is then given by $\frac{C_1}{C_1+C_2} = \frac{1}{1+1.09}$, i.e. about 48%, regardless of the number of threads. Now, taking into account the other two critical sections contained in the first and the last `barrier_wait` primitives, the global reduction is 22.66%.

However, since the part of stall times in the total WCET is limited up to 8 threads (see Table 2), the improvement on the speedup, shown in Figure 9, is only significant when 16 or more threads are used. For the 32-thread version, which is the sweet-spot for this benchmark, the impact of the ST_{WC} on the WCET is decreased from 48.70% to 42.34% and the speedup is improved from 13.70 to 15.40, i.e. by 12.4%. It is also worth noting that our analysis always yields to better results than the pessimistic analysis of [17].

Critical Section in a Loop.
We evaluated our improved analysis considering the `dta` and `hist` benchmarks introduced in Section 2.2. In both benchmarks, threads concurrently execute a loop that contains a single critical section (cf. Fig. 3 and Fig. 4).

Table 3: Impact of critical sections on the WCET using improved analysis.

# of threads	jacobi	dta	hist
2	0.15%	0.75%	2.18%
4	0.92%	2.21%	6.27%
8	4.12%	5.00%	13.50%
16	15.39%	10.13%	25.06%
32	42.34%	18.90%	45.47%
64	74.00%	32.15%	68.08%

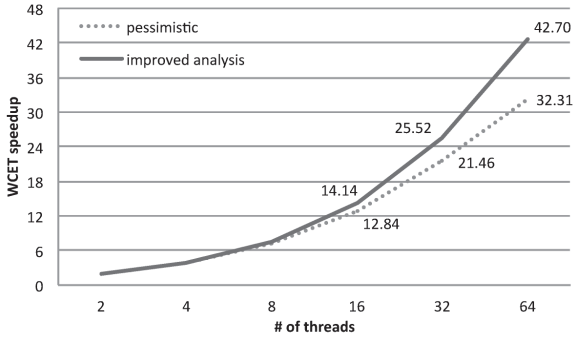

Figure 10: Speedup of `dta`

For `dta`, $(T-1)\cdot C$ is always smaller than the WCET of the computational part (E), regardless of the number of threads. As a result, the ST_{WC} is halved by the improved analysis as explained in Section 3.2.

For `hist`, $(T-1)\cdot C$ is smaller than E for $T \leq 16$, and the ST_{WC} is also halved by the improved analysis. But for $T \geq 32$, the total cost of $T-1$ critical sections becomes greater than (E). In this case, Equation 5 must be used and it yields to a value of ST_{WC} that is about 60.4% of the pessimistic stall time for 32 threads and about 76.2% of the pessimistic stall time for 64 threads.

The share of the stall times computed with our improved analysis in the final WCET is shown in Table 3.

Figure 10 plots the speedups obtained with both the pessimistic and improved analyses for `dta`. The benefit is significant for 16 threads or more, and the speedup increases from 32.31 to 42.70, i.e. by 32.2%, for the 64-thread version. The reason is that the impact of stall times in critical sections is so small for up to 8 threads (cf. Table 2) that optimising their analysis can only slightly increase the overall speedup. But again, with 16 or more threads, the stall times at critical sections start to take an important part on the WCET (more than about 18%) and refining their estimation becomes important.

Figure 11 shows the speedups for the `hist` benchmark. Contrarily to `jacobi` and `dta`, `hist` benefits from a significant improvement starting from 8 threads. With 64 threads, the speedup jumps from 14.87 to 18.04.

To summarise, when the stall times represent more than about 15% of the total WCET, our improved analysis yields to significant enhancement on the speedup, varying from 4.5% up to 32.2%. As Table 2 shows, the part of the stall

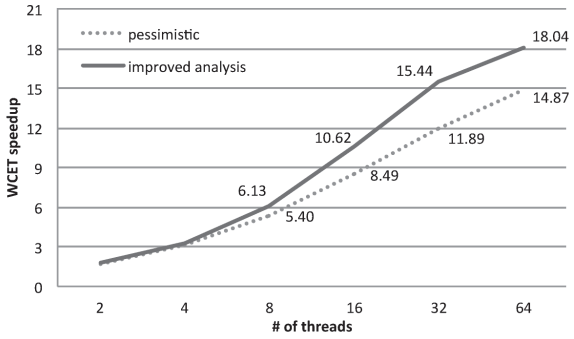

Figure 11: Speedup of `hist`

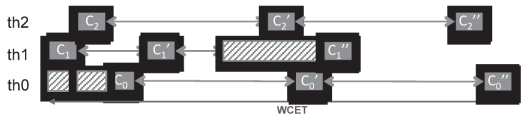

Figure 12: Critical section in a loop, with the CO-LoG primitive

times at critical sections exponentially increases with the number of threads. Thus, for widely parallel applications running on future manycores, our improved analysis can be a key strategy to increase speedups.

4. NEW LOCKING PRIMITIVES FOR TIGHT WCST ANALYSIS

4.1 The CO-LoG Policy

We consider threads executing the same code, i.e. with the same maximum distance E between two successive calls to the locking primitive. The stall time ST_{WC} at a critical section in a loop would be reduced to its minimum value if threads were guaranteed to get the lock in the same order at each iteration of the loop. This is illustrated in Figure 12.

Note that such a behaviour is *not* enforced by well-known ticket locks. A ticket lock keeps an ordered list of requesters and grants the lock in a fair way, i.e. in the order of the list. However, it does not imply that this order remains identical over successive invocations. This is the reason why we need to introduce a set of mutex lock handling primitives that implement our so-called Constant-Order Lock Granting (CO-LoG) policy. This includes: (a) two primitives, `mutex_get_in` and `mutex_get_out` that allow a thread to declare that it will start (resp. stop) competing for the lock; (b) modified `mutex_lock` and `mutex_unlock` primitives that consider an ordered list of competing threads. The pseudo-code of these routines is given in Figure 13, while the pseudo-code of the original routines (ticket lock) is shown in Figure 14 for reference.

The CO-LoG policy requires maintaining a list of the threads competing for the lock. For better performance, the list is implemented as two statically allocated arrays, `prev` and `next` (we assume that the maximum number of

```
 1: function MUTEX_GET_IN(mutex_t *mutex)
 2:     SPINLOCK_LOCK(mutex→guard)
 3:     my_ticket = mutex→next_ticket++
 4:     next[prev[mutex→now_serving]] = my_ticket
 5:     prev[my_ticket] = prev[mutex→now_serving]
 6:     next[my_ticket] = mutex→now_serving
 7:     prev[mutex→now_serving] = my_ticket
 8:     SPINLOCK_UNLOCK(mutex→guard)
 9: end function
10:
11: function MUTEX_GET_OUT(mutex_t *mutex)
12:     SPINLOCK_LOCK(mutex→guard)
13:     next[prev[my_ticket]] = next[my_ticket]
14:     prev[next[my_ticket]] = prev[my_ticket]
15:     SPINLOCK_UNLOCK(mutex→guard)
16: end function
17:
18: function MUTEX_LOCK(mutex_t *mutex)
19:     while mutex→now_serving != my_ticket do
20:     end while
21: end function
22:
23: function MUTEX_UNLOCK(mutex_t *mutex)
24:     SPINLOCK_LOCK(mutex→guard)
25:     mutex→now_serving = next[mutex→now_serving]
26:     SPINLOCK_UNLOCK(mutex→guard)
27: end function
```

Figure 13: New mutex primitives implementing the CO-LoG policy

threads is fixed). In `mutex_get_in`, a thread first gets a ticket (*line 3*), i.e. a unique integer which is then used to index the arrays. Then the thread is appended to the list (*lines 4-7*). Note that these operations must be performed within a critical section, protected by a spin lock. Naturally, the stall time at this critical section must be accounted for when estimating WCETs but (a) its value is small because the critical section is very short, and (b) the function will generally not be invoked in a loop. To get the lock, a thread executes the `mutex_lock` routine and waits until its ticket is selected (*line 19*). Once it terminates the execution of a critical section, it invokes `mutex_unlock` that grants the lock to the thread holding the next ticket in the list (*line 25*). Again, this is protected by a spin lock that must be taken into account when computing WCETs. In `mutex_get_out`, the ticket of the thread is removed from the list. This way, the thread will not be indefinitely waited for by other threads trying to get the lock.

Figure 15 provides an example of how the new primitives can be used in an application code. Note that it is absolutely needed that the thread call `mutex_get_out` when it exits the loop, otherwise there would be a risk of deadlock.

The CO-LoG policy, which ensures the lock is granted always in the same order, guarantees a synchronized progress of threads in loop iterations. It makes it impossible for thread `thx` to execute iteration $i+2$ while thread `th0` is still in iteration i. This removes the cause of possible stalls described in Section 3.2 for $E \geq (T-1) \cdot C$ and illustrated in Figure 8 between `th1` and `th0`. As a result, using CO-LoG policy, a thread taking the maximum possible time to execute the computation part is only stalled in the first iteration

```
 1: function MUTEX_LOCK(mutex_t *mutex)
 2:     SPINLOCK_LOCK(mutex→guard)
 3:     my_ticket = mutex→next_ticket++
 4:     SPINLOCK_UNLOCK(mutex→guard)
 5:     while mutex→now_serving != my_ticket do
 6:     end while
 7: end function
 8:
 9: function MUTEX_UNLOCK(mutex_t *mutex))
10:     mutex→now_serving++
11: end function
```

Figure 14: Original mutex primitives

```
 1: function WORK
 2:     ...
 3:     MUTEX_GET_IN(&mutex)
 4:     for ... do
 5:         ...
 6:         MUTEX_LOCK(&mutex)
 7:         critical section
 8:         MUTEX_UNLOCK(&mutex)
 9:     end for
10:     MUTEX_GET_OUT(&mutex)
11: end function
```

Figure 15: Usage of new mutex primitives

if $E \geq (T-1) \cdot C$. This yields to:

$$ST_{WC} = (T-1) \cdot C \tag{7}$$

On the other hand, if $E < (T-1) \cdot C$, the longest thread is stalled for $\big((T-1) \cdot C - E\big)$ at each iteration, except for the first one, because the lock is still being held by other threads. By considering maximum contention for the fist iteration, the maximum stall time can be formulated by:

$$ST_{WC} = (T-1) \cdot C + (N-1) \cdot ((T-1) \cdot C - E) \tag{8}$$

4.2 Experimental Results

Results for `dta`.
As previously said, $(T-1) \cdot C < E$ for `dta`. Then, when using the CO-LoG primitives, the longest thread only waits in the first iteration of the loop. This yields to a total stall time that is about 0.003% of the stall time found for the pessimistic analysis for 2 threads, and 0.1% for 64 threads. In other words, the synchronisation is almost for free.

Figure 16 plots the speedups obtained with CO-LoG and with the pessimistic analysis respectively. The speedup improvement achieved by CO-LoG is 10.5% for 8 threads and 94.6% for 64 threads. Unsurprisingly, this is significantly more than the increases obtained by the improved analysis since the stall times are much more drastically reduced by the CO-LoG policy. It should also be noted that even though only 9.52% of the total WCET is spent on waiting at critical sections for 8 threads (cf. Table 2), the CO-LoG policy still yields to a significant increase of the speedup.

Results for `hist`.
In Section 3.3, we stated that $(T-1) \cdot C < E$ when $T \leq 16$ for `hist`. In that case, and when using CO-LoG, the longest thread only waits at the critical section in the first iteration

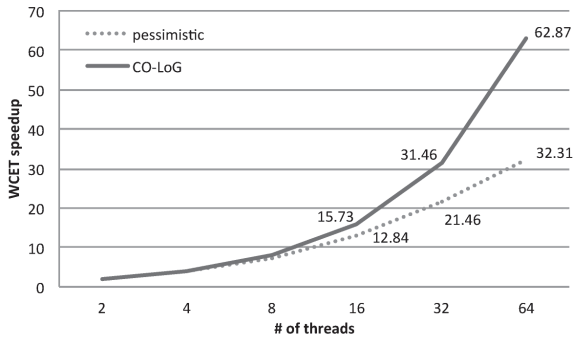

Figure 16: Speedup of `dta`

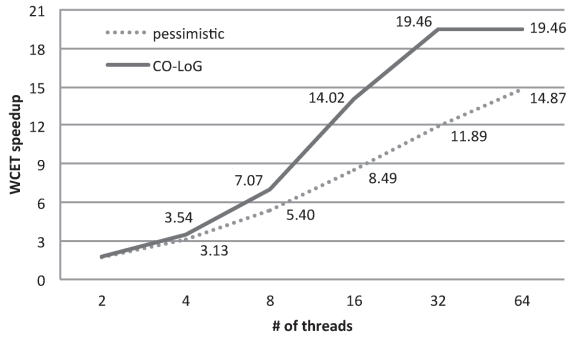

Figure 17: Speedup of `hist`

of the loop. The stall time in the worst case decreases to 0.2% of the value computed with the pessimistic analysis for 2 threads, and to 0.78% for 16 threads. This results in negligible stall times compared to the total WCET. Now, for more than 32 threads, $(T-1)\cdot C > E$ and ST_{WC} must be estimated with Equation 8. This yields to a total stall time that is about 33% of pessimistic stall time for 32 threads and about 68% of pessimistic stall time for 64 threads.

Figure 17 shows that CO-LoG achieves very high speedups up to 32 threads. For the 64-thread version, $(T-1)\cdot C$ becomes so high that the reduction in stall times degrades to 32% and the improvement of the speedup becomes less significant than for the 32-thread version. Nevertheless, CO-LoG achieves a 30.9% increase in speedup for 64 threads. This shows that even though the relative decrease in the stall times can be hurt when the number of threads increases our CO-LoG policy still yields to strong enhancements in speedups. The sweet-spot for `hist` benchmark using CO-LoG policy is the 32-thread version where the speedup is almost equal to that of the 64-thread version. In that case, the improvement on the speedup is 63.7%.

5. RELATED WORK

Impact of critical sections. Amdahl's law [1] says that the speedup of parallel applications is limited by the amount of time spent in sequential parts. Critical sections are, by definition, executed sequentially (by at most one thread at a time) and then contribute to limiting the performance of parallel programs [7]. For this reason, optimising critical sections has been the target of many research works. Some aimed at applying compiler optimisations to reduce the size of critical sections (e.g. strength reduction and code motion [3]), others introduced joint hardware/software approaches to accelerate critical section [26]. These techniques, designed for high-performance parallel programs would also improve the performance of real-time parallel application.

Time-predictable synchronisations. In a real-time system, the execution time of every piece of software must be analysable, i.e. an upper bound must be computable. This also concerns routines in the operating system code, and in particular synchronisation primitives. So-called *strong semaphores*, i.e. semaphores with linear waiting and bounded resource usage are introduced in [16] and several hardware and software implementations are proposed. In [9], time-predictable solutions for mutex locks and barriers are investigated and analysed. A hardware mechanism, called *split-phase synchronisation* is proposed in [8] to reduce the execution time of atomic instructions used to implement synchronisation primitives.

Static WCET analysis. Static WCET analysis approaches aim at deriving safe upper bounds on the execution times of tasks, considering any feasible input data set. They work on a Control-Flow Graph representation of the program and they break down into three steps. The first one, *flow analysis*, determines loop bounds [6, 5] and (in-)feasible paths [25, 14]. The second step, *low-level analysis*, computes the worst-case execution cost of every basic block considering a model of the target hardware [24, 12, 23]. Finally, the third step generates an integer linear program to determine the task WCET, as in the IPET method [15].

WCET analysis of parallel programs. In [20], parallel applications where threads communicate through message passing are considered. A joint analysis is proposed, where the analysis of worst-case communication times is integrated into the analysis of the global worst-case execution time. The approach consists in merging the control flow graphs of parallel threads, then adding edges to model the synchronisations related to sending/receiving messages. Such edges introduce dependencies among code parts of each thread (they are oriented). This approach does not seem to support shared memory synchronisations like mutual exclusion mechanisms, since such mechanisms do not allow making any assumption on the order in which concurrent threads may enter a critical section.

In [10], timed automata and model checking techniques are used to estimate the behaviour of shared-memory parallel programs running on multicore architectures. This approach considers any possible interleavings among threads at critical sections. The WCET of a parallel program is computed through a binary search, where model checking shows wether the WCET can be proved lower than a given value.

In [11], a shared-memory parallel programming language is introduced and abstract interpretation techniques are used to determine the behaviour of programs written with this language. The steps of the analysis are shown on a toy

example.

6. CONCLUSION

To reach their required level of performance, future real-time systems will be implemented on multicore architecture and their software will have to be parallelised. Even on a time-predictable multicore, parallel threads are submitted to delays due to synchronisation operations. These delays must be accounted for when computing the WCET of a parallel critical application. In a recent paper [17], we proposed a solution to compute safe upper bounds on these delays. In this paper, we introduce two solutions that contribute to achieving tighter analysis of synchronisation-related delays: (a) a refined analysis of the stall times in sequences of critical sections (either different critical sections executed in a row or a single critical section embedded in a loop); and (b) a new strategy for lock granting, named CO-LoG, that is implemented in a set of primitives and helps to reduce drastically estimated stall times. Experimental results show noticeable decreases of estimated stall times and improvements on WCET estimates.

Acknowledgments

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement no. 287519 (parMERASA).

7. REFERENCES

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Spring Conference*, 1967.

[2] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In *SEUS*, 2011.

[3] R. Barik, J. Zhao, and V. Sarkar. Interprocedural strength reduction of critical sections in explicitly-parallel programs. In *PACT*, 2013.

[4] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *ERTS2*, 2010.

[5] M. De Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *RTCSA*, 2008.

[6] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Workshop on WCET Analysis*, 2007.

[7] S. Eyerman and L. Eeckhout. Modeling critical sections in Amdahl's law and its implications for multicore design. In *Comp. Architecture News*, 2010.

[8] M. Gerdes, F. Kluge, T. Ungerer, and C. Rochange. The split-phase synchronisation technique: Reducing the pessimism in the WCET analysis of parallelised hard real-time programs. In *RTCSA*, 2012.

[9] M. Gerdes, F. Kluge, T. Ungerer, C. Rochange, and P. Sainrat. Time analysable synchronisation techniques for parallelised hard real-time applications. In *DATE*, 2012.

[10] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In *Workshop on WCET Analysis*, 2010.

[11] A. Gustavsson, J. Gustafsson, and B. Lisper. Toward static timing analysis of parallel software. In *Workshop on WCET Analysis*, 2012.

[12] C. A. Healy, R. D. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Trans. on Comp.*, 48(1), 1999.

[13] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proc. of the IEEE*, 91(7), 2003.

[14] N. Holsti. Analysing switch-case tables by partial evaluation. In *Workshop on WCET Analysis*, 2007.

[15] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, 1995.

[16] L. Molesky, C. Shen, and G. Zlokapa. Predictable synchronization mechanisms for multiprocessor real-time systems. *Real-Time Systems*, (2), 1990.

[17] H. Ozaktas, C. Rochange, and P. Sainrat. Automatic WCET analysis of real-time parallel applications. In *Workshop on WCET Analysis*, 2013.

[18] M. Panić, G. Rodriguez, E. Quiñones, J. Abella, and F. J. Cazorla. On-chip ring network designs for hard-real time systems. In *RTNS*, 2013.

[19] M. Paolieri, J. Mische, S. Metzlaff, M. Gerdes, E. Quiñones, S. Uhrig, T. Ungerer, and F. J. Cazorla. A hard real-time capable multi-core SMT processor. *ACM Trans. Emb. Comput. Syst.*, 12(3), 2013.

[20] D. Potop-Butucaru and I. Puaut. Integrated Worst-Case Execution Time Estimation of Multicore Applications. In *Workshop on WCET Analysis*, 2013.

[21] A. Pyka, M. Rohde, and S. Uhrig. Performance evaluation of the time analysable on-demand coherent cache. In *TrustCom*, 2013.

[22] C. Rochange. An overview of approaches towards the timing analysability of parallel architecture. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, 2011.

[23] C. Rochange and P. Sainrat. A context-parameterized model for static analysis of execution times. *Trans. on HiPEAC*, 2, 2009.

[24] J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *LCTES*, 1999.

[25] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *DAC*, 2006.

[26] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ACM Sigplan Notices*, volume 44, 2009.

[27] T. Ungerer, et al. parMERASA – Multi-core execution of parallelised hard real-time applications supporting analysability. In *DSD*, 2013.