Parallelism Analysis: Precise WCET Values for Complex Multi-Core Systems

Timon Kelter and Peter Marwedel

Department of Computer Science, TU Dortmund
Otto-Hahn-Straße 16, 44227 Dortmund, Germany
{timon.kelter,peter.marwedel}@tu-dortmund.de

Abstract. In the verification of safety-critical real-time systems, the problem of determining the *worst-case execution time* (WCET) of a task is of utmost importance. Safe formal methods have been established for solving the single-task, single-core WCET problem. The de-facto standard approach uses abstract interpretation to derive basic block execution times and a combinatorial path analysis which derives the longest path through the program. WCET analyses for multi-core computers have extended this methodology by assuming that shared resources are partitioned in either time or space and that therefore each core can still be analyzed separately. For real-world multi-cores this assumption is often not true, making the classic WCET analysis approach either inapplicable or highly pessimistic. To overcome this, we present a new *technique to explore the interleavings of a parallel task system* as well as an *exclusion criterion* to prove that certain interleavings can never occur. We show how this technique can be integrated into existing WCET analysis approaches and finally provide results for the application of this new analysis type to a collection of real-time benchmarks, where average WCET reductions of 32% were observed.

Keywords: WCET, Multi-Core, Parallelism, Shared Resources

1 Introduction

WCET analysis is an important prerequisite for schedulability analysis and for overall system validation of safety-critical real-time systems, i.e. systems in which tasks must complete within a given deadline. The runtime of any task τ depends on its inputs, on the system state at the start of τ and on the interference imposed on τ by preempting tasks on the same core or by parallel tasks running on other cores. To compute the WCET, first an abstract interpretation on the domain of abstract system hardware states is run. With the resulting hardware state overestimations a safe bound on the runtime of each basic block can be derived. This procedure is called *microarchitectural analysis* (MA). As the last step, the *path analysis* determines the longest path through the program with the help of the basic block runtimes determined by the MA [19]. In this paper we propose an abstract interpretation of the system hardware state that is able to efficiently explore all possible interactions between multiple concurrently running tasks.

As soon as multiple cores may access a shared hardware resource in parallel, the runtimes of parallel tasks are no longer independent but they depend on

1. the order in which the requests arrive at the shared resource and
2. the policy with which requests to the shared resource are arbitrated.

Previous work has eliminated the first dependency by choosing a *state-partitioned* arbitration strategy which guarantees that the actions of any core C cannot modify the state of the shared resource as seen by cores $C_o \neq C$. This implies, that the delay for any access from C is independent of the potential concurrent accesses from all $C_o \neq C$. Therefore, we can still perform a per-core analysis and the state space does not become much bigger than for the single-core case. An example for such a state-partitioned strategy is *time-division multiple access* (TDMA) [5]. However, state-partitioned arbitration increases the average access delay compared to *state-permeable* strategies like *fair arbitration* (FAIR) and *fixed-priority arbitration* (PRIO) [6]. WCET analysis for these types of arbitration has been nonexistent or pessimistic at best. Therefore our main goal in this paper is to make a first step towards a precise WCET analysis for shared state-permeable resources, since they are often found in real-world systems.

2 Related Work

WCET analysis There is an extensive body of work on single-core WCET analysis as summarized in [19], which led to the standard approach of separating the microarchitectural analysis from the path analysis. Our techniques also build upon this concept by extending the former analysis to multi-cores.

The first known approach to multi-core WCET analysis is based on the *Real-Time Calculus* (RTC) [14,15]. It uses "access curves" to strongly abstract from the concrete system, which introduces strong pessimism in the results and is restricted to timing-compositional architectures [4]. The only known, non-RTC-based approach to the analysis of shared state-permeable resources is based on parallel summaries [10]. For a shared cache, it precomputes worst-case interference summaries for each core which contain the effects that *all* program points in *all* possibly concurrently running tasks can have on the state of the shared resource, which also introduces considerable pessimism. The authors of [1] combined the summary-based shared cache approach from [10] with a safe abstraction for the analysis of TDMA buses [5], which results in a scalable but pessimistic WCET analysis for multi-core WCET estimation. Finally, model-checkers have been used to determine multi-core WCETs [3] and these could potentially also handle state-permeable resources. Unfortunately the approach does not scale to bigger programs or realistic systems, since the generic model checker has few possibilities of pruning the huge search space.

Parallel program analysis Static analysis of the synchronization structure of concurrent programs was first considered by [17] where the analysis of the

"concurrency state" of the system and the notion of a parallel execution graph was first established. We build our work on this, though the analysis in [17] worked at a far more coarse-grained level. A reference approach to bit-vector-based abstract interpretation on programs with explicit fork-join parallelism is given in [9]. Unfortunately, the microarchitectural analysis that we are examining here is not a bit-vector problem. In reachability analysis for parallel programs "stubborn sets" [18] can be used to prune the search space, but again the microarchitectural analysis differs significantly from reachability analysis. Finally, a recent publication [12] examines the computation of feasible synchronization-aware parallel interleavings. Their approach focuses on path analysis and is thus orthogonal to ours.

3 System and Task Model

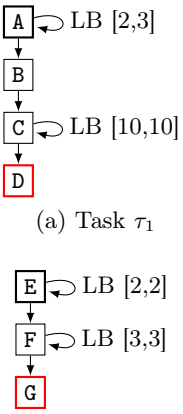
We assume a task set T containing only strictly periodic tasks, as often found in hard real-time systems. In the following sections, we will need a common reference point in time for all running tasks, where times are measured in multiples of the shortest clock cycle. Therefore we first require that all $\tau_i \in T$ are sharing the same period $p_i = p_T$ and that each task is executed non-preemptively on a separate core. We will discuss how to lift these restrictions in Section 5. Each task τ_i may have a different release time r_i within the common period.

The analysis can be adapted to any topology, but for our experiments we will use an example architecture with $n = |T|$ ARM7TDMI cores,[1] each having a private cache and a scratchpad. The cores are connected to a shared bus which is arbitrated under either TDMA, FAIR round-robin or fixed core priorities. Behind the bus, shared instruction and data caches are located as well as non-cached memories.

4 Parallelism Analysis

Before starting with the formal part of the framework, we briefly sketch the intuition behind the analysis procedure. Our goal will be to efficiently explore all feasible interleavings of multiple tasks running in parallel. As an example, consider the execution of the tasks from Figure 1 under the assumption that both tasks start concurrently at time 0. For this assumption we can find all valid parallel execution scenarios from the *parallel execution graph* (PEG) shown in Figure 2. The construction of this graph starts with nodes corresponding to the initial system states, in this case with only the node `AE` (the δ-values will be explained below). From these start nodes, we iteratively simulate cycle steps of the system. To keep our example PEG from Figure 2 sufficiently small, we assume that every block will take one cycle to complete. Therefore, our initial block `AE` is terminated after the first cycle and the execution must continue in

[1] The choice of ARM7TDMI cores is motivated by the fact that we already have an implementation of the abstract pipeline model for these cores (compare Section 4.4).



(a) Task τ_1

(b) Task τ_2

Fig. 1: Two example tasks with given loop bounds.

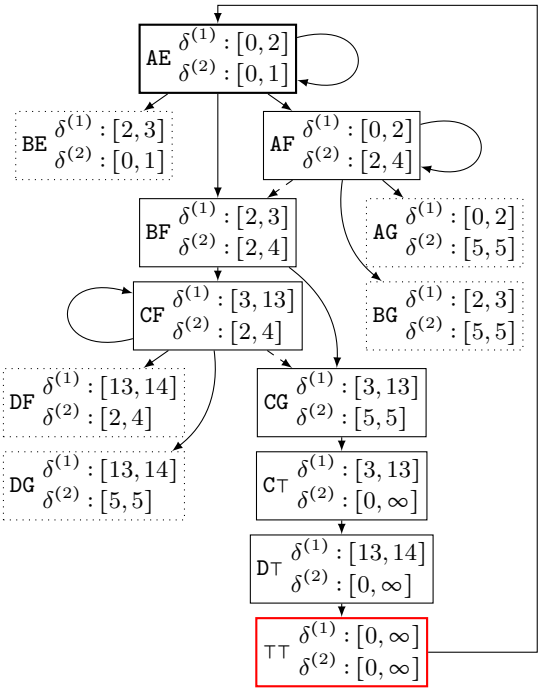


Fig. 2: The final Parallel Execution Graph for tasks τ_1 and τ_2 from Figure 1, starting synchronously at time 0.

one of the nodes AE, BE, BF and AF. To generate these successors we simply follow all combinations of successor blocks in the task CFGs. The loop bounds are not used here. If we continue the graph construction in this manner, we will end up with a full product graph of the task CFGs. When every core has reached the end of its task, indicated by the "⊤" sign in Figure 2, we add a back-edge from ⊤⊤ to AE to account for the repeated execution of the tasks in the cyclic schedule. The purpose of this final PEG is, that it contains each basic block of each task in all possible parallel execution scenarios. Thus we can derive the WCET of each basic block from the PEG and use these to compute the task WCETs.

As visible, the PEG in Figure 2 is *not* a full product graph of the graphs from Figure 1. The construction of the graph has been stopped at nodes BE, AG, BG, DF and DG. To explain why this was done, and why it is correct, we need the δ-values and the loop bounds. We define $\delta^{(i)}$ as an interval containing all points in time, measured from the beginning of the common period p_T, at which a node may be entered on core i. Initially we set $\delta^{(1)} = \delta^{(2)} = [0,0]$ for node AE, since core 1 (2) enters node A (E) at time 0. From here on, every time we visit a node X in the analysis, we recompute its δ intervals with the help of a path analysis which computes the length of the shortest and longest paths to the basic blocks in X. As an example, when we visit node AE the second time, we have already seen, that both block A and E complete within one cycle. Therefore, since A can

be executed at most three times and `E` at most two times (see Figure 1), the path analysis can infer that any execution of block `A` must begin in the time frame $\delta^{(1)} = [0, 2]$ and similarly any execution of block `E` must begin within $\delta^{(2)} = [0, 1]$. Thus, the path analysis always operates only on the CFGs of the individual tasks, *not* on the PEG. The PEG is only used to compute the possible runtimes of the basic blocks within the tasks.

The path analysis for node `BE` yields $\delta^{(1)} = [2, 3]$ (due to the loop at `A` which must complete before `B`) and $\delta^{(2)} = [0, 1]$. Here we can see the application of the computed δ-values: We can exclude this node from the PEG and thus from the analysis. Through the δ-values we know, that at this point blocks `B` and `E` cannot be executed concurrently because their execution time windows do not overlap. All blocks for which we can prove this can be removed from the PEG as long as their δ-values stay unmodified. In Figure 2 these removed blocks are marked by a dotted border. If accesses to a shared resource, with a duration of one cycle, would occur in `B` and `E` we would still obtain the same PEG which shows that these accesses can never interfere with each other.

4.1 Framework

The phases of our WCET analysis framework are shown in Figure 3. We are using the same CFG reconstruction, value analysis and path analysis stages as the classical WCET analysis [19]. These stages also work for each task in separation. Only for the microarchitectural analysis, we first construct the initial PEG states, based on the system schedule. Then we conduct a data-flow analysis on the PEG until the PEG itself as well as the associated system states have reached a fix-point. From this converged PEG we extract the basic block runtimes that are finally used to compute the WCET and BCET in an IPET-based path analysis.

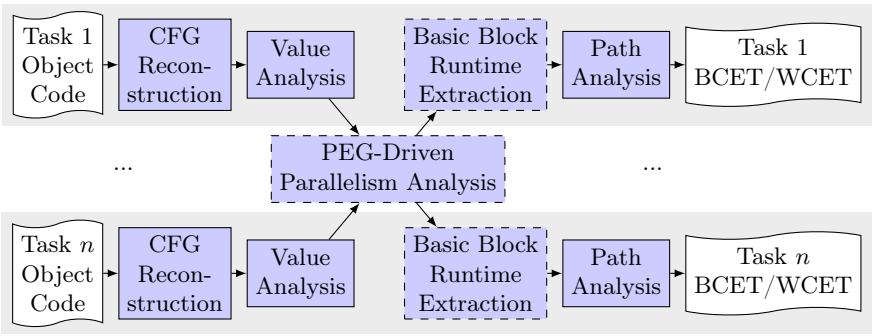


Fig. 3: The analysis framework. The dashed parts are new contributions compared to [19] and will be discussed in the next sections.

4.2 Prerequisites

To precisely define our analysis procedure we will need some terminology which is introduced in the following.

Given a set of tasks T together with CFGs $G_\tau = (V_\tau, E_\tau)$ for all $\tau \in T$, a *task execution position* ψ_τ is a tuple (v, i, c, d), where $v \in V_\tau$ is a basic block, $i \in v$ is an instruction within that basic block and c is the number of cycles that were already spent on the processing of this instruction. Finally, d is the number of cycles that the task must wait until its execution will begin. A *system execution position* (SEP) Ψ on n cores is an n-tuple with $\Psi \in \hat{\Psi} = \bigtimes_{i=1}^{n} \hat{\psi}_{\tau_i} \cup \{\top\}$, τ_i being the task mapped to core i. The special token $\top$ indicates that the respective core is currently running idle. Here and in the following we use $\hat{A}$ to denote the set of all tuples of type A. The motivation for this definition is, that other than in our introductory example from Figure 2, real basic blocks will contain more than one instruction[2] each of which may take multiple cycles to complete. Still we need to be able to split the execution of each basic block into chunks which may be as small as a single CPU cycle, as we will see in the following. We will use SEPs to specify the point at which the execution is resumed in a PEG block, therefore SEPs correspond to the block labels from Figure 2 (e.g. `AE`, `BE`, `AF`, etc).

An *abstract parallel system state* (APSS) $\Sigma \in \hat{\Sigma}$ is a structure which models a set of concrete states of an entire parallel system, including all cores and memory hierarchy elements. Again, $\hat{\Sigma}$ is the set of all possible APSSs. We give more detail on how to form proper APSSs at a later point, for now we only require a *cycle step function* $\xi_\Sigma : \hat{\Sigma} \times \hat{\Psi} \times 2^{\{1,\dots,n\}} \to (\{0,1\}^n \times \hat{\Sigma})$. The invocation of $\xi_\Sigma(\Sigma, \Psi, \alpha)$ must simulate all possible state transfers that may happen when a single clock cycle is executed at position Ψ in system state Σ. However, only the cores in the set $\alpha \subseteq \{1, \dots, |T|\}$ may perform a cycle step, to be able to account for different release times. For any *instruction completion vector* $c \in \{0,1\}^n$ which may occur in this cycle, it must specify the result state, where c defines for each core, whether it has completed the execution of its current instruction (1) or not (0). The "current instruction" is always given by the "program counter" register value.

The APSSs will be subject to a data-flow analysis, therefore we also require a partial order $\sqsubseteq$ on $\hat{\Sigma}$ such that $(\hat{\Sigma}, \sqsubseteq)$ is a lattice [7], with a *supremum* or *join* function $\sqcup : \hat{\Sigma} \times \hat{\Sigma} \to \hat{\Sigma}$. Intuitively, since APSSs represent sets of concrete states, $\Sigma_1 \sqsubseteq \Sigma_2$ specifies whether Σ_2 completely contains Σ_1. To ensure the termination of the data-flow framework ξ_Σ must also be monotonic with respect to $\sqsubseteq$.

A *Parallel Execution Graph* $G_P = (V_P, E_P)$ is a directed graph with node set $V_P \subseteq \hat{\Psi} \cup \{\bot\}$ and edge set $E_P \subseteq V_P \times V_P$. $\bot$ is a special PEG node which is exclusively used to model the situation that the execution of the parallel system has not yet started. For any PEG we define a *block time window* function $\delta : V_P \to \hat{I}^n$, an *edge state function* $\lambda : E_P \to \hat{\Sigma}$ and a *block length function* $\omega_P : V_P \to \mathbb{N}$. $\hat{I} = \{[x, y] \subset 2^{\mathbb{N}} | x \leq y\}$ is the set of all execution time intervals, measured in cycles from the last point where all cores were synchronized. The

[2] In the example we have not even differentiated between basic blocks and instructions.

Algorithm 1 PEG-driven parallelism analysis

```
1  function ParallelismAnalysis(Σ_start, G_τ1, ..., G_τn)
2    ∀τ : ∀v ∈ V_τ : ω_C(v) = ∅                    ▷ Initialize all context block runtimes to ∅
3    Q ← (v_τ1^start, 0, 0, r_1) × ··· × (v_τn^start, 0, 0, r_n)        ▷ Initialize start block
4    G_P ← (Q ∪ {⊥}, ∅)
5    δ(⊥) ← [0,0]^n, ω_P(⊥) ← ∞                    ▷ Initialize pre-execution state ⊥
6    ∀v ∈ Q : δ(v) ← ∅^n, λ((⊥, v)) ← Σ_start, ω_P(v) = ∞        ▷ Initialize start state
7    while Q ≠ ∅ do
8      v = PopFront(Q)                                          ▷ Analyze next block
9      ω_C ← GatherNewBBTraces(ψ, G_P, ω_P, ω_C)                ▷ Update ω_C
10     for i ∈ {1, ..., n} do                  ▷ Update δ-window for all cores
11       δ(v)^(i) ← ⋃_(u,v)∈E_P δ(u)^(i) + ω(u)
12       if IsLoopHeadOrExit(v^(i)) then
13         δ(v)^(i) = r_i + PathAnalysis(v^(i), G_τi, ω_C)
14     if ∀_i∈{1,...,n} δ(v)^(i) ≠ ∅ ∧ ⋂_i=1^n δ(v)^(i) = ∅ then   ▷ If BEC holds ...
15       continue                                  ▷ ... skip the current block v ...
16     else                                                       ▷ ... else analyze v
17       λ_prev ← λ, G_P,prev ← G_P
18       (G_P, λ, ω_P) ← AnalyzeBlock(v, G_P, λ, ω_P)
19       if λ_prev ≠ λ ∨ G_P,prev ≠ G_P then   ▷ If graph or states were altered ...
20         ∀(v, z) ∈ E_P : PushBack(Q, z)              ▷ ... propagate the changes.
21         if E_P,prev ≠ E_P then                        ▷ If edges were added ...
22           ∀v ⇝_G_P z : PushBack(Q, z)                ▷ ... propagate δ-changes
23   return ω_C
```

time window function will be used to rule out infeasible SEPs as indicated in Figure 2, the edge state function is used to propagate the possible hardware states from one PEG node to the other and the block length function specifies how many cycles were spend on the execution of a PEG node. The three functions are not defined a priori. They will be computed by the algorithms presented in the following.

We denote by $v_1 \rightsquigarrow_G v_2$ that there is a path in the directed graph $G = (V, E)$ from $v_1 \in V$ to $v_2 \in V$, i.e. that v_2 is *reachable* from v_1.

Our goal in the parallelism analysis is to compute the *CFG block lengths* $\omega_C : V_\tau \to \hat{I}$, which are then used by the path analysis. Note this these are *not* identical to ω_P. The block lengths in Figure 1 are given by ω_C, whereas the block lengths in Figure 2 are given by ω_P.

4.3 Analysis Algorithm

The outline of the main analysis is shown in Algorithm 1.It starts with an initialization of the initial context block runtimes ω_C in line 2 and of the work-list Q in line 3. According to the system schedule, the SEP consists of the begin of the start block of each task ($v_{\tau_i}^{\text{start}}$) with a delay of r_i cycles. This SEP is assigned a time window of $[0, 0]$. We also create a virtual edge $(\bot, v)$ pointing to it, which is assigned the initial APSS Σ_{start}. The start block $\bot$ itself has a runtime of zero cycles and executes in the start window $[0, 0]$ to mark that the schedule starts

here. Then we process items from the queue Q until it gets empty (line 7). In the main loop, we extract the first block v from the queue and check whether v models the end of a basic block v_τ on any core in the call to GatherNewBBTraces in line 9. For any such context block v_τ, its runtime $\omega_C(v_\tau)$ is updated in GatherNewBBTraces as shown in Algorithm 2.

In line 11 we infer the block time window for all task positions $v^{(i)} \in v$ from the windows and runtimes of its predecessors.[3]

If v is part of a sequential block chain, the δ-update in line 11 is sufficient. On the other hand, if v is a loop head (like `A` in Figure 1) or a loop exit (like `B` in Figure 1), then we have to take the loop bounds into account to determine the block time window, like we have done in the computation of $\delta^{(1)}$ in e.g. `AE` and `BE` in Figure 2. This is done in line 13, where the existing path analysis of our framework is used to compute the shortest and the longest path from $v_{\tau_i}^{\text{start}}$ to $v^{(i)}$. We currently use an adapted IPET analysis based on Integer Linear Programming [11] here, but advanced single-source all-sinks analyses would be even better suited [8]. It follows the given loop bounds and uses ω_C as the runtime of individual basic blocks in G_{τ_i}. If any block $u \in G_{\tau_i}$ with $u \rightsquigarrow_{G_{\tau_i}} v^{(i)}$ and $\omega_C(u) = \varnothing$ exists, the path analysis will return $\varnothing$ for the path length to $v^{(i)}$, thus keeping $\delta(\psi)^{(i)} = \varnothing$.

The δ values are used in line 14, where we try to apply the *block exclusion criterion* by intersecting all block time windows. However, this test can only be applied if the time windows for each task could already be determined, i.e., if they are not empty. If the intersection is empty, this SEP cannot be reached from its current predecessors and we its analysis in line 15. This is exactly what we have done with `BE` in Figure 2. Still, we may need to analyze v in the future when it becomes accessible via new edges. Then we will re-check whether our exclusion criterion still holds. Thus, this skipping is effectively either postponing or avoiding the graph growth at v.

If the exclusion criterion does not hold (line 16), we analyze the *parallel execution block* (PEB) beginning at node v (line 18). This analysis will determine a block runtime $\omega_P(v)$, an output APPS for all out-edges of v and possibly alter G_P. If the output states or the graph are changed, we push the successors of v into the work-list at line 20. By doing this, all changes to the block time windows δ, edge states λ and block runtimes ω_P will be propagated through the graph. Finally, if we have added edges to the PEG, we also push all blocks z which are reachable from v into Q (line 22), to ensure that a new attempt to compute $\delta(z)$ is started, if z is a loop head or exit. The algorithm terminates when no more edges are added and all edge states have converged.

All in all Algorithm 1 is a standard data-flow analysis work-list algorithm, with the difference that we are dynamically expanding (line 18) the underlying graph. When ParallelismAnalysis has finished, all reachable blocks of all tasks will have been visited in one or more parallel execution blocks and BBRuntime will therefore yield valid runtimes for all basic blocks.

[3] Here and in the following we use $()^{(i)}$ to access the i-th element of a tuple.

Algorithm 2 Update of basic block runtimes

```
1  function GatherNewBBTraces(v, G_P, ω_P, ω_C)
2    for i ∈ {1,...,n}, (u,v) ∈ E_P do
3      if v^(i)(1) ≠ u^(i)(1) then          ▷ If v is context block start on core k, ...
4        u_τ,pred = u^(i)(1)                ▷ ... collect the length of all paths to starts of u_τ,pred ∈ V_τ.
5        ω_C(u_τ,pred) ← ω_C(u_τ,pred) ∪ TraceToStarts(u_τ,pred, u, i, G_P, ω_P)
6    return ω_C
7  function TraceToStarts(v_τ, v, i, G_P, ω_P)
8    if v^(i) = (v_τ, i_0, 0, 0) then     ▷ If v^(i) is a begin of v_τ, ...
9      return ω_P(v)                      ▷ ... finish this trace.
10   else                                 ▷ Else continue with the recursion.
11     return ⋃_(u,v)∈E_P {ω_P(v) + TraceToStarts(v_τ, u, i, G_P, ω_P)}
```

To complete the view on the analysis, Algorithm 3 shows the function AnalyzeBlock which is tightly coupled with Algorithm 1. First, the incoming APSSs are joined in line 2. The current system execution position Ψ_{run} is initialized to v (remember that $V_P \subseteq \hat{\Psi}$) and the block duration $\omega_P(v)$ is set to zero. Then we simulate the effect of successive system cycle steps on Ψ_{run} and Σ_{run}, until on any core, either a) the end of a basic block is reached or b) the successor SEP is ambiguous. The latter happens, when it is uncertain in APSS Σ_{run} whether the current instruction of at least one core will complete or not. In this case we track all completion combinations in separate successor blocks.

The first step in each cycle is to invoke the APSS cycle step function ξ_Σ, which is done in line 5, but only for those cores with zero delay cycles (set α). The APSS cycle step function ξ_Σ returns a mapping $\kappa \subseteq \hat{I} \times \hat{\Sigma}$, i.e. it associates instruction completion vectors to successor APSSs. Line 6 checks the two block termination conditions a) and b) mentioned above. The helper function $\phi_c^\alpha : \hat{\Psi} \to \hat{\Psi}$ generates the successor SEP for a given SEP Ψ, instruction completion vector c and active core set α. If neither a basic block end is reached, nor the successor SEP is ambiguous, we take over the results of the cycle step as our new working SEP Ψ_{run} and APSS Σ_{run} in line 7 and increment the cycle counter for this block in line 8. Here, $\Psi_{\text{run}}^{(i)(1)}$ is the basic block executed by core i, $\kappa^{(1)(1)}$ is the first instruction completion vector and $\kappa^{(1)(2)}$ is its associated successor APSS.

If the block end is detected, we terminate the current block as shown from line 9 on. It will be one invariant of our analysis that the length of a block can only stay the same or be reduced in successive analyses of the same block. Therefore we only check in line 10, whether the block has been shortened. This may happen due to a newly joined-in APSS, that triggers an earlier ambiguous successor SEP. In this case, we remove all previous out-edges of the current block v (line 11). In any case, we add for each instruction completion vector c an out-edge to $\phi_c^\alpha(\Sigma_{\text{run}})$ which gets annotated with the respective out-state Σ_c (lines 13–16). In the end, the modified graph, edge states and block lengths are returned in line 18.

Algorithm 3 PEG block analysis

```
1  function AnalyzeBlock(v, G_P, λ, ω_P)
2    Σ_run ← ⊔_{∀e=(u,v)∈E_P} λ(e)                          ▷ Join incoming states
3    Ψ_run ← v, ω_{P,prev} ← ω_P, ω_P(v) ← 0
4    while true do
5      κ ← ξ_Σ(Σ_run, Ψ_run, α = {i | Ψ_run^(i) = (·,·,·,0)})   ▷ Simulate next cycle in block
6      if |κ| = 1 ∧ ∄i : (φ^α_{κ^(1)(1)}(Ψ_run))^(i)(1) ≠ Ψ_run^(i)(1) then   ▷ Split/Basic block end?
7        Σ_run ← κ^(1)(2), Ψ_run ← φ^α_{κ^(1)(1)}(Ψ_run)        ▷ If not, prepare next cycle
8        ω_P(v) ← ω_P(v) + 1
9      else                                                  ▷ Else terminate the current block
10       if ω_P(v) < ω_{P,prev}(v) then                      ▷ Remove old edges on block shrinking
11         E_P ← E_P ∖ {(v, w) ∈ E_P}
12       for (c → Σ_c) ∈ κ do                                ▷ Add new successors and out-states
13         V_P ← V_P ∪ {v_new = φ^α_c(Σ_run)}
14         δ(v_new) ← ∅^n, ω_P(v_new) ← ∞
15         E_P ← E_P ∪ {e_new = (v, v_new)}
16         λ(e_new) ← Σ_c
17       break
18   return (G_P, λ, ω_P)                                    ▷ Return all modifications
```

With Algorithm 3 we completed the macroscopic side of the analysis. In the next subsection we will examine the microscopic perspective, namely how to efficiently represent abstract parallel system states.

4.4 Parallel System State Models

An APSS must model the *state* of all microarchitectural components which are relevant to the timing of the system, i.e. all cores and their pipelines and all *memory hierarchy elements* (MHEs) like private and/or shared caches, buses and memories. Here, *state* denotes an approximation of the relevant content of the component as well as the operation that the component is currently performing.

Therefore we define an APSS Σ as a set of tuples, where each tuple contains abstract states for each pipeline and memory hierarchy element in the system. The rationale behind Σ being a set of tuples is, that we may have to split the state, e.g. when two different paths in the pipeline must be considered. These different execution paths may have identical instruction completion vectors, but still we need to maintain them separately in a common Σ set, to trace the different microarchitectural behaviors.

The driving force behind the microarchitectural simulation are the cores' pipelines, which are modeled as non-deterministic finite-state machines [19]. In each cycle, the abstract pipeline states follow all transitions which are enabled according to their current state which includes the currently executing instructions. Multiple transitions may be enabled due to uncertainty in the analysis, e.g. due to statically unknown memory access targets and register values. In such a case, one successor state is generated for every possible transition. During the abstract cycle step, the pipeline models issue memory transactions as dictated

by the machine specification. Completion of such transactions is signaled back from the abstract MHE states to the affected pipeline state. Finally, the completion of instructions, known as the *commit* of an instruction, is communicated to our framework via an entry in the instruction completion vector as introduced in Section 4.2.

In every cycle step, i.e. every invocation of ξ_Σ, we perform the cycle step independently on each tuple $\sigma \in \Sigma$. The results are then sorted by completion vector and returned to returned to the PEG block analysis (Algorithm 3). Inside the individual σ tuples we use established abstract domains, namely *abstract finite state machines* for pipelines [19], *cache block age maps* for caches [19] and *TDMA offset sets* for TDMA busses [5]. For FAIR and PRIO arbitration no suitable abstractions were found in the per-core analysis. Since we explicitly track parallel interleavings in the PEG, we can analyze these protocols for the first time by providing *abstract arbitration functions* as shown in the following.

Arbitration functions A simplified version of the bus state is illustrated in Figure 4, where a PEG block Ψ is shown. The state Σ_{run} for this block (see Algorithm 3) holds two sub-states, of which σ_2 is presented in more detail. In this sub-state the two cores in this example are currently performing a multiplication and an instruction fetch. Bus `B1` is a TDMA bus, from its state we know that we currently are either at cycle 0 or 4 in the fixed-length, cyclic TDMA schedule. The state for FAIR-arbitrated buses like `B2` holds an overapproximation of the cores which may have last accessed the bus. In the case of `B2` this reveals that the last access has definitely been carried out by core 2.

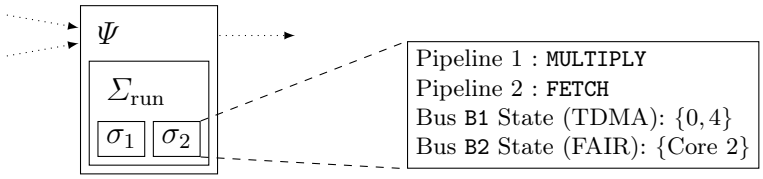


Fig. 4: An example PEG block Ψ with attached APSS Σ_{run}.

With these state definitions we can easily define the abstract arbitration functions which determine possible arbitration winners:

- **TDMA**: All cores whose *grant window* has a non-empty intersection with the current TDMA offsets *may* be granted. If we assume a schedule of length 10 cycles, in which cycles $[0-4]$ are assigned to core 1 (grant window of core 1) and cycles $[5-9]$ are assigned to core 2 (grant window of core 2), then in the state from Figure 4 a request to `B1` would only be granted for core 1.
- **FAIR**: All cores which are the next in the core list for at least one previously accessing core c_p *may* be granted. In Figure 4 if both cores request access to `B2`, only the request from core 1 will be granted.
- **PRIO**: All requests with the highest priority *may* be granted. Thus for PRIO we do not need to maintain any kind of state, since the arbitration can be done solely based on the fixed priorities.

Different arbitration outcomes are then distributed to different result tuples σ. Since the PEG already carries the burden of constructing all possible interleaving scenarios, we can formulate the arbitration analysis in a rather simple manner, here. By construction, this has not been possible for the standard per-core WCET analysis approach.

4.5 Correctness

Formally complete proofs cannot be given here due to space constraints, but we try to provide some intuition on why the analysis is correct. In the following, we use G_P^i, λ^i, ω_P^i and δ^i to denote the PEG and the values of the three functions *after* i-th iteration of the main loop of Algorithm 1. Also, we denote the PEG node v that is analyzed in iteration i as v^i. The special iteration number 0 is used to denote the state *before* the first iteration of the main loop. First of all, through the monotonicity of ξ_Σ, we can prove Lemma 1, which states that with rising analysis iteration count, for each $v \in V_P$ the block runtime will only shrink, the incoming APSS will only get more imprecise and the execution time intervals for each task execution position will only become wider.

Lemma 1. *For any iteration j of the main loop of the parallelism analysis (line 7 in Algorithm 1), any iteration $i < j$ and any SEP v, the following invariants hold:*

1. $\forall u \in V_P^i : u \rightsquigarrow_{G_P^i} v \implies u \rightsquigarrow_{G_P^j} v$,
2. $\lambda_{in}^i(v) \sqsubseteq \lambda_{in}^j(v)$ *where* $\lambda_{in}^i(v) = \bigsqcup_{e=(u,v)\in E_P^i} \lambda^i(e)$,
3. $\omega_P^i(v) \geq \omega_P^j(v)$, *and*
4. $\forall k \in \{1,\ldots,n\} : \delta^i(v)^{(k)} \subseteq \delta^j(v)^{(k)}$.

For any possible task set execution, which we model as a sequence S of SEPs, we can prove with Lemma 1, that the APSSs attached to the converged PEG are safe over-approximations of the concrete system states with which S is traversed. This yields Theorem 1.

Theorem 1. *The basic block runtimes ω_C as returned by Algorithm 1 are safe over-approximations of the concrete block runtimes in any possible parallel execution scenario.*

5 Analysis Extensions

If the underlying architecture is guaranteed to be free of *timing anomalies* [4], then in each block analysis (Algorithm 3, line 5) we can skip all instruction completion vectors $c \in \kappa$ which are dominated by another vector, i.e. $c_1 \prec_c c_2 \Leftrightarrow \forall i \in \{1,\ldots,n\} : c_2^{(i)} \Rightarrow c_1^{(i)}$. The dominated vectors correspond to an earlier termination of an instruction and since in a timing-anomaly-free architecture every local worst-case action is always also the global worst-case action, we can

assume that they are never part of the worst-case path. This can drastically reduce the state space and the PEG size.

In task sets with explicit synchronization points we have to consider these points in the path analysis as shown in [13]. In addition we can also use them to prune the PEG as we have done in Section 4, since a task which is waiting for synchronization cannot progress until a partner has arrived to complete the rendez-vous. This idea has already been used in [17] and similar to there, it can be used on top of the timing information to further prune the PEG.

The extension of our framework to task sets with non-uniform periods is also possible. With non-uniform task periods we can still compute the global hyperperiod, i.e. the smallest common multiple of all task periods and build a PEG for this hyperperiod. The problem that we face here is, that with the current framework we cannot determine the absolute point in time at which we are when a task instance has finished executing, since then we can no longer compute the block time window on the basis of the local CFG and a task release time. This means we would have to assume in every successive cycle step, that the next task instance might start or not, which would drastically increase the PEG size. However this can be limited if we take into account synchronization structures or if timing-based approximations of the task instance spawn behavior can be found.

6 Evaluation

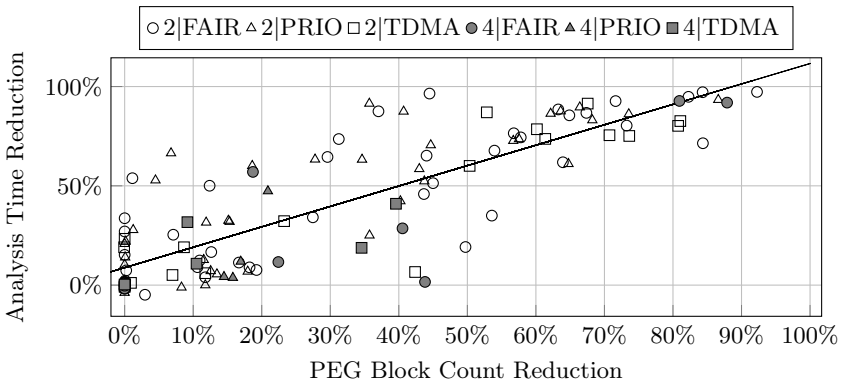


Fig. 5: Efficiency of the block exclusion criterion on example benchmarks for varying number of cores and arbitration policies. The solid line is a linear regression of the data points.

We implemented the analysis algorithms inside the WCC compiler framework [2], which was also used in [6]. We ran our evaluations on single-core tasks from the MRTC and DSPStone real-time benchmark suites. Out of these single-core tasks we formed packages of 2 to 4 tasks, all of which were assigned a release time of 0. We analyzed the system topology from Section 3 with 2 or 4 cores, depending on the task set. In the evaluation, we focus on analyzing state-permeable

Table 1: Average analysis time and PEG sizes.

Schedule	Analysis	Duration	#PEBs
FAIR	C|N	4s	0
FAIR	P|O|N	1,695s	2,177
FAIR	P|B|N	583s	1,223
FAIR	C|T	6s	0
FAIR	P|O|T	2,065s	9,595
FAIR	P|B|T	801s	7,828
PRIO	P|O|N	1,438s	1,800
PRIO	P|B|N	514s	1,175
PRIO	P|O|T	1,971s	6,971
PRIO	P|B|T	808s	5,118

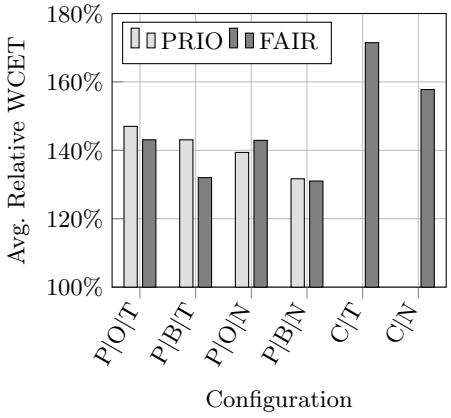


Fig. 6: Relative WCET results.

bus arbitration methods (PRIO and FAIR) which were not analyzable (PRIO) or not precisely analyzable (FAIR) without the presented parallelism analysis. The bus which is arbitrated by these methods is the shared memory bus introduced in Section 3.

In Figure 5 the results of our block exclusion criterion (BEC) from Algorithm 1, line 14 are shown. Each mark represents one analysis run on one task set. The circle marks indicate runs where the shared bus was configured for FAIR arbitration, the triangles correspond to fixed priority-based arbitration and the squares correspond to TDMA. Non-filled (filled) marks are analysis runs with the 2-core (4-core) system. The x-axis value is the number of PEG blocks that are generated during the analysis, when the BEC is used compared to the case when it is not used (100%). On the y-axis the required analysis time is shown, also compared to the case that the BEC was not used (100%). From the data points and the solid regression curve it is visible that the analysis time scales roughly linearly with the number of PEG blocks, which was expected, since the runtime of the main loop in Algorithm 1 depends on the total number of blocks. The variations stem from the convergence behavior of the individual benchmarks, i.e. how often loops have to be visited until the attached APSSs converge. More importantly, we can see from Figure 5 that the BEC is effective, as on average it rules out 35.6% of all blocks and leads to a reduction in analysis time of 49.7%.

The average resulting analysis time is presented in Figure 1. The column "Analysis" shows which type of WCET analysis was tested. We compare the classical multi-core WCET analysis [1] (abbr. "C") to our new parallelism analysis with (abbr. "P|B") and without (abbr. "P|O") usage of the block exclusion criterion. As already seen in Figure 5, "P|B" is always superior to "P|O" but both are slower than the classical approach "C" by a factor of 130 on average. This is a result of the more complex system state and of the thousands of parallel interleavings that have to be explored, whereas the classical analysis only operates on the CFG of a single task and the state of a single core. The last element of the "Analysis" column shows whether the architecture was assumed to have *timing*

anomalies (abbr. "T") or not (abbr. "N"). As presented in Section 5, this can be used to drastically reduce the PEG size, which is visible in Figure 1 in column "#PEBs", which holds the average number of PEG blocks for this analysis scenario. The configurations where absence of timing anomalies was assumed ("N") produce far lower PEG sizes and analysis times than their counterparts ("T").

The benefits we get from the parallelism analysis ("P"-configurations) at the price of increased analysis times are that we can *analyze the PRIO arbitration for the first time* and that we can *significantly reduce the arbitration delay estimations for FAIR arbitration.*

Details on both aspects are presented in Figure 6, where the average of the quotient of WCET and measured runtime (MRT) is shown for different analysis configurations from Figure 1. Remember here, that we can only determine a safe upper bound $WCET_{\text{est}}$ on the real $WCET_{\text{real}}$ in all of our analyses. Therefore the above quotient is a bound on the WCET overestimation, since by $WCET_{\text{est}} \geq WCET_{\text{real}} \geq MRT$ we have that $WCET_{\text{est}} \div MRT \geq WCET_{\text{est}} \div WCET_{\text{real}}$. Each MRT was determined by simulating the task set execution for the given system configuration on the cycle-true virtual prototyping IDE CoMET [16].

First of all, we can see in Figure 6 that the PEG-based WCET analyses (all configurations containing "P") for a system with PRIO arbitration yield results that are comparable to those for FAIR arbitration. The remaining overestimation is mostly due to other unavoidable sources of imprecision, like loose loop bounds and pipeline and value analysis overestimation. Also, we see that the restriction to timing-anomaly free architectures (all configurations with "N") enables not only reduced analysis times (cf. Figure 1) but also tighter WCET estimations. The usage of the block exclusion criterion (configurations with "B") also leads to slightly decreased overestimation.

Finally, the "C"-configurations show the overestimation for the classical WCET analysis framework, which can only assume the maximum possible delay for every access in state-permeable arbitration policies. Our new parallelism-based analysis is able to clearly outperform this approach, being 32% more accurate on average, but of course at the expense of increased analysis times.

7 Conclusions

We have presented a new type of WCET analysis which can precisely bound the runtime of safety-critical tasks running on complex multi-core systems. This is achieved by exploring all possible execution interleavings of a parallel periodic task set. A *parallel execution graph* (PEG) is employed to represent the interleavings in compressed form, a concept that was already used in [17]. What is genuine to the application of the PEG in WCET analysis is firstly that here we must work at the granularity of single machine cycles which drastically increases the graph size. But secondly and more importantly we can also use the timing information that we are generating for *pruning* parts of the graph which we prove to be not reachable in any real execution through the use of a new timing-based *block exclusion criterion.*

We tested this analysis on a prototype implementation. For a shared bus scheduled under a fair round-robin policy we observed WCET reductions of 32% on average, compared to previous analysis approaches. For fixed priority-based scheduling no previous individual-access analysis methods exist. Here we could derive WCET values with a tightly bounded maximum overestimation of only 30–50% on average, which is comparable to the single-core WCET overestimation ratio of our analyzer. In the future we plan to explore combinations of the block exclusion criterion and synchronization-aware analysis to further reduce the PEG size and lift the restriction that all tasks must have a uniform period. We also seek to evaluate the performance of the PEG-based analysis for systems with shared caches, for which up to now only pessimistic analyses existed.

8 Acknowledgments

This work was partially supported by EU COST Action IC1202: Timing Analysis On Code-Level (TACLe). The authors would also like to thank Synopsys for the provision of the virtual prototyping IDE CoMET.

References

1. Chattopadhyay, S., Kee, C., Roychoudhury, A., Kelter, T., Marwedel, P., Falk, H.: A Unified WCET Analysis Framework for Multi-Core Platforms. In: Real-Time and Embedded Technology and Applications Symposium (2012)
2. Falk, H., Lokuciejewski, P.: A Compiler Framework for the Reduction of Worst-Case Execution Times. Journal on Real-Time Systems 46(2), 251–300 (October 2010)
3. Gustavsson, A.: Worst-Case Execution Time Analysis of Parallel Systems. In: Nyström, D., Nolte, T. (eds.) Real Time in Sweden 2011. pp. 104–107. Dag Nyström and Thomas Nolte (June 2011)
4. Hahn, S., Reineke, J., Wilhelm, R.: Towards Compositionality in Execution Time Analysis – Definition and Challenges. In: International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (December 2013)
5. Kelter, T., Falk, H., Marwedel, P., Chattopadhyay, S., Roychoudhury, A.: Bus-Aware Multicore WCET Analysis through TDMA Offset Bounds. In: Euromicro Conference on Real-Time Systems. pp. 3–12. Porto, Portugal (July 2011)
6. Kelter, T., Harde, T., Marwedel, P., Falk, H.: Evaluation of Resource Arbitration Methods for Multi-Core Real-Time Systems. In: International Workshop on Worst-Case Execution Time Analysis (July 2013)
7. Kildall, G.A.: A Unified Approach to Global Program Optimization. In: Symposium on Principles of Programming Languages. pp. 194–206. ACM, New York, USA (1973)
8. Kleinsorge, J.C., Falk, H., Marwedel, P.: Simple Analysis of Partial Worst-Case Execution Paths on General Control Flow Graphs. In: Proceedings of the International Conference on Embedded Software. pp. 1–10 (September 2013)
9. Knoop, J., Steffen, B., Vollmer, J.: Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs. ACM Trans. Program. Lang. Syst. 18(3), 268–299 (May 1996)

10. Li, Y., Suhendra, V., Liang, Y., Mitra, T., Roychoudhury, A.: Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. In: IEEE Real-Time Systems Symposium. pp. 57–67. IEEE Computer Society, Washington, USA (2009)
11. Li, Y.T.S., Malik, S.: Performance Analysis of Embedded Software Using Implicit Path Enumeration. In: Proceedings of the Annual ACM/IEEE Design Automation Conference. pp. 456–461. ACM, New York, USA (1995)
12. Mittermayr, R., Blieberger, J.: Timing Analysis of Concurrent Programs. In: International Workshop on Worst-Case Execution Time Analysis. pp. 59–68 (2012)
13. Potop-Butucaru, D., Puaut, I.: Integrated Worst-Case Execution Time Estimation of Multicore Applications. In: Maiza, C. (ed.) International Workshop on Worst-Case Execution Time Analysis. pp. 21–31. Dagstuhl, Germany (2013)
14. Schliecker, S., Negrean, M., Nicolescu, G., Paulin, P., Ernst, R.: Reliable Performance Analysis of a Multicore Multithreaded System-on-chip. In: International Conference on Hardware/Software Codesign and System Synthesis. pp. 161–166. ACM, New York, USA (2008)
15. Schranzhofer, A., Pellizzoni, R., Chen, J.J., Thiele, L., Caccamo, M.: Worst-Case Response Time Analysis of Resource Access Models in Multi-Core Systems. In: Design Automation Conference (2010)
16. Synopsys Inc.: CoMET System Engineering IDE. `http://www.synopsys.com`
17. Taylor, R.N.: A General-purpose Algorithm for Analyzing Concurrent Programs. Communications of the ACM 26(5), 361–376 (May 1983)
18. Valmari, A.: Eliminating redundant interleavings during concurrent program verification. In: Odijk, E., Rem, M., Syre, J.C. (eds.) Parallel Architectures and Languages Europe, Lecture Notes in Computer Science, vol. 366, pp. 89–103. Springer Berlin Heidelberg (1989)
19. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. ACM Trans. Embed. Comput. Syst. 7(3) (2008)