Subgraph-Based Refinement of Worst-Case Execution Time Bounds

Florian Brandner
Computer Science and System Engineering Department
ENSTA-ParisTech
florian.brandner@ensta-paristech.fr

Alexander Jordan
Department of Applied Mathematics and Computer Science
Technical University of Denmark
alejo@dtu.dk

As real-time systems increase in complexity to provide more and more functionality and perform more demanding computations, the problem of statically analyzing the Worst-Case Execution Time bound (WCET) of real-time programs is becoming more and more time-consuming and imprecise.

The problem stems from the fact that with increasing program size also the number of potentially relevant program and hardware states to be considered during the WCET analysis increases. However, only a relatively small portion of the program actually contributes to the final WCET bound. Large parts of the program are thus irrelevant and are analyzed in vain. In the best case this only leads to increased analysis time. Very often, however, the analysis of irrelevant program parts interferes with the analysis of those program parts that turn out to be relevant.

We explore a novel technique based on *graph pruning* that promises to reduce the analysis overhead and, at the same time, increase the analysis' precision. The basic idea is to eliminate those program parts from the analysis problem that are known to be irrelevant for the final WCET bound. This reduces the analysis overhead, since only a subset of the program and hardware states have to be tracked. Consequently, more aggressive analysis techniques can be applied to the smaller problem, effectively reducing the overestimation of the WCET. As a side-effect, interference from irrelevant program parts are eliminated, e.g., on addresses of memory accesses, on loop bounds, or on the cache or processor state.

First experiments using a commercial WCET analysis tool show that our approach is feasible in practice and leads to reductions of up to 6% when a standard IPET approach is used for the analysis.

1 Introduction

Foremost size and complexity of software are causing the analysis overhead to grow rapidly, as the number of potential states of the program under analysis increases. For WCET analysis this effect is amplified since in order to arrive at a safe WCET bound, hardware states need to be considered as well, adding to the number of potential software states. Even when only small portions of a complex software program are relevant for the final WCET estimation, the analysis, has to account for *all* of the program's code to derive a safe bound. Except for trivial analysis problems, this inevitably reduces the precision of the WCET analysis, as irrelevant code parts interfere with the analysis of relevant code parts and lead to unnecessary overestimation of the determined WCET bound compared to the actual worst-case behavior.

Previous work is able to eliminate instructions irrelevant to flow analysis (program slicing) and refine the WCET bound by identifying infeasible paths (see Section 5 for details). With Iterative Graph Pruning (`IGP`), described in detail in the following sections, we also aim to limit WCET analysis to relevant parts of a program, but do so on the lower CFG-level. Based on *criticality*, basic blocks are grouped into sets according to the length of their respective paths. The sets are then processed iteratively by decreasing path length. During each iteration a subgraph of the original CFG is formed by unifying the subgraph of the previous iteration with the basic blocks from the currently considered set. A potentially more advanced WCET analysis is then applied to the program represented by the new subgraph. The algorithm terminates, with a possibly refined WCET estimate, as soon as a safe bound, valid for the original program, has been reached.

The benefits from this approach are that (1) the analysis problems defined by the subgraphs at each iteration are much smaller than the original analysis problem. This promises to reduce the analysis overhead, while still providing tight bounds. Furthermore, (2) processing the sets of basic blocks according to their decreasing path lengths, eliminates interference from other basic blocks, whose longest paths are known to be shorter. This improves the precision of the WCET analysis precisely for those code parts of the real-time program that impact the WCET estimate the most.

The main contributions of this paper are as follows:

- We present a novel WCET analysis technique based on graph pruning that focuses the analysis effort on relevant code parts of the real-time program.
- Due to reduced analysis overhead, more elaborate analysis techniques can be applied to the smaller sub-programs, leading to improved precision.
- We evaluate our approach using a commercial off-the-shelf WCET analysis tool and demonstrate considerable improvements of WCET bounds.

The remainder of this paper is structured as follows. We first give some background and motivation in Section 2. We then describe our novel graph pruning technique and follow its main steps on a realistic example in Section 3. Section 4 presents a detailed evaluation of our approach on well-established real-time benchmarks and a realistic processor architecture. Related work is covered in Section 5 before concluding in Section 6.

2 Background and Motivation

This section covers some basic definitions of control-flow graphs, paths, and WCET analysis, followed by a brief discussion of recent findings motivating this work.

2.1 Background

We assume that static WCET analysis proceeds in two phases as proposed by Theiling et al. [17]. Local worst-case execution times of individual basic blocks and control-flow edges are computed first, followed by a longest path search over a weighted CFG, where the weights are given by the local worst-case execution times.

Weighted Control-Flow Graph A weighted *Control-Flow Graph* (CFG) is a tuple $G = (V, E, r, t, \mathcal{W})$, where V is a set of nodes representing basic blocks that are connected by control-flow edges in E. We assume that every CFG contains a root node r and sink node t. The function $\mathcal{W}\colon\ V\ \cup\ E \rightarrow \mathbb{R}$ associates the basic blocks and edges with a weight.

Longest Path An ordered sequence of nodes $(v_1, \ldots, v_n)$ such that for $0 < i < n$ all edges (v_i, v_{i+1}) are in E, is called a path. The length of a path $|p|$ is given by the sum of all its node and edge weights: $\sum_{0<i\leq n} \mathcal{W}(v_i) + \sum_{0<i<n} \mathcal{W}((v_i, v_{i+1}))$. A path p is a longest path, when there exists no other path q such that $|p| < |q|$.

Local Execution Times The weight of individual CFG nodes and edges is usually computed using program analysis techniques that derive information on potential program and processor states at all relevant program points. This information is then combined to compute an upper bound of the local execution time of each basic block and CFG edge.

Longest Path Search From the local execution times a weighted CFG is constructed and a longest path search is performed to find the global WCET. This problem can be solved via *Integer Linear Programming* (ILP) using the *Implicit Path Enumeration Technique* (IPET) [10, 13]. Here, ILP variables represent execution counts of basic blocks and constraints define legal execution paths in the CFG. A standard ILP solver then determines the WCET. Alternatively dynamic programming [4] allows to directly compute the longest path on acyclic CFGs. Cyclic CFGs can be handled by applying this technique to acyclic subgraphs [15].

Worst-Case Execution Path A *Worst-Case Execution Path* (WCEP) is a longest path in the weighted CFG as computed by the longest path search.

Worst-Case Execution Time The WCET of a program corresponds to the length of the WCEP as computed by the longest path search. It is generally not feasible to compute the actual WCET, we thus usually seek a (tight) estimation, the WCET bound. For the purpose of brevity, we will use the term *WCET* to refer to the *WCET bound*.

2.2 Motivation

Recent work [2, 3] proposes a technique to *profile* the worst-case in real-time programs using static program analysis. The authors define the Criticality metric for each basic

		Criticality Intervals*					
Problem	**BBs**	I_0	I_1	I_2	I_3	I_4	I_5
debie-1	83	4	2	0	13	19	45
debie-2a	23	16	0	0	0	0	7
debie-2b	23	1	0	0	6	1	15
debie-2c	23	8	0	0	0	1	14
debie-3a	74	16	0	0	0	1	57
debie-3b	74	15	0	0	0	0	59
debie-3c	74	15	0	0	0	0	59
debie-4a	285	31	192	0	19	3	40
debie-4b	285	236	3	14	0	3	29
debie-4c	285	260	0	4	0	5	16
debie-4d	285	264	0	4	0	1	16
debie-5a	138	13	0	0	1	4	120
debie-5b	138	5	0	0	0	1	132
debie-6a	376	53	24	2	105	0	192
debie-6b	376	52	22	4	106	0	192
debie-6c	376	52	22	143	4	0	155
debie-6d	376	12	24	2	0	144	194

*Intervals: $0 \leq I_0 < 0.25 < I_1 < 0.5 < I_2 < 0.75 < I_3 < 0.9 < I_4 < 0.99 < I_5 \leq 1$

Table 1: Criticality statistics for the Debie1 benchmark

		Criticality Intervals*					
Problem	**BBs**	I_0	I_1	I_2	I_3	I_4	I_5
papa-a1	626	45	0	0	0	271	310
papa-a2a	1522	535	0	0	0	19	968
papa-a2b	1522	66	125	0	220	8	1103
papa-a3a	981	717	0	2	0	59	203
papa-a3b	981	201	0	0	2	89	689
papa-a4	334	56	0	0	0	54	224
papa-a5	438	0	0	0	2	10	426
papa-a6	682	25	0	4	42	81	530
papa-f1a	285	1	0	0	0	122	162
papa-f1b	285	279	0	0	0	0	6
papa-f2	8	0	0	0	3	1	4

*Intervals: $0 \leq I_0 < 0.25 < I_1 < 0.5 < I_2 < 0.75 < I_3 < 0.9 < I_4 < 0.99 < I_5 \leq 1$

Table 2: Criticality statistics for PapaBench

block in the CFG as the length of the longest path passing through a basic block divided by the global WCET. This gives a value between 0 and 1 that indicates how relevant a given basic block is in relation to the WCET of the program. This information can then be used to guide program optimizations in order to improve the WCET of the program.

However, the metric is not only interesting for program optimization. In particular the distribution of the Criticality values in a real-time program is highly interesting. Table 1 and 2 show profiling results of several WCET analysis problems for the Debie1 and PapaBench real-time programs. From the columns $I_0, ..., I_5$, the number of basic blocks in six predefined Criticality intervals can be seen. The variation of block distributions hints at an underlying difference of program structure with regard to WCET-critical code. A surprisingly large number of basic blocks is relatively unimportant with a Criticality value below 0.8. Only 13% of the basic blocks of the `debie-4a` benchmark, for instance, are highly critical. Over all analysis problems of Debie1 only 54% of the basic blocks have a Criticality above 0.80. Considering all analysis problems, in the mean 67% of the basic blocks are highly critical, i.e., have a Criticality above 0.8.

Considering this observation, one could ask whether it would be possible to improve the precision and computation time of a WCET analysis by excluding the uncritical code parts from the analysis? This is precisely the goal of this work. We exclude uncritical code parts by pruning the control-flow graph and iteratively deriving a refined, but still provably correct, WCET bound. A detailed description of this approach follows in the next section.

3 Algorithm

In a nutshell, iterative graph pruning (`IGP`), presented as pseudo code in Algorithm 1, performs WCET analysis by iteratively merging a sequence of basic block sets (S_i), terminating when an upper bound (ub_{wcet}) is found to be a safe bound for the input program.

The blocks are then inserted into sets S_i based on the path length. The respective path length of a set can later be retrieved by $longestpath(S_i)$. The sets S_i are generated by computing the longest path going through every basic block within G [2, 3]. At every iteration i, the vertex-induced subgraph G' is created from the union of the i first (and most critical) sets S_i (l. 7 – l. 8). G' is then targeted by a full WCET analysis run (`WCEToverAny` l. 13), which entails abstract interpretation on the program's subgraph G' to generate the weighting function $\mathcal{W}'$, followed by a longest path search. `WCEToverAny`, additionally, forces the longest path search to only consider those paths passing through blocks in the current S_i. All other paths in G' are uninteresting, since these paths have already been bounded in the previous iterations. Note that G' may not contain any such path that is feasible. `WCEToverAny` then returns 0 and the current upper bound remains unchanged. If, at the end of an iteration, the current upper bound (ub_{wcet}) is less than the just computed bound (WCET$_i$), it needs to be updated (l. 10).

The algorithm terminates at the latest when all sets have been considered (i.e., $V' = V$ and G' is the same as the graph of the original program) or before when the termination

Algorithm 1 Graph-pruning algorithm

Require: $G = (V, E, r, t)$ CFG of the input program.
$S_1, \ldots, S_n$ Block-sets sorted by longest path length, from highest to lowest.

```
1: ub_wcet = 0                         ▷ Will increase until a safe WCET bound
2: for i = 1 → n do
3:     ▷ Terminate when no longer paths can exist
4:     if ub_wcet ≥ longestpath(S_i) then
5:         return ub_wcet
6:     ▷ Construct and analyze a subgraph
7:     Let V' ← S_1 ∪ ··· ∪ S_i, E' ← E ∩ V' × V' in
8:         G' ← (V', E', r, t)
9:         WCET_i ← CalcPrunedWCET(G', S_i, ub_wcet)
10:        ub_wcet ← max(WCET_i, ub_wcet)
11: return ub_wcet
12: function CalcPrunedWCET(G', S_i, ub_wcet)
13:     return WCEToverAny(G', S_i)
```

condition (l. 4) is met. The latter case occurs when the remaining longest path induced by the remaining S_is is shorter than the current WCET bound ub_{wcet}. We will prove that ub_{wcet} is a valid bound for G in the next section.

Example 1. *Consider the CFG shown in Figure 1, where weights are shown for each basic block along with the longest paths passing through the respective blocks. Furthermore, assume that the number of loop iterations at* `BB4` *depends on a variable* `x` *either assigned to* 10 *in* `BB1` *or* 7 *in* `BB2`.

A first, unmodified WCET analysis discovers both assignments to `x`. *Thus, it derives a loop bound of* 10 *iterations for* `BB4`. *The resulting longest path* (p_1) *has length* 60 *and covers* r, `BB0`, `BB2`, `BB4`, `BB5` *and* t. *These blocks are thus assigned to the basic block set* S_0. *The second longest path passing through a node not covered by* p_1 *is* p_2, *with length* 56. *Since all blocks on* p_1 *are already assigned to* S_0, *the only new block of* p_2 *is assigned to* $S_1 = \{$`BB1`$\}$. *The same applies to path* p_3 *and block set* $S_3 = \{$`BB3`$\}$.

Our Algorithm starts by reanalyzing subgraph G_1 *induced by* S_0 *shown by Figure 2. The analysis now discovers that there is only a single assignment to* `x`, *limiting the number of loop iterations at* `BB4` *to* 7. *In addition, a more precise estimation of the local execution time within the loop is derived, i.e.,* $\mathcal{W}($`BB4`$)$ *is now* 4 *instead of* 5. *The initial length of* 60 *for* p_1 *alone was heavily overestimated, since the longest path in* G_1 *is much shorter. This results in an upper bound* ub_{wcet} *of* 38. *The algorithm continues to iterate at this point since the path length associated with* S_1 *is longer than the current upper bound. The second iteration leads to the construction of* G_2 *induced by* $S_0 \cup S_1$ *(see Figure 2). A*

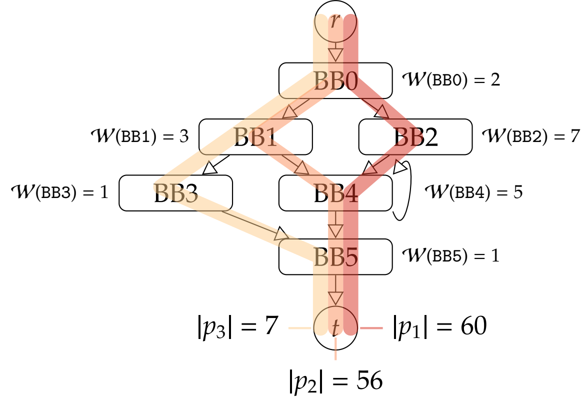

Figure 1: A weighted CFG and its longest paths (see Example 1)

new WCET analysis run is forced to consider only those paths passing through `BB1` *and finds a loop bound of* 10 *iterations. This results in an upper bound* ub_{wcet} *of* 56*. Since the upper bound now represents a path longer than any that could be uncovered by including the next block set* S_3*, the algorithm terminates with a more precise WCET bound of* 56 *instead of the initial* 60*.*

3.1 Correctness

To show the correctness of our approach, we have to consider the impact of graph pruning on the typical phases of a WCET analysis run. We assume, without loss of generality, that WCET analysis is performed in two phases [17]. A first phase, based on abstract interpretation [6], delivers local worst-case execution times for each basic block. In the second phase, a longest path search [10, 13] is performed on a weighted CFG, computed from these local execution times.

In our approach, abstract interpretation is applied to a subgraph G' of the original

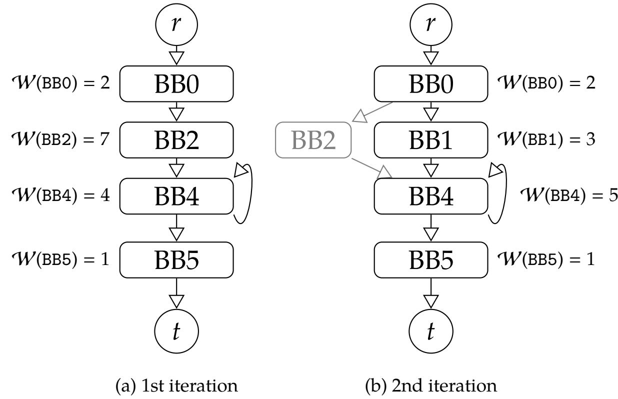

Figure 2: Weighted CFGs formed during Iterative Graph Pruning.

CFG G of the input program. In order to show correctness we thus have to investigate some properties of the subgraph G'.

Lemma 1. *A subgraph $G' = (V', E', r, t)$ constructed by Algorithm 1 is connected, i.e., for every CFG node $v' \in V'$ a path from r to t, passing through v', exists.*

Proof. This follows immediately from the way subgraphs are constructed. Remember that the blocks in the subgraph G' of the m-th iteration are computed by unifying all the basic block sets S_i, $i \in \{0...m\}$, whose path lengths are longer than the path length associated with S_m, i.e., $V' = \bigcup_{i \in 0..m} S_i$.

Assume G' is not connected, i.e., a CFG node v' exists that is not reachable from the root node r in G'. Since $v' \in V'$ it follows that a corresponding path $p = (r, ..., n', ..., t)$ has to exists in the original graph G. The length of this path corresponds to the path length associated with S_m. As G' is not connected, at least one node of p is not in V'. However, this is impossible, since the existence of p implies that this node either is in S_m or another set S_k, $k < m$. The subgraph G' is thus connected. □

Lemma 2. *Abstract interpretation applied to a subgraph G' delivers correct results with respect to the potential execution paths through G'.*

Proof (sketch). A sound analysis based on abstract interpretation delivers sound results with regard to potential executions of a program, e.g., represented by its CFG G. When applying the same analysis to a subgraph G', the set of potential executions is reduced (excluding those executions that do not lead to the sink node t). Since the analysis is sound, this approximation trivially remains sound with regard to the executions represented by G'. Note, however, that the analysis result is not guaranteed to be sound with regard to all executions of the original program. □

The previous two lemmas ensure that, independent of the concrete analysis performed, the local execution times obtained by abstract interpretation in the WCET analysis tool are sound with respect to a subgraph G'. It remains to show that the WCET bound computed during the longest path search by Algorithm 1 is a safe bound.

Lemma 3. *The worst-case execution time bound computed by Algorithm 1 for a subgraph G' is safe.*

Proof. Consider the subgraph G' and its basic block set S_m of the m-th iteration as well as the subgraph G'', with its bound $\text{WCET}(G'')$, of the previous iteration.

First, assume that the longest path p through G' represents a valid path in G'', i.e., p does not contain any node in S_m. In relation to the previous iteration, the current WCET estimate for p might increase, due to overestimation of the local worst-case execution times, i.e. $\text{WCET}(G'') \leq \text{WCET}(G')$. However, irrespective of the WCET computed for G', the bound established by the previous iteration for G'' still holds. Paths of this structure are thus uninteresting, as the length of all paths in G'' has been bounded by the previous iteration. It remains to bound those paths in G' that do not have a corresponding path in G''. The longest path search thus only needs to consider paths

containing at least one node in S_m.

Three cases for the longest path p need to be considered:

1. $|p| > \text{WCET}(G'')$:
Since $|p|$ is longer than the previously established WCET bound, it follows that $\text{WCET}(G') = |p|$.

2. $|p| \leq \text{WCET}(G'')$:
As the length of p is not longer than the previously established bound, it follows that $\text{WCET}(G') = \text{WCET}(G'')$. The longest path through G'' also represents the longest path through G', ignoring any additional overestimation caused by blocks in S_m.

3. No feasible path containing a block in S_m exists:
This case happens when the abstract interpretation finds that no execution in G' exists that passes through a block in S_m, i.e., none of the conditions of the branches leading to a block in S_m can be satisfied. It follows that $\text{WCET}(G') = \text{WCET}(G'')$. Note that paths over these blocks might become feasible in later iterations, e.g., when code making the, yet unsatisfiable, conditions satisfiable is added.

Using induction, we can finally prove that Algorithm 1 delivers safe WCET bounds for all subgraphs considered during the iterative processing. □

Theorem 1. *Algorithm 1 computes a safe WCET bound.*

Proof. This follows immediately from the previous lemmas and the termination condition of the algorithm.

The lemmas prove that applying abstract interpretation on subgraphs is sound and that the algorithm computes safe bounds with respect to the subgraphs considered during the iterative processing.

It remains to show that no other paths exist, which could be longer than the WCET bound delivered by the last iteration. This is guaranteed by the order in which the sets of blocks are processed and the termination condition (Algorithm 1, l. 4). Together these ensure that all basic blocks not yet considered by the iterative refinement can only induce paths that are shorter than the computed bound. □

3.2 Complexity

The number of sets S_i is an upper bound on the iterations that will be performed by `IGP`. The former is again bounded by the number of basic blocks in the input program. Since its iterations are linear in the number of blocks, `IGP` is dominated by the complexity of the WCET analysis, i.e., abstract interpretation and longest path search.

The sets S_i, which are assumed as input in Algorithm 1, can be efficiently computed using the criticality algorithms [2, 3]. This may be performed either during a preprocessing step, or on demand, while the graph pruning algorithm iterates.

Algorithm 2 WCET computation function CalcPrunedWCET using two-stage analysis

Require: $G' = (V', E', r, t) \ldots$ A CFG.
$S_i \ldots$ The set of newly added blocks.
$ub_{wcet} \ldots$ The current upper bound of the WCET.

1: $\text{WCET}_i \leftarrow$ `WCEToverAnyFast`(G', S_i)
2: **if** $\text{WCET}_i > ub_{wcet}$ **then**
3: $\quad \text{WCET}'_i \leftarrow$ `WCEToverAnyPrecise`(G', S_i)
4: $\quad$ **return** WCET'_i
5: **return** WCET_i

3.3 Algorithm Variants

It may be the case that the WCET analysis tool targeted by graph pruning, can be configured for different levels of precision. This usually involves a trade-off between tightness (precision) of the WCET bound and longer analysis runtime. `IGP` can be used to incorporate analyses with different precision. Running the higher-precision analysis on a previously pruned graph would be a straight-forward way of further improving the WCET bound. But Algorithm 1 can also be modified to make use of multiple levels of precision directly. To do this, we replace the CalcPrunedWCET function in Algorithm 1 with the variant given in Algorithm 2. This algorithm performs a second, more precise WCET analysis (`WCEToverAnyPrecise`) to lower the estimate of WCET_i for the current graph G', whenever the imprecise analysis (`WCEToverAnyFast`) would increase the overall WCET bound.

Another valid, but trivial way of incorporating a higher-precision analysis into graph pruning, would be to run `WCEToverAnyPrecise` on the pruned subgraph exactly once after the iterative processing. This would reduce the computational overhead and further lower the WCET bound (down to the path length of the next block set). One could even avoid the iterative processing entirely, by heuristically constructing a subgraph and applying the precise analysis to this subgraph. This would foremost reduce the computational overhead and leave the burden of reducing overestimation on the precise WCET analysis.

4 Evaluation

We start by giving a complete example of running iterative graph pruning on the first problem from the Debie1 benchmarks, which has no loops and 83 basic blocks. We then extend our evaluation to a number of WCET benchmarks using standard Iterative Graph Pruning (`IGP`) and its two-stage variant (`IGP-TS`).

4.1 Case Study: `debie-1`

We start by analyzing the original program represented by its CFG G. This yields a first valid global WCET bound in cycles (denoted by $\text{WCET}(G)$).

Set	S_i	$longestpath(S_i)$
S_0	43	1,621
S_1	1	1,613
S_2	1	1,608
S_3	5	1,601
S_4	1	1,592
S_5	4	1,585
S_6	8	1,560
S_7	1	1,472
S_8	3	1,456
S_9	3	1,415
S_{10}	3	1,352
S_{11}	3	1,324
S_{12}	1	1,231
S_{13}	2	657
S_{14}	1	321

Table 3: Criticality based basic block sets for `debie-1`

Pre-run: $\text{WCET}(G) = 1621$

At the same time we can perform a basic-block-level WCET profiling according to the definition of *criticality*. This yields, for every block b in the CFG G, the maximum length (a value in the range $[0, \text{WCET}(G)]$) of all paths going through b. Blocks with a value close to the global WCET are considered critical, while those close to 0 are uncritical. Based on their longest paths, we now insert blocks into several sets, so that blocks with the same path length end up in the same set. In our example 15 such sets exist, their path length ($longestpath(S_i)$) and cardinality is given in Table 3.

The pruning algorithm kicks off with a WCET analysis, but this time considers only the blocks on the global WCEP, i.e., the blocks in S_0. Using these blocks a vertex-induced subgraph G_1 is created and its WCET bound is analyzed.

Iteration 1: $G_1 = S_0$ $\text{WCET}(G_1) = 334$

$\text{WCET}(G_1)$ is drastically lower than the original WCET, even though all blocks of the original WCEP are contained in G_1. Using abstract interpretation the WCET tool has derived that the path is actually infeasible, i.e., no legal execution for the path exists. It is necessary to add more blocks back to the program at this stage. And since we suspect that those blocks with longer paths have a larger impact on the attainable WCET bound, we select the most critical blocks available, namely S_1.

Iteration 2: $G_2 = G_1 \cup S_1$ $\text{WCET}(G_2) = 334$
Iteration 3: $G_3 = G_2 \cup S_2$ $\text{WCET}(G_3) = 334$
Iteration 4: $G_4 = G_3 \cup S_3$ $\text{WCET}(G_4) = 1423$

Adding S_1 to the graph did in fact not change the current WCET bound, neither did

S_2. But after adding S_3, a considerably longer path becomes feasible and our current WCET bound jumps to 1423 cycles. Let us pause for a moment and consider what we have done so far. G_4 contains the most critical code ($S_0 \cup \cdots \cup S_3$), it contains 42 basic blocks (compared to 80 for the whole program). Blocks from eleven sets have not yet been added and we have a current bound of WCET(G_4) < WCET(G). Is this already a valid WCET bound for the original problem? No, looking at the remaining sets, we can see there are still blocks in sets $S_4, \ldots, S_8$, which may be part of a longer path in an extended subgraph, i.e., $\mathit{longestpath}(S_i) \geq 1423,\ 4 \leq i \leq 8$. With this in mind, we resume adding blocks with S_4.

Iteration 5: $G_5 = G_4 \cup S_4$ $\quad$ WCET(G_5) = 1525
Iteration 6: $G_6 = G_5 \cup S_5$ $\quad$ WCET(G_6) = 1525
Iteration 7: $\underline{G_7} = G_6 \cup S_6$ $\quad$ WCET(G_7) = $\underline{1525}$

When we are about to add set S_7, we notice that in the worst case, it would uncover a path with length 1472. Since the WCET bound changed again in iteration 5, this is below the current value of 1525 and thus would have no impact. This signals termination for our algorithm. At this point, blocks from all sets are either already part of the current graph, or do not contain blocks which could enable a path longer than 1525 cycles. Our result is the pruned graph G_7, which is a WCET-bound-maintaining subgraph of the original program. By using this graph, we have reduced the initial WCET bound of the analysis by 96 cycles ($\sim 6\%$).

4.2 Setup for Experiments

We evaluate our approach using the commercial WCET analysis tool aiT (`a3` version 12.10i) and well-establisheded WCET benchmarks: The Mälardalen benchmark suite[1], Debie1 [8] (version e) and PapaBench [12] (version 0.4). The analysis problems for the latter two are taken from the WCET Tool Challenge 2011.

The prototype implementation of graph pruning that we use for this evaluation, treats the WCET analysis tool as a black-box and thus works with an unmodified version of aiT. The graph pruning is realized through scripting and the `AIS` annotations (`IS NEVER EXECUTED` and `FLOW`). This means that due to our setup, *all* information computed by an iteration is discarded and *not* reused by the following iteration. Thus, any analysis runtime results would be dramatically increased. We discuss this shortcoming of our evaluation in Section 4.5).

We apply the two graph pruning variants to an overall of 37 analysis problems and compare the attainable WCET bound with the original result of aiT. The first variant (`IGP`), performs iterative graph pruning using standard IPET (Algorithm 1). The second variant (`IGP-TS`), uses the two-stage WCET computation (Algorithm 1 with the extension in Algorithm 2). Here, the standard computation is potentially refined by a second, more precise, but also more expensive, WCET analysis using aiT's *prediction file* technique [16].

We additionally examine properties of the analysis problem that have a direct influence on overestimation. These are (1) hardware splits, a measure for the amount of dupli-

[1] `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

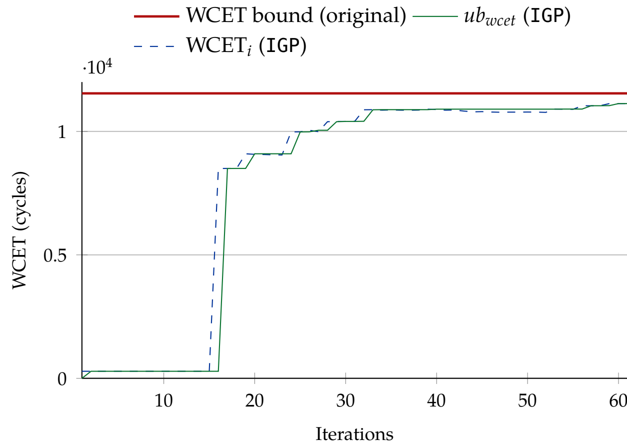

Figure 3: WCET bounds per iteration for the `f1a` benchmark using Iterative Graph Pruning. (`IGP`, lower is better)

cated states in the presence of unpredictable hardware behavior (i.e., caches, branch prediction), and (2) the size of the analysis problems at different stages (i.e., subgraph size).

4.3 Iterative Graph Pruning

Table 4 summarizes the WCET bound improvement and aggregates it by benchmark suite. The selected Mälardalen programs, especially the ones representing algorithmic cores with almost single-path execution behavior, not surprisingly do not benefit from graph pruning. Their WCET bound improvement is 2% at best. However, the larger application-like programs, Debie1 and PapaBench, benefit from graph pruning and some of their analysis problems have their WCET bound reduced significantly. When we look at `f1a` from PapaBench in detail, we can see in Figure 3 that within the first 20 iterations, the upper bound (ub_{wcet}) leaps to a level close to its final value. At this point, the most critical basic block sets have been added to the subgraph and WCET analysis returned a good candidate for the actual global WCEP. Step sizes subsequently decrease and from iteration 35 on, the upper bound (ub_{wcet}) roughly holds while more blocks are added to the `IGP` subgraph (see Figure 4).

Another group of problem instances does not exhibit properties that are exploitable by graph pruning. These can be identified in Table 5 by their low number of iterations: after one or two iterations, `IGP` terminates since all (feasible) blocks are either on the global WCEP, or there is no interference between WCET-critical and unrelated code.

There is also a third class of benchmark problems, which exhibits a high number of iterations without a significant improvement of the WCET bound. Studying the most severe cases (problems `a2a` and `a2b` from PapaBench), we have found that almost all of the WCEPs found in subgraphs are infeasible on their own (i.e., their feasibility depends on blocks in other block sets). This is likely caused by a particular program structure

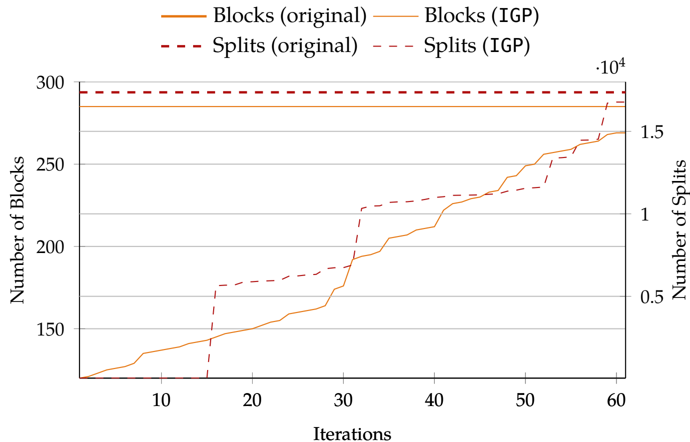

Figure 4: Subgraph size and hardware splits per iteration for the `f1a` benchmark using Iterative Graph Pruning. (`IGP`, lower is better)

Suite	Contains	Algo.	Improved	WCET Refinement *max*	*min*	*mean*
debie	16	`IGP`	12	−6.43%	0%	−1.74%
mdh	11	`IGP`	7	−2.15%	0%	−0.58%
papa	10	`IGP`	6	−3.73%	0%	−1.95%
debie	16	`IGP-TS`	10	−5.34%	0%	−0.82%
mdh	11	`IGP-TS`	5	−1.28%	0%	−0.26%
papa	10	`IGP-TS`	4	−2.83%	0%	−1.03%

Table 4: WCET Reduction using Iterative Graph Pruning (Summary)

and the WCET analysis failing to exclude infeasible paths from the longest path search (*flow facts*).

For all non-trivial benchmarks (two `IGP` iterations or more), we see that the minimal (and average) reduction of hardware splits is high (see Figure 5). Average subgraph sizes (Figure 6) are likewise significantly smaller than their originals. This tells us that overestimation is being effectively addressed by `IGP`. At the same time it can benefit from analysis problems, which are roughly half the original size. In Table 5 we present the detailed results from graph pruning using the `IGP` algorithm. It contains the number of iterations in column "Iter.", the number of unique WCET paths encountered in column "$|\mathcal{P}|$", and a comparison of graph sizes before and after pruning in the last two columns. Note that the number of infeasible blocks is contained in $|V|$ and will always be pruned. `IGP` improves WCET bounds up to 6% compared to aiT's result on the original program. The average improvement among non-trivial benchmark problems is 2%.

Benchmark	WCET (cycles) Original	IGP	Refinement	Iterations	$\lvert\mathcal{P}\rvert$	Graph size (nodes) $\lvert V\rvert$	$\lvert V'\rvert$
debie-1	1,621	1,525	−5.92%	7	6	83	63
debie-2b	566	534	−5.65%	2	2	23	16
debie-2c	489	489	0.00%	1	1	23	14
debie-3a	5,094	5,047	−0.92%	5	5	74	57
debie-3b	28,515	28,451	−0.22%	8	9	74	59
debie-3c	29,227	29,163	−0.22%	8	9	74	59
debie-4a	4,913	4,843	−1.43%	4	4	285	43
debie-4b	1,789	1,674	−6.43%	4	5	285	32
debie-4c	910	891	−2.09%	2	2	285	17
debie-4d	968	968	0.00%	1	1	285	16
debie-5a	6,119	6,047	−1.18%	6	6	138	120
debie-5b	112,538	112,467	−0.06%	7	7	138	130
debie-6a	44,273	44,007	−0.60%	12	10	376	192
debie-6b	44,273	44,007	−0.60%	12	10	376	192
debie-6c	63,133	62,145	−1.57%	13	10	376	155
debie-6d	46,337	46,049	−0.62%	12	10	376	194
mdh-compress	26,697	26,601	−0.36%	8	8	92	84
mdh-expint	793,236	785,573	−0.97%	3	3	25	19
mdh-fft1	651,767	651,249	−0.08%	98	77	491	455
mdh-lcdnum	4,162	4,162	0.00%	1	2	22	16
mdh-ludcmp	973,441	971,502	−0.20%	104	104	430	423
mdh-minver	298,523	292,108	−2.15%	103	104	466	457
mdh-prime	194,136	194,055	−0.04%	3	3	23	22
mdh-qurt	649,934	649,434	−0.08%	101	98	423	408
mdh-select	261,214	260,317	−0.34%	11	12	99	93
mdh-sqrt	219,363	219,074	−0.13%	98	99	518	449
mdh-statemate	24,145	23,657	−2.02%	70	71	420	363
papa-a1	8,551	8,247	−3.56%	101	92	626	484
papa-a2a	81,656	81,187	−0.57%	177	146	1,522	944
papa-a2b	100,939	100,190	−0.74%	199	163	1,522	1,079
papa-a3a	9,138	8,922	−2.36%	65	64	981	255
papa-a3b	28,193	27,691	−1.78%	146	139	981	746
papa-a4	11,104	10,879	−2.03%	44	38	334	253
papa-a5	20,545	20,320	−1.10%	96	97	438	426
papa-a6	29,092	28,006	−3.73%	118	114	682	608
papa-f1a	11,542	11,128	−3.59%	60	47	285	269
papa-f2	237	237	0.00%	1	1	8	4

Table 5: Detailed results for Iterative Graph Pruning (IGP)

Benchmark	WCET (cycles) Original	IGP-TS	Refinement	Iterations	$\|\mathcal{P}\|$	Graph size (nodes) $\|V\|$	$\|V'\|$
debie-1	1,507	1,479	−1.86%	7	9	83	63
debie-2b	520	520	0.00%	2	2	23	16
debie-2c	474	474	0.00%	2	2	23	15
debie-3a	4,787	4,735	−1.09%	6	10	74	58
debie-3b	24,756	24,742	−0.06%	8	15	74	59
debie-3c	25,348	25,334	−0.06%	8	15	74	59
debie-4a	4,810	4,783	−0.56%	4	5	285	43
debie-4b	1,630	1,543	−5.34%	4	5	285	32
debie-4c	877	877	0.00%	4	4	285	20
debie-4d	957	957	0.00%	1	1	285	16
debie-5a	5,784	5,753	−0.54%	8	15	138	123
debie-5b	78,338	78,307	−0.04%	9	18	138	133
debie-6a	42,945	42,680	−0.62%	12	19	376	192
debie-6b	42,945	42,680	−0.62%	12	19	376	192
debie-6c	61,042	60,129	−1.50%	13	14	376	155
debie-6d	44,677	44,390	−0.64%	12	19	376	194
mdh-compress	26,344	26,297	−0.18%	8	15	92	84
mdh-expint	792,298	784,693	−0.96%	3	3	25	19
mdh-fft1	589,137	588,871	−0.05%	103	163	491	468
mdh-lcdnum	4,140	4,140	0.00%	2	5	22	21
mdh-ludcmp	899,460	899,236	−0.02%	105	209	430	426
mdh-minver	275,957	275,805	−0.06%	104	209	466	458
mdh-prime	194,053	194,031	−0.01%	3	4	23	22
mdh-qurt	595,432	595,085	−0.06%	104	200	423	416
mdh-select	222,750	222,215	−0.24%	12	25	99	99
mdh-sqrt	194,497	194,471	−0.02%	127	255	518	503
mdh-statemate	22,173	21,890	−1.28%	98	195	420	409
papa-a1	7,050	7,049	−0.01%	141	267	626	581
papa-a2a	68,641	68,340	−0.44%	200	337	1,522	987
papa-a2b	84,785	84,280	−0.60%	210	347	1,522	1,111
papa-a3a	8,488	8,347	−1.66%	70	136	981	262
papa-a3b	24,063	23,984	−0.33%	164	312	981	780
papa-a4	9,473	9,288	−1.95%	57	100	334	278
papa-a5	19,020	18,965	−0.29%	104	205	438	434
papa-a6	25,045	24,507	−2.15%	122	236	682	635
papa-f1a	10,359	10,066	−2.83%	71	114	285	284
papa-f2	236	236	0.00%	1	1	8	4

Table 6: Detailed results for two-stage Iterative Graph Pruning (IGP-TS)

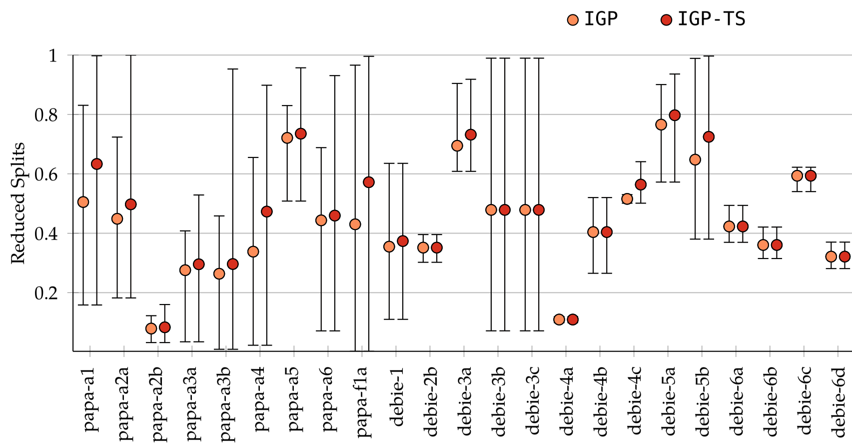

Figure 5: Reduction of hardware splits using Iterative Graph Pruning. (IGP vs. IGP-TS, lower is better. Vertical bars display iterations' min./max., marker is average.)

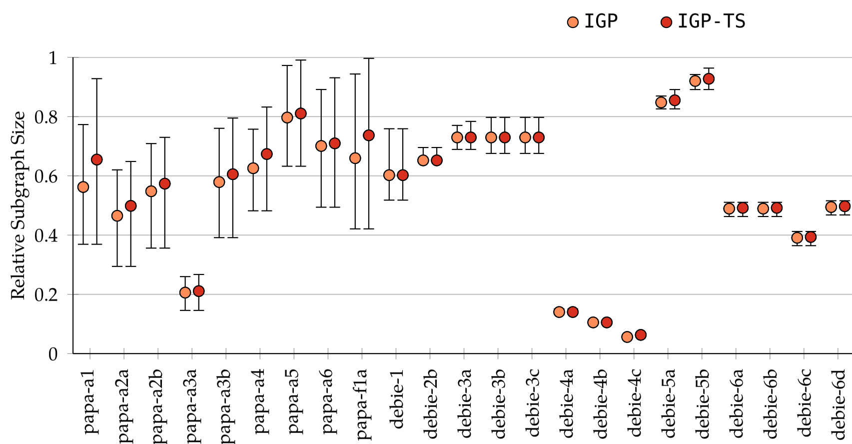

Figure 6: Subgraph sizes using Iterative Graph Pruning. (IGP vs. IGP-TS, lower is better)

4.4 Two-stage Iterative Graph Pruning

To evaluate the two-stage approach of iterative graph pruning (`IGP-TS`), we make use of the prediction file (PF) based IPET solver in aiT as a second stage analysis (i.e., for the `WCEToverAnyPrecise` invocation in Algorithm 2). When performing PF-based WCET computation, the timing information during longest path search is not restricted to a single WCET for each basic block, but may encompass multiple architectural states [16]. Tightening WCET bounds in this way comes at the cost of —depending on program size— much larger ILP problems and thus time needed for solving. (Note that resource demand, i.e., memory, for the larger problems of `papabench` on a related out-of-order PowerPC architecture is bordering on infeasibility.) Fast WCET analysis remains unchanged with regard to `IGP`.

The WCET bound improvement of `IGP-TS`, compared to the PF-enabled analysis as a baseline, behaves similar to that of `IGP`. Benchmark problems, for which the WCET bound was improved by `IGP`, also improve by `IGP-TS`, but the effect is less pronounced (see Table 4). We thus conclude that our approach is profitable even compared to a state-of-the-art WCET analysis tool using its most sophisticated analysis technique. How its profitability increases inversely proportional to the use of mechanisms that prevent overestimation.

The reduction of hardware splits and graph sizes measured over all iterations also behaves similar (compare `IGP` and `IGP-TS` in Figures 5 and 6), although it fails to "cut off" sources of overestimation in the same was as `IGP` does. Furthermore, as we expected, the number of unique WCEPs found by `IGP-TS` is higher (see $|\mathcal{P}|$ in Table 5 versus Table 6). This is due to the more precise analysis being used. For the same reason, we can see an increase in the number of iterations.

4.5 Discussion

We have evaluated graph pruning using a state-of-the-art, commercial WCET tool. aiT uses powerful abstract interpretation and is able to produce good WCET bounds on its own. Even so, graph pruning can eliminate sources of overestimation and significantly tighten the WCET bound. While we can configure aiT to analyze subgraphs and extract all results we need from it, our setup is only suitable as a proof-of-concept. The analysis tool is treated as a block box, which leads to needless overhead that could be avoided. We thus do not present detailed measurements of the analysis time here. However, even with these short-comings we observed an increase in analysis time by a factor of 9 on average (tests were performed on an AMD Opteron 8356 at 2.3 GHz, running Linux Kernel version 2.6, with CPLEX version 10 solving the IPET ILP problems).

We expect that most of the analysis overhead can, in fact, be eliminated by designing the WCET analysis to take advantage of the iterative processing. The overhead of performing a complete run of abstract interpretation on every iteration can, for instance, be avoided. Abstract interpretation usually is performed by searching for a fixed-point. Adding basic blocks, as done by our algorithm, can easily be handled by this approach. The fixed-point search can continue from the abstract states computed for the previous

iteration to quickly derive a new fixed-point for the current subgraph. Other forms of incremental analysis should equally reduce the overhead of performing a longest path search on structurally similar subgraphs. These techniques, combined with smaller problem sizes (due to smaller subgraphs and reduced hardware splits), promise to even reduce the analysis overhead, compared to a full analysis run using aiT's prediction file technique. We even observed this behavior in our tests for `IGP-TS` and the `a2b` benchmark. Despite an increase in the analysis time by a factor of 14 for the abstract interpretation, a reduction of the ILP solving time lead to an overall reduction of the analysis time of about 15%.

We observed that our technique addresses the problem of overestimation very well, in particular during early iterations. However, we also observed that in many cases the overestimation grew fast, often outweighing large initial gains. The main problem is that the subgraphs steadily grow larger. We could address this problem by restricting the subgraphs to only those nodes reachable from the current basic block set (S_i). However, one could similarly change the strategy for growing subgraphs, e.g., by estimating the impact on the number of hardware splits. In a similar way, neighboring basic block sets may be merged in order to avoid excessive iteration counts.

5 Related Work

We divide related approaches for tightening WCET bounds roughly by the main technique they employ. Since most methods —ours included— are complementary to each other and thus can be combined within an integrated analysis, some natural overlap occurs.

5.1 Program Slicing

Several pruning techniques, similar in spirit to our technique, have been proposed in the past based on program slicing [18]. The basic idea of program slicing is to improve the precision and the computational overhead of static program analyses by discarding *program statements* that are irrelevant to the goal of the analysis. Consider, for instance, the case when the goal of a static analysis is to determine the value of a given variable in a program. When forming a slice for that particular variable, only those statements are considered during the analysis that directly and indirectly contribute to the computation of that variable. All other statements are ignored. The goal in our approach is to improve the analysis of the WCET itself, our technique thus can be seen as a form of *program slicing on the timing domain*.

Sandberg et al. [14], for example, propose to use program slicing to improve the static analysis of flow facts. They construct program slices based on either all conditions of branches in a program, on all loop-exit conditions, or on the loop-exit conditions of a particular loop. Based on the computed slices, flow facts, such as loop bounds, are computed. In contrast to our work, the focus here is on deriving flow facts only, regardless of the relevance or impact to the final WCET. Their technique can, however, be combined

with our approach. This would, for instance, allow to derive flow facts that are only valid with respect to the current subgraph under consideration.

A similar approach is proposed by Lokuciejewski et al. [11]. They combine abstract interpretation, polytope models, and program slicing to derive precise loop bounds. The approach again does not consider the *relevance* of the respective loops under analysis with regard to the final WCET.

5.2 Infeasible-Path Elimination

Bang and Kim [1], similar to our technique, propose an iterative approach to refine the attainable WCET using standard IPET. The basic idea is to perform a regular WCET analysis run. The resulting WCEP is subsequently checked for feasibility and, in the case of an infeasible WCEP, additional constraints are added to the IPET problem to exclude the path. This process is repeated until a feasible path is encountered. It is important to note that the presented feasibility checks are conservative and only consider individual basic blocks and pairs or blocks, but not the entire path. The major problem of this approach is that the refinement is based on individual paths through the program, whose number is potentially exponential in the number of conditional branches in the program. The additional constraints are, furthermore, only applied during the final IPET run. Contrary to our approach, the technique thus cannot improve the precision of previous analysis phases, such as the cache- or pipeline analysis.

Zwirchmayer et al. [9] propose a related scheme called WCET Squeezing. The authors iteratively check the feasibility of the current WCEP using symbolic execution and exclude paths found to be infeasible from the IPET problem. In contrast to Bang and Kim, this technique considers the entire path and may thus potentially derive more complex constraints. The technique similarly does not allow to improve the precision of other analysis phases than the final IPET. To its advantage, WCET squeezing is an *anytime* algorithm, i.e., when interrupted at any time, a possible up-to-then achieved refinement is sound. Our current graph pruning algorithm does not have the anytime property, in fact it needs to run until the WCET bound from a pruned subgraph has been proved valid.

5.3 Abstract Interpretation

We evaluated graph pruning using AbsInt's aiT WCET analyzer, which makes use of abstract interpretation for its value analysis. While in this specific setting, the precision of value analysis benefits from the smaller graphs that we provide through pruning, context-sensitive abstract interpretation techniques [6, 7] themselves, in fact, share the same goal with us. They are used for the automatic computation of control-flow bounds (i.e. loop bounds or flow facts) that ultimately aim to tighten the WCET bound. The challenge for abstract interpretation is to cope with an overwhelming combination of program-, pipeline-, and cache states through safe approximation of values withing their domain and the merging of related hardware states. Efficient widening (and narrowing) operations are essential for analysis precision. [5]

The approach presented here is inspired by the Criticality metric proposed by Brandner et al. [2]. The metric assigns a numeric value in the interval $[0, 1]$ to each basic block of a real-time program, where values close to 0 indicate code that is irrelevant and values close to 1 indicate code that is critical to the final WCET. We adopt the idea of discovering the longest path passing through basic blocks to refine the attainable WCET bound. The authors' algorithm to compute Criticalities on-demand has been adopted for our approach [3].

6 Conclusion

Exploiting problem structure has proven to be a key element in many optimization problems. We argue about the structure of a program's CFG and its properties specifically with regard to WCET analysis.

WCET analysis tools have to continuously restrict problem size, in order to meet space as well as time constraints and maintain feasibility: (1) During abstract interpretation, the overwhelming combination of program-, pipeline-, and cache states may require merging. (2) Solving the longest path problem, while accounting in detail for all processor states, again is only feasible up to a certain magnitude of states (variables). The downside of these techniques is, that the precision of the attainable WCET bound is reduced. While other approaches for tightening an IPET-based WCET bound are designed to refine the worst-case path in a *post-processing* step, graph pruning can be used in order to decrease problem size and increase precision during WCET analysis. Compared to existing program slicing techniques, our approach extends to the lower-level timing analysis too. With the algorithms given above, we further demonstrated, that WCET profiles are a suitable first guide to this search.

Acknowledgments

This work is partially funded under the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST) and was supported by the Austrian Science Fund (FWF) under contract P21842.

References

[1] Ho Jung Bang, Tai Hyo Kim, and Sung Deok Cha. An iterative refinement framework for tighter worst-case execution time calculation. In *Proceedings of the Symposium on Object and Component-Oriented Real-Time Distributed Computing*, ISORC '07, pages 365–372. IEEE, 2007.

[2] Florian Brandner, Stefan Hepp, and Alexander Jordan. Static profiling of the worst-case in real-time programs. In *Proceedings of the Conference on Real-Time and Network Systems*, RTNS '12, pages 101–110. ACM, 2012.

[3] Florian Brandner, Stefan Hepp, and Alexander Jordan. Criticality: Static profiling for real-time programs. *Real-Time Systems*, pages 1–34, 2013.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 3rd edition, 2009.

[5] Agostino Cortesi and Matteo Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems &Structures*, 37(1):24–42, 2011.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages*, POPL '77, pages 238–252. ACM, 1977.

[7] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis. In Christine Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Pisa, Italy, July 3, 2007*, volume 07002 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[8] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. Using a worst-case execution-time tool for real-time verification of the DEBIE software. In *Proceedings of the Data Systems in Aerospace Conference (ESA SP-457)*, DASIA '00, pages 307–312. ESA, 2000.

[9] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. WCET squeezing. In *Proceedings of the 21st International conference on Real-Time Networks and Systems - RTNS '13*, page 161. ACM Press, 2013.

[10] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the Design Automation Conference*, DAC '95, pages 456–461. ACM, 1995.

[11] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '09, pages 136–146. IEEE, 2009.

[12] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. Papabench: A free real-time benchmark. In *Proc. of the Workshop on Worst-Case Execution Time Analysis*, pages 63–68. OCG, 2006.

[13] Peter P. Puschner and Anton V. Schedl. Computing maximum task execution times - a graph-based approach. *Real-Time Systems*, 13(1):67–91, July 1997.

[14] Christer Sandberg, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Faster WCET flow analysis by program slicing. In *Proceedings of the conference on Language, Compilers, and Tool Support for Embedded Systems*, LCTES '06, pages 103–112. ACM, 2006.

[15] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *Proc. of the Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 132–140. ACM, 2001.

[16] Ingmar Jendrik Stein. *ILP-based Path Analysis on Abstract Pipeline State Graphs.* PhD thesis, Universität des Saarlandes, 2010.

[17] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, 2000.

[18] Mark Weiser. Program slicing. In *Proceedings of the Conference on Software Engineering*, ICSE '81, pages 439–449. IEEE, 1981.