DERIVING WCET BOUNDS BY ABSTRACT EXECUTION

Andreas Ermedahl[1,2], Jan Gustafsson[1,3], Björn Lisper[1,4]

Abstract

Standard static WCET analysis methods today are based on the IPET technique, where WCET estimation is formulated as an integer linear programming (ILP) problem subject to linear program flow constraints with an objective function derived from the hardware timing model. The estimate is then calculated by an ILP solver. The hardware cost model, as well as the program flow constraints, are often derived using a static program analysis framework such as abstract interpretation.

An alternative idea to estimate the WCET is to add time as an explicit variable, incremented for each basic block in the code. The possible values of this variable can then be bound by a value analysis. We have implemented this idea by integrating the time estimation in our Abstract Execution *method for calculating program flow constraints. This method is in principle a very detailed value analysis. As it computes intervals bounding variable values, it bounds both the BCET and the WCET. In addition, it derives the explicit execution paths through the program which correspond to the calculated BCET and WCET bounds.*

We have compared the precision and the analysis time with the traditional IPET technique for a number of benchmark programs, and show that the new method typically is capable of calculating as tight or even tighter WCET estimates in shorter time. Our current implementation can handle simple timing models with constant execution times for basic blocks and edges in the CFG, but it is straightforward to extend the method to more detailed hardware timing models.

1. Introduction

The *worst-case execution time* (WCET) is a key parameter for verifying real-time properties. *Static WCET analysis* finds an upper bound to the WCET of a program from models of the hardware and software involved. If the models are correct, the analysis will derive a *safe* timing estimate i.e., greater than or equal to the WCET. Static WCET analysis is traditionally divided into three phases [14]:

1. A *flow analysis* phase, where program flow constraints such as upper bounds on the number of loop iterations are obtained.
2. A *low-level analysis* phase, where upper bounds on the execution time of basic blocks are obtained.
3. A *bound calculation* phase, where the flow- and timing bounds for instructions, derived in the first two phases, are combined to derive a WCET bound for the whole program.

The IPET approach [9, 11], which today is the preferred technique for WCET analysis, follows this scheme. In IPET, the flow analysis results are expressed as linear constraints on execution counters

[1] School of Innovation, Design and Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden. {andreas.ermedahl,jan.gustafsson,bjorn.lisper}@mdh.se

[2] Andreas Ermedahl is funded by Vinnova and Artemis through the CHESS project. [3] Jan Gustafsson is funded by Mälardalen University. [4] Björn Lisper is partially funded by Vinnova through the TIMMO-2-USE project and partially by the European Commission through the ArtistDesign Network of Excellence.

for different program parts, and the WCET estimate is calculated by solving an integer linear programming (ILP) problem subject to these constraints. The scheme above is modular, and IPET allows for taking quite general program flow constraints into account in the WCET bound calculation.

However, the approach has some potential problems. We have shown [7] that detailed flow constraints are important for tight WCET estimates; only loop bounds are not always sufficient[1]. Further, during flow analysis it is hard to identify flow facts that will not affect the resulting estimate and therefore can be excluded from the IPET solving. Thus, to guarantee tight WCET bounds, maximal production of flow facts are often required. Since ILP is NP-hard, this leads to a potential complexity problem. For instance, abstract execution [7], when run with full precision, generates more than 2.5 million flow constraints for the program `nsichneu` (4253 LoC) from the Mälardalen benchmark suite [5].

Thus, it is interesting to investigate other, more integrated approaches to WCET calculation. One interesting approach, first suggested by Holsti [8], is to add time as an explicit variable in the program. For each basic block in the code, the time is incremented with the maximal execution time of that block. The execution time of the program can now be bounded by a conventional value analysis, e.g., based on Abstract Interpretation (AI) [1]. This approach has some potential advantages:

- AI can be used both for program flow analysis [4, 7], and low-level analysis [13]. This makes it easier to integrate the WCET calculation, and eliminate the need to generate explicit flow constraints.
- There are many tradeoffs between analysis time and precision for static value analyses with respect to choice of abstract domain, context sensitivity, etc. This allows for adjusting the precision of the WCET calculation to obtain reasonable calculation times. E.g., value analyses can be designed to run in polynomial time, which eliminates the risk of exponential running time associated with ILP.
- It is easy to add a BCET analysis by using an abstract domain that bounds values from below.
- If a relational abstract domain is used, such as Halbwachs' polyhedral domain [2], then parametric WCET estimates can be calculated.

The basic approach assigns constant execution times to each basic block. If basic blocks have varying execution times, the increment of the time variable can be modeled by a safe execution time interval in the value analysis. Pipeline overlap can be modeled by decrements of the time variable. Context-sensitive basic block execution times, arising from a precise low-level analysis, can also be taken into account by making the value analysis sensitive to the same contexts as the low-level analysis.

This paper describes how we have extended our WCET analysis tool [7] with such an integrated method. SWEET performs a program flow analysis by *Abstract Execution* (AE), which in principle is a very context-sensitive AI. The underlying AI framework is used to bind the possible variable values at different program points in a context-sensitive manner. The following sections describe how we have extended AE to also compute bounds for the execution time: thus, SWEET can perform a direct WCET estimation without generating program flow constraints, nor using an IPET calculation. In Section 6, we compare the analysis speed and precision of WCET estimates calculated with this extended AE to the traditional three-phase WCET analysis of SWEET. Our results show that the extended AE method often runs considerably faster, with preserved or improved precision.

[1] Some examples in Table 2 in Section 6 also show this clearly.

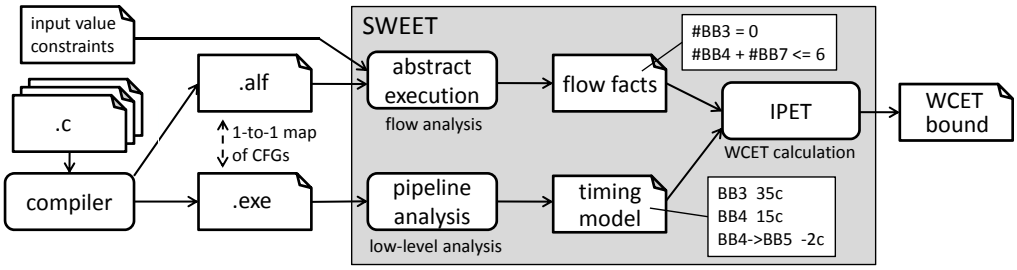


Figure 1. Traditional WCET analysis by SWEET

2. Related work

There are many types of advanced flow analysis methods presented in the WCET literature, see e.g., [14]. However, to our knowledge, there are few methods presented, where the flow information extraction is integrated with a precise WCET bounds calculation. We have already mentioned Holsti's work [8] where time is introduced as an explicit variable. His idea has similarities to our outlined method. However, while Holsti proposes using a Presburger solver to find relations that can bound the values of the time variable, we instead use AE, a type of detailed value analysis, to derive both a WCET bound. We also include extensive evaluations of our method, while Holsti does not.

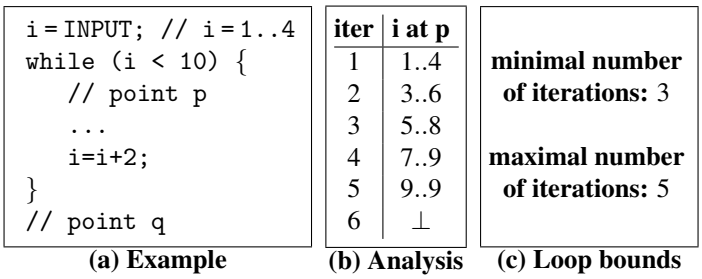
Our method has some similarities with the work of Lundqvist [10], which derives a WCET bound by a combined hardware- and flow analysis method. However, AE uses a more detailed value domain and is based on an AI framework. Also, our low-level timing analysis is made in a modular step, clearly separated from the WCET bound calculation.

3. SWEET and Abstract Execution

SWEET is a WCET analysis tool developed at Mälardalen University. As shown in Figure 1, it is modularly built, using the standard WCET analysis tool architecture with a flow analysis, a low-level analysis, and a final WCET estimate calculation. The low-level analysis of SWEET supports the NECV850E and ARM9 processors.

The main flow analysis method of SWEET is called *Abstract Execution* (AE) [7]. AE can be seen as a fully context-sensitive abstract interpretation (AI), where each loop iteration or function call is analysed separately. Rather than using traditional fixed-point analysis, AE executes the program in the abstract domain, with abstract values of variables and abstract versions of language operators. The abstract value held by a variable, at some point, corresponds to a set containing the actual concrete values that the variable can hold at that point. An *abstract state* is a collection of all abstract variables.

As an illustration of AE, using intervals as abstract values, please consider Figure 2. When entering the loop, the variable `i` can hold any integer value from 1 to 4. Each execution of an abstract state by the `while` condition might give rise to at least one, and at most two resulting states (the *true* and *false* branch). During the first three executions of the loop condition, there is no value of `i` which terminates the loop. However, the fourth time the condition is executed, `i` will have a value of [7..10], giving that the analysis produces two resulting states. Thus, `i` will have a value of [7..9] at point `p`, and [10..10] at point `q`. Similarly, during the following execution of the loop condition both branches can be taken. At the sixth execution of the loop condition, the set of values for the *true* branch of the loop condition is empty, i.e., only the *false* branch is possible, and AE of the loop terminates.



(a) Example **(b) Analysis** **(c) Loop bounds**

Figure 2. Example of Abstract Execution

The normal output of AE is a set of *flow facts* [3], i.e., constraints on the program flow, which is given as input to the subsequent calculation phase. To derive these constraints, AE extends abstract states with *recorders*, used to collect flow information. Program parts to be analyzed are extended with *collectors*, which are used to successively accumulate recorded information from the states. For example, in Figure 2 each state may be given a *loop bound recorder* for recording the number of executions it makes in the loop. Similarly, a *loop bound collector* can be used to accumulate the loop body executions (3, 4 and 5 respectively) recorded by the states. The accumulated recordings are used to generate the loop bound constraints in Figure 2(c). Flow constraint generation supported by AE include lower and upper (nested) loop bounds, infeasible nodes and edges, upper node and edge execution bounds, infeasible pairs of nodes, and longer infeasible paths [7].

As illustrated in Figure 2, when using abstract values, conditionals cannot always be decided. In these cases, AE must then execute both branches separately in two different abstract states. This means that AE may have to handle many abstract states, representing different possible execution paths, concurrently. In order to curb the growing number of paths, *merging* of abstract states for different paths can take place at certain program points (*merge points*). If the states are merged using the least upper bound operator "$\sqcup$" on the abstract domain of states, then the result is one abstract state safely representing all possible concrete states. Thus, a single-path abstract execution, representing the execution of the different paths, can continue from the merge point. Merge points can be selected at will, but typical placements are join points where different program flows meet, like after if-statements, and exits from functions and loops.

The underlying algorithm for processing abstract states is outlined in Figure 3. It is a quite straightforward worklist algorithm, which iterates over a set of abstract states, generating new abstract states from old ones. Abstract states at merge points are moved to a special merge list, and final states are removed. When the worklist is empty, all states in the merge list which are at the same merge point are merged, and the resulting states are inserted in the worklist. The algorithm terminates when both the merge list and the worklist are empty.

The AE is *input-sensitive*. For example, in Figure 2, the restricting interval for the initial value of `i` is directly influencing the resulting loop iteration bounds. SWEET includes an expressive annotation language for allowing the user to give input value constraints. The analysis methods of SWEET are general and not a priori tied to any language or instruction set. To take advantage of this generality, SWEET's flow analysis analyses the code format ALF [6]. ALF is an intermediate format designed to be able to faithfully represent code on different levels, from source to binary level. Thus, SWEET can analyse code both on binary and source level provided that a translator to ALF is present.

```
work_list <- {init_state};
merge_list <- empty;
final_list <- empty;
REPEAT
  WHILE work_list /= empty DO {
    s <- select_from(work_list);
    work_list <- work_list \ {s};
    new_states <- ae(s);
    FOREACH s' in new_states DO
      CASE merge_point(s'): merge_list <- merge_list U {s'}
           final_state(s'): final_states <- final_states U {s'}
           otherwise: work_list <- work_list U {s'};
  }
  WHILE merge_list /= empty DO {
    s <- select_from(merge_list);
    merge_list <- merge_list \ {s};
    FOREACH s' in merge_list DO
      IF same_merge_point(s, s') THEN
        s <- merge(s, s');
        merge_list <- merge_list \ {s'};
    work_list <- work_list U {s};
  }
UNTIL work_list = empty
```

Figure 3. Underlying AE algorithm

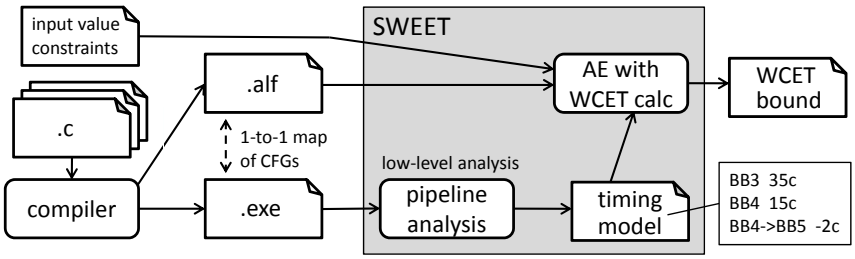


Figure 4. AE-based WCET analysis by SWEET

4. Abstract Execution with WCET calculation

As explained above, SWEET's flow analyses, and particular the AE, may generate large amount of flow facts. Both the flow fact generation in itself and the ILP solving may take a lot of time. To overcome this problem, we have developed an extension to SWEET where WCET estimates are derived directly in the AE. Figure 4 illustrates the extended AE analysis and its inherent phases.

We extend the abstract states of the AE to keep track of the smallest (`min_t`) and the largest (`max_t`) cost for all the executions that have resulted in the particular state. Each state also holds the execution paths, `min_path` and `max_path`, that resulted in these timing values. We also need to update the `ae` and `merge` functions, see Figure 3, according to the added state content.

Figure 5 shows the new versions of the `ae` and `merge` operations. Basically, whenever a basic block is abstractly executed, see Figure 5(a), its corresponding cost is looked up in the timing model (`tm`), and the best and worst case timing for the state are updated accordingly. The executed basic block is also added to the best- and worst-case paths held in the state. It is straightforward to extend the `ae`

```
ae(s) :-
  return_states <- empty;
  s.min_t <- s.min_t + tm.min_cost(s.node);
  s.max_t <- s.max_t + tm.max_cost(s.node);
  s.min_path.push_back(s.node);
  s.max_path.push_back(s.node);
  FOREACH successor node n of s.node in cfg DO
    s' <- s.copy();
    s' <- abstractly_execute_to_succ(s',s.node.stmt,n);
    IF feasible(s) THEN
      s'.node <- n;
      return_states <- return_states U {s'};
  RETURN return_states;
```

(a) Execution of abstract state

```
merge(s1,s2) :-
  s <- s1.copy();
  FOREACH variable v in s DO
    s.v <- s1.v U s2.v;
  IF s1.min_t > s2.min_t THEN
    s.min_t <- s2.min_t;
    s.min_path <- s2.min_path;
  IF s1.max_t < s2.max_t THEN
    s.max_t <- s2.max_t;
    s.max_path <- s2.max_path;
  RETURN s;
```

(b) Merge of abstract states

Figure 5. Extended AE operations

to take into account more complex timing models, e.g., assigning timing to individual statements or transitions between basic blocks. When two abstract states are merged, see Figure 5(b), their included variable values are, as before, merged using the least upper bound operator. Moreover, the smallest and largest timings of the two states are extracted, as well as their corresponding execution paths, and are set to be the smallest and largest timings and paths of the resulting state.

5. An illustrative example

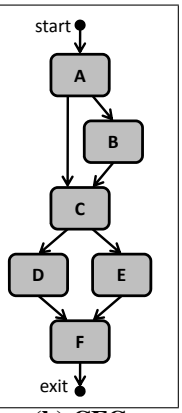
For an illustration on how the extended AE works consider the example program in Figure 6(a). For illustration purposes C code is given, instead of ALF code. We have identified six basic blocks in the code, labeled `A` to `F`. Figure 6(b) holds the resulting control-flow graph (CFG). To each basic block a cost has been associated, as illustrated in Figure 6(c). This cost corresponds to the clock cycles, e.g., derived by some low-level analysis, it takes to run its inherent code on a given target platform. We here assume a rather simple target, e.g., an 8- or 16-bit processor, giving that most nodes have a constant cost. However, some nodes, such as `B` and `D`, have lower and upper costs associated to them, e.g., due to what parameters the multiply instruction is run with. On more advanced targets, non-constant basic block costs may come from hardware features such as pipelines, caches or branch predictors [14]. For such targets the timing model may also need to include more long-reaching timing effects. The `x` and `res` variables are global, and have also been given initial value constraints.

Figure 7 illustrates how the extended AE analysis would process the code in Figure 6(a) when using merge of states after if-statements. The first abstract state `S1` holds the initial values of `x` and `res`. At the execution of the last statement in `A` the state will be split into two states, `S2` and `S3`, corresponding to the *true* and *false* branches. The `res = res + 1;` statement will update the `res` variable to hold values in between 2 and 11, while the following if-condition will constrain the values of the `x` variable according to the *true* or *false* branches. For example, the state proceeding to the `B` node, `S2`, is constrained to only hold values of `x` which are smaller than 10. Each state has also been updated with the cost for executing `A`, and `A` has been appended to the min and max paths.

`S2` is then updated according to the code in `B` to become `S2'`. Next, `S2'` and the `S3` are merged into `S4`. The new values of variables in `S4` are the least upper bound of the corresponding variable values in `S2'` and `S3`. The smallest min time is found in `S3`, and correspondingly the min path of `S4` is set to the min path of `S3`. Similarly, the largest time and the corresponding max path is found in `S2'`.

```
// x = 1..100 res = 1..10
int foo(void) {            // A
   res = res + 1;
   if(x < 10)
      res = res * 2 + 1; // B
   res = res + x;          // C
   if(x > 32)
      res = res * 3;      // D
   else
      res = res - 1;      // E
   return res;             // F
}
```

(a) Example code

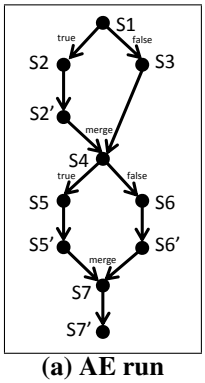


(b) CFG

node	cycles
A	22
B	11..18
C	6
D	8..15
E	3
F	14

(c) Timing model

Figure 6. Example code with timing model

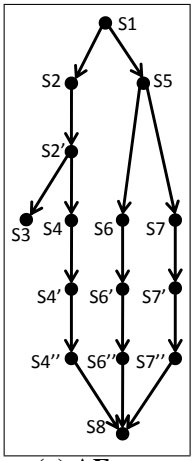


(a) AE run

state name	program point	variable values x	variable values res	time min	time max	paths min	paths max
S1	start→A	1..100	1..10	0	0	-	-
S2	A→B	1..9	2..11	22	22	A	A
S2'	B→C	1..9	5..23	33	40	A,B	A,B
S3	A→C	10..100	2..11	22	22	A	A
S4	_→C	1..100	2..23	22	40	A	A,B
S5	C→D	33..100	3..123	28	46	A,C	A,B,C
S5'	D→F	33..100	9..369	36	61	A,C,D	A,B,C,D
S6	C→E	1..32	3..123	28	46	A,C	A,B,C
S6'	E→F	1..32	2..122	31	49	A,C,E	A,B,C,E
S7	_→F	1..100	2..369	31	61	A,C,E	A,B,C,D
S7'	F→exit	1..100	2..369	45	75	A,C,E,F	A,B,C,D,F

(b) AE states

Figure 7. AE run with merge at joins



(a) AE run

state name	program point	variable values x	variable values res	time min	time max	paths min	paths max
S1	start→A	1..100	1..10	0	0	-	-
S2	A→B	1..9	2..11	22	22	A	A
S2'	B→C	1..9	5..23	33	40	A,B	A,B
S3	C→D	⊥	6..31	39	46	A,B,C	A,B,C
S4	C→E	1..9	6..31	39	46	A,B,C	A,B,C
S4'	E→F	1..9	6..31	41	49	A,B,C,E	A,B,C,E
S4''	F→exit	1..9	6..31	55	63	A,B,C,E,F	A,B,C,E,F
S5	A→C	10..100	2..11	22	22	A	A
S6	C→D	33..100	12..111	28	28	A,C	A,C
S6'	D→F	33..100	36..333	36	43	A,C,D	A,C,D
S6''	F→exit	33..100	36..333	50	57	A,C,D,F	A,C,D,F
S7	C→E	10..32	12..111	28	28	A,C	A,C
S7´	E→F	10..32	11..110	31	31	A,C,E	A,C,E
S7''	F→exit	10..32	11..110	45	45	A,C,E,F	A,C,E,F
S8	_→exit	1..100	6..333	45	63	A,C,E,F	A,B,C,E,F

(b) AE states

Figure 8. AE run with merge at program end

Similarly to how S1 was split into two states, S4 is split into S5 and S6, corresponding to the *true* and *false* branches, respectively. S5 is then updated according to the code in the D node, and S6 is updated according to the code in the E node. The resulting states are then merged at the E node, and the resulting state S7 is updated with the code of the E node. This gives the resulting BCET and WCET bounds of 45 and 75 clock cycles respectively. The explicit paths corresponding to these bounds are also returned. Please note that the resulting WCET bound is a safe but somewhat pessimistic value. This is because the A,B,C,D,F path actually is infeasible, since x can't be less than 10 and larger than

Program	Description	#LC	#F	#L	#C	#CSF	#CSL
adpcm	Adaptive pulse code modulation algorithm.	879	17	18	37	38	27
bs	Binary search in an array of 15 integer elements.	114	2	1	3	2	1
bsort100	Bubblesort program, sorting 100 integers.	128	3	3	6	3	3
cnt	Counts non-negative numbers in a matrix.	267	6	4	5	6	4
cover	Program with many paths (using loops and switches).	640	4	3	6	4	3
crc	Cyclic redundancy check computation on 40 data bytes.	128	3	3	9	5	6
edn	Finite Impulse Response (FIR) filter calculations.	285	9	12	12	9	12
expint	Series expansion computing an exponential integral.	157	3	3	6	3	3
fdct	Fast Discrete Cosine Transform.	239	2	2	2	2	2
fibcall	Iterative Fibonacci, used to calculate fib(30).	72	2	1	1	2	1
fir	Finite impulse response filter (signal processing).	276	2	2	4	2	2
inssort	Insertion sort on an array of size 30.	92	1	2	2	1	2
jcomplex	Nested loop program.	64	2	2	4	2	2
jfdctint	Discrete-cosine transformation on 8x8 pixel block.	375	2	3	3	2	3
loop3	Loops with context-sensitive execution behaviour	76	1	150	150	1	150
matmult	Matrix multiplication of two 20x20 matrices.	163	6	5	5	8	7
nsichneu	Simulates an extended Petri net (many paths).	4253	1	1	253	1	1
statemate	Automatically generated code.	1276	8	1	99	8	1
ud	Linear equations by LU decomposition.	161	2	11	13	2	11

Table 1. Benchmark programs

32 at the same time. The reason why this is not detected is due to the merge of states made at node C, which remove the previous connections between the possible variable values and the nodes executed.

Figure 8 illustrates how the AE would process the code when using merge only at the end of the program. Thus, merging of states is no longer made at the C or F nodes. As a result, more concurrent states will be generated, but with more precise value constraints, time and path information. We can first observe that the A,B,C,D,F path no longer is considered feasible. This is because that S3 state, which originates from the split of S2' state at the end of node C, can't assign any feasible value to the x variable, and therefore the AE can't proceed further with S3. This gives that we now get a tighter WCET bound, 63 cycles instead of 75, and a different resulting WCET path: A,B,C,E,F. We conclude that merging allows us to trade BCET- and WCET bound precision with analysis time. In the following section this fact will be shown for a number of benchmark programs.

6. Evaluation

The outlined method has been evaluated using 16 programs from the Mälardalen WCET Benchmark Suite [5]. Table 1 gives some basic data about the programs: lines of C code (**#LC**), number of functions (**#F**), loops (**#L**), and conditional statements (**#C**). The **#CSF** column gives the number of functions found in a fully context-sensitive call graph, and **#CSL** gives the corresponding amount of loops found in this graph. The reason for giving the latter information is that SWEET can provide flow facts in a context-sensitive manner. This is useful when we want very precise WCET bounds, and we have code parts with context-sensitive flow behaviour. For example, a loop may get different upper loop bounds in different calling contexts.

Table 2 and 3 compares traditional vs. AE-based WCET calculation[2]. The analysis was performed on a Macintosh MacBook Pro with Intel Core 2 Duo 2.53 GHz and 4 GB memory, running OS X 10.5.8. We use a timing model derived by the SWEET ARM9 low-level analysis.

Table 2 gives the analysis results when analysing the programs in the traditional way. Column **#input**

[2]We do not include evaluations of BCET calculations since SWEET's low-level analysis currently does not derive safe lower timing bounds for code parts.

Program	#input comb.	Merge strategy	FF generation	Time AE	#flow facts	Time IPET	#linear constr.	WCET est. cycles	+%
adpcm	1	-	all	13.8	16581	43.6	19461	87221	0
		-	ulb	7.9	27	1.6	2880	160034	83
bs	2^{32}	full	all	0.10	261	0.04	351	124	0
		full	ulb	0.07	1	0.01	90	124	0
		none	all	0.10	261	0.04	351	124	0
		none	ulb	0.07	1	0.01	90	124	0
bsort100	2^{3200}	full	all	23.8	321	0.07	467	99802	0
		full	ulb	16.2	3	0.01	146	196430	97
		none	all	-	-	-	-	-	-
		none	ulb	-	-	-	-	-	-
cnt	1	-	all	0.35	434	0.14	644	11903	0
		-	ulb	0.27	4	0.02	210	12603	6
cover	1	-	all	17.4	658316	-	-	-	-
		-	ulb	1.0	3	2.2	3184	26123	24
crc	1	-	all	3.8	2706	1.3	3184	83275	0
		-	ulb	2.9	6	0.07	481	165171	98
edn	44	full	all	4.1	1286	0.7	2026	9241	0
		full	ulb	3.1	12	0.1	740	9241	0
		none	all	13.6	1286	0.7	2026	9241	0
		none	ulb	9.8	12	0.1	740	9241	0
expint	1	-	all	0.18	646	0.14	840	645	0
		-	ulb	0.12	3	0.02	195	11337	1658
fdct	1	-	all	0.14	122	0.05	428	37	0
		-	ulb	0.12	2	0.03	306	37	0
fibcall	1	-	all	0.04	90	0.01	160	76	0
		-	ulb	0.03	1	0.01	70	76	0
fir	1	-	all	0.28	393	0.01	561	877	0
		-	ulb	0.23	2	0.01	168	967	10
inssort	2^{320}	full	all	0.13	127	0.02	219	332	0
		full	ulb	0.10	2	0.01	92	404	22
		none	all	-	-	-	-	-	-
		none	ulb	-	-	-	-	-	-
jcomplex	900	full	all	0.32	397	0.08	505	4508	634
		full	ulb	0.14	2	0.01	108	11557	1782
		none	all	142.4	402	0.01	510	614	0
		none	ulb	65.2	2	0.01	108	9133	1387
jfdctint	1	-	all	0.14	166	0.05	561	321	0
		-	ulb	0.13	3	0.02	258	321	0
loop3	1	-	all	2.04	33090	85.0	4832080	2549	0
		-	ulb	0.51	120	0.84	2068	2549	0
matmult	1	-	all	10.5	688	0.34	968	12205	0
		-	ulb	7.1	7	0.03	280	12205	0
statemate	1	-	all	5.98	166001	768	168343	1346	0
		-	ulb	0.64	1	1.56	2342	5096	279
nsichneu	1	-	all	81.6	2598331	-	-	-	-
		-	ulb	2.22	1	15.5	5388	23830	11
ud	1	-	all	0.95	2112	1.15	2460	2160	0
		-	ulb	0.84	11	0.04	348	2905	34

Table 2. Results from traditional WCET calculation

comb. shows the number of input value combinations that the analysis is run with. If this value is 1, we do a single path analysis, if it is > 1, we have a (potentially) multi-path analysis. For example, we do a multi-path analysis for `inssort` with 10 input variables of size 32 bits which are all set to $\top$, i.e., unknown, yielding 2^{320} input value combinations. The possible input values of the program are specified in SWEET's input value annotation language. **Merge strategy** is only valid for multi-path analysis; "full" meaning that all possible merge points are used (see Section 3), "none" indicates that no merging is performed. **FF generation** shows the level of flow fact generation: "all" means that all types of flow facts are generated (see Section 3), "ulb" means only upper loop bounds (required to bind the WCET). **Time AE** gives SWEET's analysis time (in seconds). The column **#flow facts** shows the number of flow facts generated by SWEET. **Time IPET** shows the time spent in the IPET solver, and **#linear constr.** the number of solved constraints. For some programs the time is represented with

Program	#input comb.	Merge strategy	Time AE	Time -% vs. trad	WCET est. cycles	WCET est. +%
adpcm	1	-	8.2	86/14	87221	0
bs	2^{32}	full	0.07	50/13	124	0
		none	0.07	50/13	124	0
bsort100	2^{3200}	full	16.6	30/-2	99802	0
		none	-	-	-	-
cnt	1	-	0.27	45/7	11903	0
cover	1	-	1.0	-/69	21083	0
crc	1	-	3.0	41/-1	83275	0
edn	44	full	3.2	33/0	9241	0
		none	9.6	33/3	9241	0
expint	1	-	0.13	59/7	645	0
fdct	1	-	0.12	59/7	37	0
fibcall	1	-	0.03	40/57	76	0
fir	1	-	0.23	21/4	877	0
inssort	2^{320}	full	0.10	33/9	332	0
		none	-	-	-	-
jcomplex	900	full	0.10	75/33	4508	634
		none	67.3	53/-3	614	0
jfdctint	1	-	0.13	32/13	321	0
loop3	1	-	0.50	99/63	2549	0
matmult	1	-	7.20	34/-1	12205	0
nsichneu	1	-	2.22	-/87	21562	0
statemate	1	-	0.76	100/65	1346	0
ud	1	-	0.79	62/10	2160	0

Table 3. Results from AE-based WCET calculation

”-”, meaning that no result was obtained in reasonable time. **WCET est.** shows the WCET estimate in cycles, and **+%** the overestimation compared to the tightest result we have got during our analyses.

Table 2 shows that SWEET can bind all loops in the benchmark programs and calculate a WCET estimate. But it can be noticed that it is often necessary to generate a large amount of flow facts to obtain a tight WCET-estimation; only loop bounds are not always sufficient. We can also see that the these large amounts of flow facts can take a very long time, compared to using only loop bounds.

Table 3 gives the analysis results when performing the new AE-based WCET calculation. It has the three first columns in common with Table 2. **Time AE** shows the SWEET analysis time. The column **Time -% vs. trad.** shows the time savings in % compared to traditional WCET calculation. The notion ”x/y” means x = compared to the total time when generating all flow facts, and y = compared to the time for only loop bound generation. The last two columns are the same as in Table 2.

Table 3 shows that AE-based WCET calculation gives the same WCET estimates as the traditional WCET calculation in Table 2, but in most cases much faster. For two benchmarks (`cover` and `nsichneu`), the new method even gives a tighter estimation. In both tables, we also see that merging cuts the analysis time considerably and that it can be necessary for getting a result at all (`inssort` and `bsort100`). On the other hand, it may lead to overestimation (`jcomplex`).

7. Conclusions and future work

We conclude that our extension of AE is capable of deriving precise WCET estimates much faster than traditional IPET-based methods, by eliminating the need for flow fact generation and ILP solving. It can also find the execution path corresponding to the WCET estimate.

Future work includes to extend the method to handle more context-sensitive timing information, since one instruction may get different execution time depending on the instructions executed before it. This

should be important when analysing processors with history-dependant timing behaviour, e.g., due to caches or branch predictors [14]. Challenges includes how to extract such timing information, how to define a suitable timing model (especially if the processor exhibits timing anomalies [12]), and how to use the resulting timing model in the AE. Further, in our current implementation, only the whole program's WCET is extracted in the extended AE. However, using the recorder and collector machinery, as described in Section 3, it should be straight-forward to also derive WCET bounds for smaller code parts, such as individual functions or loops. Another upcoming challenge is to make SWEET and its inherent flow analyses easily available for other WCET researchers. We are currently, with researchers from Vienna University of Technology, developing a ALF translator, based on the LLVM framework (`www.llvm.org`). LLVM supports a variety of languages and processor back-ends.

References

[1] COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages* (Los Angeles, Jan. 1977), pp. 238–252.

[2] COUSOT, P., AND HALBWACHS, N. Automatic discovery of linear restraints among variables of a program. In *Proc. 5th ACM Symposium on Principles of Programming Languages* (1978), pp. 84–97.

[3] ERMEDAHL, A. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala University, Sweden, June 2003.

[4] ERMEDAHL, A., SANDBERG, C., GUSTAFSSON, J., BYGDE, S., AND LISPER, B. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Proc. 7th International Workshop on Worst-Case Execution Time Analysis (WCET'2007)* (Pisa, Italy, July 2007), C. Rochange, Ed.

[5] GUSTAFSSON, J., BETTS, A., ERMEDAHL, A., AND LISPER, B. The Mälardalen WCET benchmarks – past, present and future. In *Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010)* (Brussels, Belgium, July 2010), B. Lisper, Ed., OCG, pp. 137–147.

[6] GUSTAFSSON, J., ERMEDAHL, A., LISPER, B., SANDBERG, C., AND KÄLLBERG, L. ALF – a language for WCET flow analysis. In *Proc. 9th International Workshop on Worst-Case Execution Time Analysis (WCET'2009)* (Dublin, Ireland, June 2009), N. Holsti, Ed., OCG, pp. 1–11.

[7] GUSTAFSSON, J., ERMEDAHL, A., SANDBERG, C., AND LISPER, B. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proc. 27th IEEE Real-Time Systems Symposium (RTSS'06)* (Dec. 2006).

[8] HOLSTI, N. Computing time as a program variable: a way around infeasible paths. In *Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET'2008)* (Prague, Czech Republic, July 2008).

[9] LI, Y.-T. S., AND MALIK, S. Performance analysis of embedded software using implicit path enumeration. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'95)* (La Jolla, CA, June 1995).

[10] LUNDQVIST, T. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, June 2002.

[11] PUSCHNER, P. P., AND SCHEDL, A. V. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems 13*, 1 (July 1997), 67–91.

[12] REINEKE, J., WACHTER, B., THESING, S., WILHELM, R., POLIAN, I., EISINGER, J., AND BECKER, B. A definition and classification of timing anomalies. In *Proc. 6th International Workshop on Worst-Case Execution Time Analysis (WCET'2006)* (July 2006).

[13] THESING, S. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[14] 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., AND STENSTRÖM, P. The worst-case execution time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS) 7*, 3 (2008), 1–53.