View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Merging Techniques for Faster Derivation of WCET Flow Information using Abstract Execution

Jan Gustafsson and Andreas Ermedahl
School of Innovation, Design and Engineering, Mälardalen University
Box 883, S-721 23 Västerås, Sweden
{jan.gustafsson, andreas.ermedahl}@mdh.se

Abstract
Static Worst-Case Execution Time (WCET) analysis derives upper bounds for the execution times of programs. Such bounds are crucial when designing and verifying real-time systems. A key component in static WCET analysis is to derive flow information, such as loop bounds and infeasible paths.

We have previously introduced abstract execution *(AE), a method capable of deriving very precise flow information. This paper present different merging techniques that can be used by AE for trading analysis time for flow information precision. It also presents a new technique,* ordered merging, *which may radically shorten AE analysis times, especially when analyzing large programs with many possible input variable values.*

1. Introduction

The *worst-case execution time* (WCET) is a key parameter for verifying real-time properties. A *static WCET analysis* finds an upper bound to the WCET of a program from mathematical models of the hardware and software involved. If the models are correct, the analysis will derive a timing estimate that is *safe*, i.e., greater than or equal to the WCET.

To statically derive a timing bound for a program, information on both the *hardware timing characteristics*, such as bounds on the time different instructions may take to execute, as well as the program's *possible execution flows*, such as bounds on the number of times each instruction can be executed, needs to be derived. The latter, so called *flow information*, includes information about the maximum number of times loops are iterated, which paths through the program that are feasible, dependencies between code parts, etc.

The goal of a *flow analysis* is to calculate such flow information as automatically as possible. Flow analysis research has mostly focused on *loop bound* analysis, since upper bounds on the number of loop iterations must be known in order to derive a WCET estimate. Flow analysis can also identify *infeasible paths*, i.e., paths which are executable according to the control-flow graph structure, but not feasible when considering the semantics of the program and possible input data values [9]. In contrast to loop bounds, infeasible path information is not required to find a WCET estimate, but may tighten the resulting WCET estimate. In general, a flow analysis which can take constraints on input data values into consideration, a so called *input sensitive analysis*, should be able to derive more precise

This research has been supported by the KK-foundation through grant 2005/0271. Travel support has been provided by the ARTIST2 European Network of Excellence.

flow information than an analysis which cannot [2].

One promising flow analysis method is *Abstract Execution* (AE), which is able to find precise flow information for many different types of programs [2, 9]. AE works by abstractly executing the paths which the program may execute for all its different input values. To avoid a combinatorial explosion of the number of paths to analyse, different types of *merging techniques* are used by AE. This article describes these techniques in more detail. The contributions of this article are:

- We present merging techniques used by AE for trading analysis time for flow information precision.
- We present a new technique, *ordered merging*, which may radically reduce AE analysis times for large programs with many possible input variable values.
- We evaluate the effect of the different techniques w.r.t. analysis time and flow information precision on some benchmark programs.

Even though our merging techniques are presented in the context of AE, we believe that they should be applicable for many other WCET analysis techniques where many program paths are explicitly explored.

The rest of the paper is organized as follows: Section 2 presents our WCET tool and AE. Section 3 presents the need for, and the drawbacks of, merging. Section 4 presents related work and other methods which should benefit of our techniques. Section 5 describes our merge point placement techniques and Section 6 presents our new technique for ordering of merge points. Section 7 presents analysis results. In Section 8 we draw some conclusions and discuss future work.

2. SWEET and Abstract Execution

SWEET (SWEdish Execution time Tool) is a prototype WCET analysis tool developed at Mälardalen University [14]. It consists of three main phases; a *flow analysis* where bounds on the number of times different instructions or larger code parts can be executed is derived [9], a *low-level analysis*, where timing cost bounds for different instructions or larger code parts are derived [5], and a *calculation* where the most time-consuming path is found using the information derived in the first two phases [6].

Abstract Execution. AE is a form of symbolic execution based on an AI [4] framework. Rather than using traditional fixed-point iteration, AE executes the program in the abstract domain, with abstract values for the program variables and abstract versions of the operators in the language. For instance, the abstract domain can be the domain of intervals: each numeric variable will then hold an interval rather than a number, and each assignment will calculate a new interval from the current intervals held by the variables. As usual in AI, the abstract value held by a variable, at some point, represents a set containing the actual concrete values that the variable can hold at that point. An *abstract state* is a collection of abstract values for all variables at a point. AE is *input data sensitive*, allowing the user to explore how different input data value constraints affect the program flow [2, 7].

Illustrative example. As an illustration of AE, using intervals as abstract values, please consider Figure 1. 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 raise 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 `r`. Similarly, during the following execution of the

```
i=INPUT; // i=[1..4]
while (i < 10) {
   ...   // p
   i = i + 2;
   ...   // q
}
// r
```

(a) Example

iter	i at p
1	[1..4]
2	[3..6]
3	[5..8]
4	[7..9]
5	[9..9]
6	impossible

(b) Analysis

min. #iter: 3

max. #iter: 5

(c) Result

Figure 1. Example of abstract execution

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.

Deriving flow constraints. The output of AE is a set of *flow facts* [6], 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 1 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 1(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 [9].

3. Merging

When using abstract values, conditionals cannot always be decided, as illustrated by the above example. 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. The number of possible abstract states may grow exponentially with the length of these paths.

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 after if-statements, and at entries/exits from functions and loops.

Problems with merging. Merging comes with a price, however, since it may yield abstract values that represent concrete values in a less precise way: for instance, the merging of $[6..6]$ and $[10..11]$ yields $[6..11]$, which also contains the concrete values 7, 8, 9 not present in the original intervals. The added values might, if we are unlucky, force AE to execute paths which are not feasible. Figure 2(a) shows an example with an upper loop bound = 4. AE with no merging would find that bound, whereas merging at point `p` would lead to an overestimated loop bound (8). Merging at point `q` (at the loop header) instead would lead to a lower overestimation (7) of the loop bound, i.e., also placement of

```
int i, x; // i=[-5..5]
if (i > 0) x=2;
else x=4;
// p
while (x < 10) { // q
   if (isOdd(x)) x++;
   else x=x+2;
}
```

(a) Loop bound overestimation

```
int i;    // i=[-5..5]
int x;
if (i > 0) x=2;   // A
else x=1;         // B
// r
if (i > 1) x=2*x; // C
else x=3*x;       // D
// s
```

(b) Missed infeasible paths

Figure 2. Overestimations due to merging

merge points are important.

Moreover, after a merge we cannot relate the path we executed before the merge with the path we executed after the merge, since several paths might be merged at the merge point. This means that some types of infeasible path calculation cannot be made after merging. Figure 2(b) shows an example where merging at `r` (join after if) leads to that the infeasible paths `A-C` and `B-D` both are missed. This is because the merged state at `r` has a path history where *both* nodes `A` and `B` are included. If we instead postpone the merging to `s` both infeasible paths can be found.

We conclude that merging may lead to a faster AE, but that it also may result in less precise flow information. In Section 5 we describe placement of merge points, as supported by SWEET, allowing us to trade analysis time and flow information precision. In Section 6 we present a new method, based on ordering of merge points, for minimizing the number of concurrent abstract states, and thereby achieving a much faster analysis.

4. Related Work

As mentioned in Section 2, AE can be classified as a combination of AI and symbolic execution [8, 9]. The WCET analysis method by Lundqvist et al. [13], works in a similar fashion, and can potentially also get many parallel states. Compared to Lundqvist's work, AE uses a more detailed value domain, is based on an AI framework, and derives only flow constraints.

In general, our merging techniques should be applicable for any other WCET analysis technique where the paths through the program or parts of the program are explicitly explored. For example, most path-based calculation methods use some type of merging [10, 13, 16, 17]. Similarly, path-enumeration based flow analysis approaches, such as [1, 11], may require merging when the amount of paths grow large.

On an even more general level, both *intra-procedural* and *inter-procedural* program analyses make use of different type of merging and merge points [15].

5. Placement of Merge Points

A crucial question is: where to place merge points? For example, to place them at all join points in the program could mean that we unnecessarily overestimate values, which might lead to less precise flow information, and a non-tight resulting WCET estimate.

```
int complex(int a, int b) {            // BB0
   while (a < 30) {                    // BB1
      while (b < a) {                  // BB2
         if (b > 5)                    // BB3
            b = b * 3;                 // BB4
         else
            b = b + 2;                 // BB5
         if (b >= 10 && b <= 12)       // BB6 & BB7
            a = a + 10;                // BB8
         else
            a = a + 1;                 // BB9
      }                                // BB10
      a = a + 2;                       // BB11
      b = b - 10;
   }
   return 1;                           // BB12
}

int main(void) {                       // BB13
   /* a = [0..18] b = [0..18] */
   int a = 1, b = 1, answer = 0;
   answer = complex(a, b);             // BB14
   return answer;
}
```

Figure 3. Example program

To handle this problem, our method allows the user to control the placement of merge points, in order to explore different tradeoffs between analysis speed and precision. Figure 3 shows an example program (`jcomplex` from the Mälardalen benchmarks [14]), to illustrate the possible placement of merge points. Figure 4 shows the corresponding program CFG where the function call to `complex` has been inlined in `main`, and nodes have been partitioned w.r.t. the functions and loops they belong to. This type of graph (the *scope-graph* [6]) is used by the analysis in SWEET.

The user can currently specify AE to use *no merging*, or one or more of the following merge points: at *function entries* (corresponding to nodes `BB0` and `BB13` in Figure 4), after *function exits* (`BB14`), after *loop body termination*, i.e., at the loop header (`BB1`, `BB2`), after *loop exits* (`BB11`, `BB12`), and at *joins after if-statements* (`BB6`, `BB9`, `BB10`). Note that a node can be of more than one merge point type.

6. Ordered and Unordered Merging

AE and its flow fact generating techniques has been presented in detail in [9]. The original underlying algorithm for processing abstract states is outlined in Figure 5. 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 algorithm allows any combination of merge point placements (as outlined in Section 5) to be used.

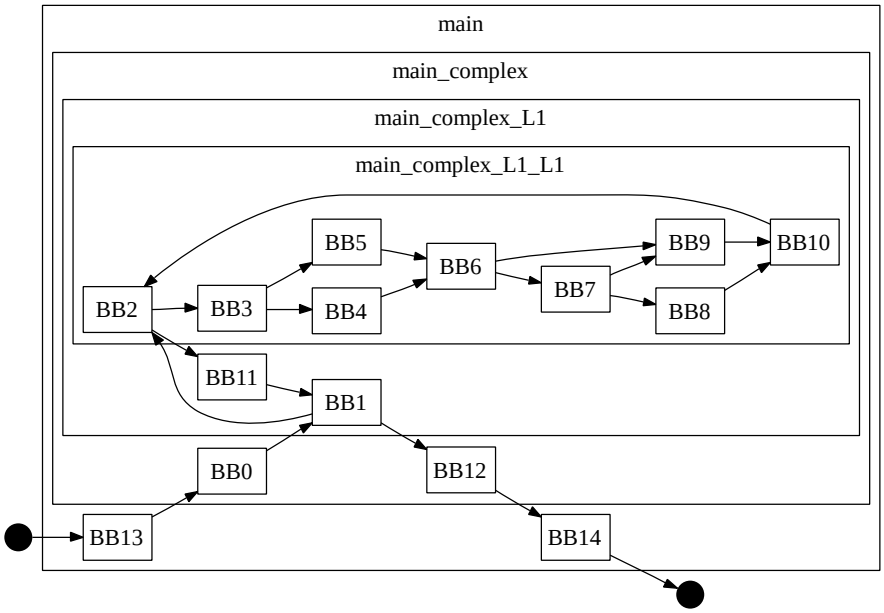

Figure 4. Scope graph for the example program

This algorithm has been successfully used to analyse all the programs in the Mälardalen WCET benchmarks suite [14] during the WCET Challenge 2006 [18]. However, during the industrial case study described in [2], we discovered that for some large programs with many possible input variable values, AE faced complexity problems, i.e., very long analysis times and large amounts of used memory. As an illustration of the inherent problem of the original algorithm, consider the example in Figure 1 again. Further assume that we have decided to use both loop body termination (i.e., `q`) and loop termination (i.e., `r`) as merge points. As explained in Section 2, the fourth time the condition is abstractly executed the analysis will spawn two abstract states, one taking the *true* branch (i=$[7..9]$), and one taking the *false* branch (i=$[10..10]$). Both states will eventually reach a merge point (`q` and `r` respectively). However, since they are not in the same merge point they cannot be merged at that time. Instead, both states are moved to the worklist, and will continue their executions in parallel.

We observe that all states resulting from abstractly executing the loop will sooner or later reach `r`. However, there is no mechanism in the original algorithm to force a state in `r` to wait for the other states to reach `r`. Thus, each state that reaches `r` will be continue executing the code following `r` in parallel with the states in the loop. This means that merging in itself is not guarantee to get few parallel states. Moreover, the code after `r` will be executed several times, with almost identical states, which means a lot of unneccessary work.

Ordering of merge nodes. We have designed and implemented an algorithm to solve the problem described above[1]. The basic idea behind the algorithm is to force a state to *wait* for all other states which sooner or later will reach the same merge node at which the state is located. This is achieved by creating an order between all merge nodes and force the processing of states to follow this order. For example, in Figure 1 we want to create an order so states at `r` should wait for all states still executing in the loop. All states resulting from the loop processing can then be merged at `r`, giving that there will be only one state that continues executing after `r`, instead of several.

The algorithm works by first creating an *immediate post-dominance (IPDom) tree* of all nodes in

[1]In the current implementation, merging for recursive programs is not supported.

```
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 5. Original algorithm for AE

the program CFG. Basically, a node n *post-dominates* another node m iff all paths from m to the program exit node intersect n. Similarly, a node n *immediately post-dominates* another node m iff n *post-dominates* m and there is no other post-dominator of m between n and m in the CFG. In the tree, n will be a direct parent of m iff n immediately post-dominates m. In our current implementation, we use the algorithm outlined by Lengauer and Tarjan [12] to create the IPDom tree, with an $O(e\ log\ v)$ time complexity, where e is the number of edges and v is the number of vertices in the CFG.

Secondly, we traverse the IPDom tree bottom-up. For each node traversed, we check if it is a merge node. If so, it is inserted to the end of a *list of merge nodes*. As a result, the list will hold all merge nodes, ordered so that if a node n post-dominates m, then m will be before n in the list. We note that there are often many merge-nodes which do not have any post-dominate relation to one another. Thus, the same set of merge nodes could result in many different orderings, depending on which order the different branches in the IPDom tree are processed. However, for all possible list orderings, it should hold that for all pair of nodes n, m in the list: if n post-dominates m, then m should be before n in the list. The graph traversal has an $O(v)$ complexity, where v is the number of vertices in the CFG.

Thirdly, the list is used to create a *priority queue*, where inserted states are indexed on the position of their corresponding merge node in the ordered list. This queue will replace the merge list in the original algorithm for AE in Figure 5. We also only pop one state at the time from the priority queue into the work list, instead of, as in the original algorithm, popping all states in the merge list to the work list simultaneously. Thus, when extracting a state from the queue, the state with the merge node closest to the beginning of the list will be popped. Moreover, if a state is to be inserted into the priority queue, and there already is a state in the queue at that merge point, we merge the two states and insert the resulting state into the queue (the merged states are deleted). Thus, the number of states stored in

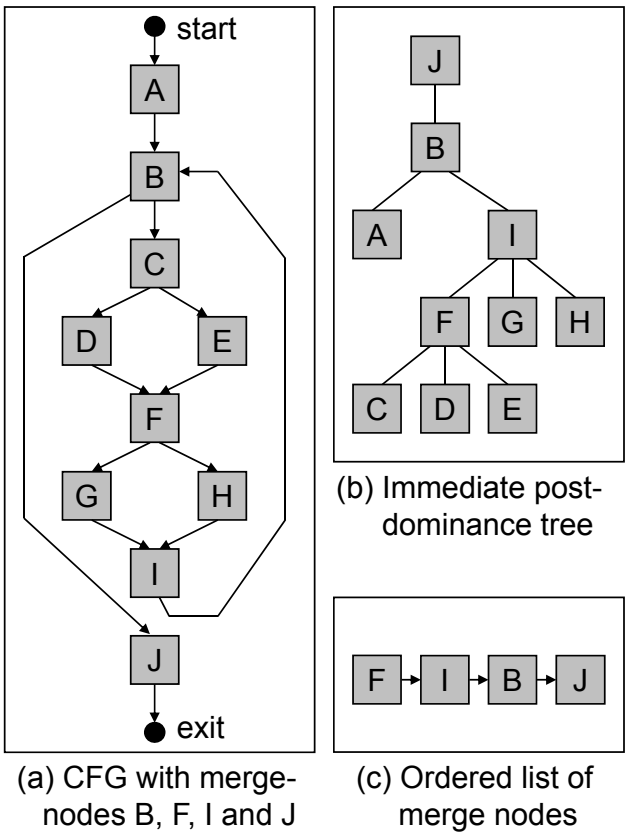

Figure 6. Example of merge node ordering

the priority queue will be less or equal to the number of merge nodes in the program.

We have implemented the priority queue as a binary heap, thereby getting an $O(log\ v)$ time complexity for both insertion and popping of states, where v is the number of merge nodes in the CFG [3].

Figure 6 gives an illustration of the ordering of merge nodes. Figure 6(a) shows a CFG with `B`, `F`, `I`, and `J` set as merge nodes, i.e., all possible join points in this program. Figure 6(b) shows the IPDom tree generated for the CFG. Figure 6(c) shows the ordered list of merge nodes resulting from the bottom-up traversal of the tree. The list gives, for example, that a state located in merge point `B` should be processed after states located in `F` or `I`, but before states located in `J`.

7. Evaluation

We have tested our merge strategies on two programs, `jcomplex` and `insertsort`, from the Mälardalen benchmarks using the ARM9 timing model of SWEET. This timing model has not been validated against real hardware. However, we consider it to be a sufficiently realistic "abstract architecture" for the purpose of evaluating flow analysis methods.

For the `jcomplex` program in Figure 3, the input space (i.e., number of possible input combinations) is $19^2 = 361$ values. SWEET's AE analysis time was 7.2 seconds when no merging was used, and the resulting calculated WCET estimate was 918 cycles for the ARM9 target CPU. Table 1 shows the results of the different analyses of `jcomplex`. **ATime** is the analysis time in seconds for AE, and **WCET** is the calculated WCET estimate for ARM9 in cycles. This information is shown for the

Merge type		Merge node type					
		FE	FT	LBT	LT	LBI	ALL
Unordered	ATime	8.7	15.0	0.72	0.18	0.84	0.77
merge	WCET	918	918	1053	2087	1053	1053
Ordered	ATime	8.7	8.3	0.13	0.18	0.18	0.16
merge	WCET	918	918	3711	2087	5395	5395

Table 1. Analysis results for `jcomplex`

Merge type		Merge node type					
		FE	FT	LBT	LT	LBI	ALL
Unordered	ATime	-	-	1.6	0.08	1.9	1.7
merge	WCET	-	-	332	332	332	332
Ordered	ATime	-	-	0.09	0.08	0.09	0.09
merge	WCET	-	-	332	332	332	332

Table 2. Analysis results for `insertsort`

following five merge point selections: **FE** = at function entries, **FT** = after function exits, **LBT** = after loop bodies, **LT** = after loop exits, **LBI** = after if-statements and loop bodies[2], and **ALL** = after all merge point types.

In the table, we see that merging at function entries and exits (**FE** and **FT**) gives an exact result compared to no merging, however to the cost of long analysis times (even exceeding the no merge time). The other merge selections are efficient, but yield some overestimation. This is mainly due to the conditional updates of the variables `a` and `b` inside the loops, where merging yields overestimation of these variables, which results in an overestimated outer loop bound. We can note that ordered merge gives a larger overestimation than unordered merge. This is probably because the ordered merge gives more merging, with fewer concurrent states, faster analysis but larger overestimation. Ordered merge is the fastest method in all cases.

Table 2 shows the results for `insertsort`, which is a sorting program using the insertion sort method for an array of 10 elements where each element is a positive 32 bit integer (i.e., given a value of $[1..2147483647]$). For this program, the input space is around 10^{93} values. Thus, it is not possible to run the program with all inputs. Nor is it possible to analyse it without merge. However, using the knowledge of the worst case behaviour for insert sort (an inversely sorted array), we can analyse the program using SWEET with the worst case input, and deduce a tight WCET estimate of 332 cycles.

In the table, we see that only some types of merging actually gives a result within a short time ('-' means that the analysis time exceeded 10 minutes). We can see that merging at either loop body termination and loop termination points (**LBT**, **LT**, **LBI**, and **ALL**) gives results in a very short time. This is not surprising, since the number of iterations in the nested loops in the programs is very high, and efficient merge should lower the analysis times considerably. We also see that this efficiency does not give any WCET overestimation penalty for this program. Ordered merge is fastest in all cases.

Table 3 shows the results for `esab_mod`, a larger program provided by one of our industry partners. The program consists of 3064 lines of C code, including 11 functions, 519 conditionals and one loop. The outcome of many of the conditionals are input dependent, and abstract execution of these

[2]We combine these types to ensure that merging always takes place after if-statements in the same iteration.

Merge type		Merge node type					
		FE	FT	LBT	LT	LBI	ALL
Unordered merge	ATime	-	-	-	-	-	-
	WCET	-	-	-	-	-	-
Ordered merge	ATime	-	-	-	-	163	161
	WCET	-	-	-	-	165795	165795

Table 3. Analysis results for `esab_mod`

conditionals may therefore generate many new states. The loop is located in a function with loop bound that is a dependent on the argument to the function. The function itself is called from 372 different call-sites. Since AE does its loop bound analysis fully context sensitive, i.e., we do a seperate analysis for each individual path to the function in the call graph, we get a large number of different calling contexts (8942) for the loop. In the example runs eight variables were given input value ranges, giving an input space of around $1.5 * 10^{12}$ values. Due to this, it was not possible to run the program with all inputs or to manually determine the input value combination that generated the WCET.

For none of the merge point options were we able to finish AE analysis when using unordered merge ('-' means that the analysis time exceeded 1 hour). For ordered merge we were able to finish the analysis within 3 minutes for both the **LBI** and **ALL** options with the same resulting WCET estimate. The ordering of merge nodes took around 82 seconds, and has been included in the total running time for AE. For single value inputs the analysis took around 8-10 seconds (depending on used input values) excluding the time for ordering merge nodes.

All measurements were performed on a 3 GHz PC with 1 Gb RAM, running Linux Ubuntu.

8. Conclusions and Future Work

In this paper, we have described the merge techniques used during abstract execution, a flow analysis method used in SWEET. We have described how to trade the precision of the results for faster analysis, by the use of different placement of merge points and a new merging technique based on sorting merge points. We have shown one example where we can use maximum merging to give a radically shorter analysis time and still get a precise WCET estimate. For another example, however, merging lead to less tight WCET estimates.

Moreover, we have shown that use of merge points is not alone a guarantee to obtain few concurrent states. To handle this, we have presented a new method, based on ordering of merge points, to reduce the number of concurrent states. We have shown that this method is able to significantly shorten the analysis time of large programs with large input spaces.

For future work we plan to investigate the effect of our different merging techniques on different industrial codes, such as the task codes presented in [2]. This will allow us to see, in more detail, how our methods scales for large programs with large input spaces. We will also study the effect of different merge point plecements on different program types and code constructs. This will allow us define guidelines on how to select merging strategy for achieving an optimal combination of analysis time and precision.

References

[1] P. Altenbernd. On the false path problem in hard real-time programs. In *Proc. 8th Euromicro Workshop of Real-Time Systems*, pages 102–107, June 1996.

[2] Dani Barkah, Andreas Ermedahl, Jan Gustafsson, Björn Lisper, and Christer Sandberg. Evaluation of automatic flow analysis for WCET calculation on industrial real-time system code. In *Proc. 20th Euromicro Conference of Real-Time Systems, (ECRTS'08)*, July 2008.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 2nd Edition*. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7.

[4] Patrick Cousot and Radhia Cousot. 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*, pages 238–252, Los Angeles, January 1977.

[5] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala, Sweden, April 2002. ISBN 91-554-5228-0.

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

[7] Johan Fredriksson, Thomas Nolte, Andreas Ermedahl, and Mikael Nolin. Clustering worst-case execution times for software components. In *Proc. 7th International Workshop on Worst-Case Execution Time Analysis, (WCET'2007)*, Pisa, Italy, July 2007.

[8] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Dept. of Information Technology, Uppsala University, Sweden, May 2000.

[9] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proc. 27th IEEE Real-Time Systems Symposium (RTSS'06)*, December 2006.

[10] C. Healy, R. Arnold, Frank Müller, David Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, January 1999.

[11] D. Kebbal. Automatic flow analysis using symbolic execution and path enumeration. In *Proc. of the 2006 International Conference Workshops on Parallel Processing (ICPPW'06)*, pages 397–404, Washington, DC, USA, 2006. IEEE Computer Society.

[12] Thomas Lengauer and Robert E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. on Programming Languages and Systems*, pages 121–141, July 1979.

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

[14] Mälardalen University. WCET project homepage, 2007. `www.mrtc.mdh.se/projects/wcet`.

[15] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis, 2nd edition*. Springer, 2005. ISBN 3-540-65410-0.

[16] Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[17] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *Proc. 4th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, (CASES'01)*, November 2001.

[18] L. Tan. The worst case execution time tool challenge 2006: The external test. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, November 2006.