Faster WCET Flow Analysis by Program Slicing

Christer Sandberg Andreas Ermedahl Jan Gustafsson Björn Lisper

Dept. of Computer Science and Electronics, Mälardalen University, Västerås, Sweden

{christer.sandberg,andreas.ermedahl,jan.gustafsson,bjorn.lisper}@mdh.se

Abstract

Static Worst-Case Execution Time (WCET) analysis is a technique to derive upper bounds for the execution times of programs. Such bounds are crucial when designing and verifying real-time systems. WCET analysis needs a program flow analysis to derive constraints on the possible execution paths of the analysed program, like iteration bounds for loops and dependences between conditionals.

Current WCET analysis tools typically obtain flow information through manual annotations. Better support for automatic flow analysis would eliminate much of the need for this laborious work. However, to automatically derive high-quality flow information is hard, and solution techniques with large time and space complexity are often required.

In this paper we describe how to use program slicing to reduce the computational need of flow analysis methods. The slicing identifes statements and variables which are guaranteed not to influence the program flow. When these are removed, the calculation time of our different flow analyses decreases, in some cases considerably.

We also show how program slicing can be used to identify the input variables and globals that control the outcome of a particular loop or conditional. This should be valuable aid when performing WCET analysis and systematic testing of large and complex real-time programs.

Categories and Subject Descriptors J.7 [*COMPUTERS IN OTHER SYSTEMS*]; C.3 [*SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS*]: REAL-TIME AND EMBEDDED SYSTEMS

General Terms Verification, Reliability

Keywords Hard real time, worst-case execution time analysis

1. Introduction

A *Worst-Case Execution Time* (WCET) analysis finds an upper bound to the worst possible execution time of a computer program. Reliable WCET estimates are crucial when designing and verifying embedded and real-time systems, especially safety-critical such systems like vehicles, military equipment and industrial power plants [16].

The traditional way to determine the timing of a program is by measurements. This is labour-intensive and error-prone work, and even worse, it cannot guarantee that the true WCET has been found since, in general, it is impossible to perform exhaustive testing.

An alternative technique is *static WCET analysis*, which determines a timing bound from mathematical models of the software and hardware involved. Given that the models are correct, the analysis will derive a timing bound that is *safe*, i.e., greater than or equal to the true WCET. To be useful, the bound must also be *tight*, i.e., provide little or no overestimation compared to the true WCET.

To statically derive a timing bound for a program, information on both the *hardware timing characteristics*, such as the execution time of individual instructions, as well as the program's *possible execution flows*, to bound the number of times the instructions can be executed, needs to be derived. The latter includes information about the maximum number of times loops are iterated, which paths through the program that are feasible, execution frequencies of code parts, etc.

The goal of a *flow analysis* method is to calculate such *flow information* as automatically as possible. For complex programs, it is hard (and in the general case impossible, due to the halting problem), to derive this information. To be feasible, flow analysis methods therefore calculate approximations of the flow information, and allow additional information to be given in terms of *manual annotations*. In general, there is a trade-off between the precision of a flow analysis method and its computational need; a coarser analysis will typically run faster but provide less detailed information. Good flow analysis is hard to do, and solution techniques with large time and space complexity are often required [20, 32].

The work presented here uses *program slicing* [41]. Program slicing finds the subset of a program (or an enclosing subset) that can affect some given part of the program, e.g., a specific condition, a loop or all loops. It is used in various areas, like debugging, testing, software measurement, program comprehension, software maintenance, and program parallelization [42]. However, to our knowledge, it has not been used for WCET analysis before.

In this article we introduce program slicing as a technique to reduce the computational need of flow analysis methods. The concrete contributions of this article are:

- We introduce program slicing as a technique to remove statements and variables which can be guaranteed to not affect the program flow. Thereby, we reduce the computational need of subsequent flow analyses, without decreasing their precision.
- We show how to slice w.r.t. a selected subset of all program constructs, e.g., for loops only, allowing for coarser but still safe flow analyses to be made.
- We show how to slice w.r.t. a particular program construct, e.g., a single loop or conditional statement. This allows us

This research has been supported by the Advanced Software Technology Center (ASTEC) in Uppsala [2], as well as by the KK-foundation through grant 2005/0271. ASTEC is a Vinnova (Swedish Agency for Innovation Systems) initiative [40].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCTES'06 June 14–16, 2006, Ottawa, Ontario, Canada.

Copyright © 2006 ACM 1-59593-362-X/06/0006. . . $5.00.

uncover dependences that need to be considered when selecting a suitable flow analysis method for that construct.

- We present an alternative program slicing algorithm, yielding results that are almost equally precise as for the standard slicing algorithms, but being simpler to implement.
- We show how to use program slicing to identify the input variables and globals that may affect the program flow.
- We evaluate the effect of our different program slicings, including the amount of code removed and the execution time reduction for our flow analysis methods.

The rest of this paper is organized as follows: Section 2 gives an introduction to static WCET analysis and presents related work. Section 3 motivates our program slicing. Sections 4 and 5 present our program model and the standard slicing algorithm used. Sections 6 and 7 present different slicing alternatives for flow analysis. Section 8 discusses flow information and code removal. In Section 9, we describe how to find input variables and globals that may affect the program flow. Section 10 presents the WCET analysis tool in which we have implemented the program slicing. Section 11 presents our measurements and evaluations. Finally, Section 12 gives our conclusions and presents future work.

2. WCET Analysis Overview and Related Work

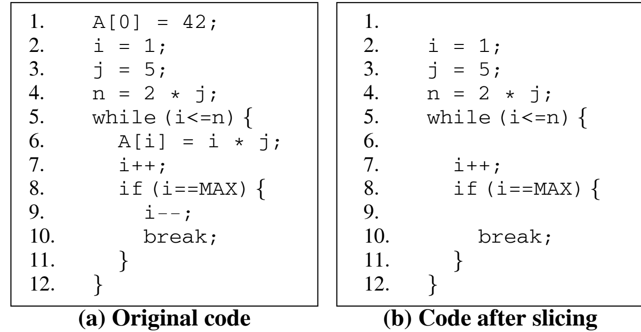
Any WCET analysis must deal with the fact that most computer programs do not have a fixed execution time. *Variations* in the execution time occur due to different input data, the characteristics of the software, as well as of the hardware upon which the program is run. Thus, both the software and the hardware properties must be considered in order to derive a safe WCET estimate.

Consequently, static WCET analysis is usually divided into three phases: a (fairly) machine-independent *flow analysis* of the code, where information about the possible program execution paths is derived, a *low-level analysis* where the execution times for instructions or sequences of instructions are decided from a performance model for the target architecture, and a final *calculation* phase where the flow and timing information are combined to yield a WCET estimate.

In low-level analysis researchers have studied effects of various hardware enhancing features, like caches, branch predictors and pipelines [3, 10, 27, 38]. A frequently used calculation method is IPET (Implicit Path Enumeration Technique), using arithmetical constraints to model the program structure, the program flow and low-level execution times [12, 24, 38].

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 WCET estimates [12]. Since the flow analysis does not know the execution path that gives the longest execution time, the information must be a safe (over)approximation including (at least) all feasible program executions, e.g., loop iteration bounds must be equal to or larger than what is actually possible. Flow analysis can also identify *infeasible paths*, i.e., paths executable according to the control-flow graph structure, but not feasible when considering the semantics of the program. Other useful flow information include execution frequencies of different code parts.

A number of flow analysis methods are used in practice, each with different precision and computational need. Whalley et al. [22, 23] use data flow analysis and special algorithms to calculate bounds for single and nested loops in conjunction with a compiler. The aiT WCET tool has a loop-bound analysis based on a combination of an interval-based abstract interpretation and pattern-matching working on the machine code [38]. The Bound-T WCET tool has a loop-bound analysis based on Presburger arithmetics working on the machine code [39]. Altenbernd and Stappert [35] use symbolic execution on source code level to derive infeasible paths. Lundqvist and Stenström [27] find loop bounds and infeasible path by symbolic simulation on the binary code. We have developed several flow analysis methods to derive flow information [20], and we use the outlined program slicing as a preceding step to these methods (see Section 10.1).



```
1.   A[0] = 42;
2.   i = 1;
3.   j = 5;
4.   n = 2 * j;
5.   while (i<=n) {
6.     A[i] = i * j;
7.     i++;
8.     if (i==MAX) {
9.       i--;
10.      break;
11.    }
12.  }
```

(a) Original code

```
1.
2.   i = 1;
3.   j = 5;
4.   n = 2 * j;
5.   while (i<=n) {
6.
7.     i++;
8.     if (i==MAX) {
9.
10.      break;
11.    }
12.  }
```

(b) Code after slicing

Figure 1. Example of program slicing

Our research group aims to develop flow analysis methods which can cope with the complexity of industrial real-time embedded code [28]. Recent case-studies, on static WCET analysis of embedded industrial codes [5, 13], have shown that it is especially important to develop better such analyses, thereby reducing the need for manual annotations.

3. Why do Program Slicing for Flow Analysis?

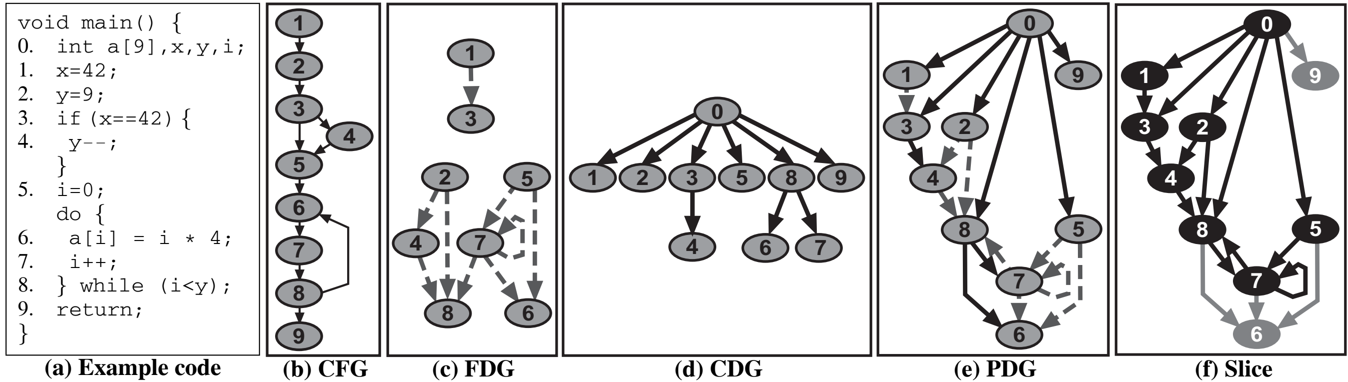
Any reasonable WCET flow analysis should need to traverse the program code at least once. Thus, its time complexity should be at least linear in the size of the program, and more precise analyses can be expected to have a considerably higher time complexity. Therefore, if the program to be analysed can be shrunk, a gain in analysis time can be expected.

The conditions in the code govern the program flow. However, there may be statements in the code that never will affect the outcome of any condition. If these statements are removed, then the code to analyze will be smaller, but the outcome of the flow analysis should still be the same. However, the analysis time will be shorter. In this process, it may also turn out that some program variables are found never to affect the conditions. These variables can then also be removed from the program, which is beneficial for the execution time of some flow analyses.

Program slicing, with respect to the conditionals in the program, can be used to identify the parts of the program that can possibly affect these conditionals. The rest of the program can safely be removed before the flow analysis takes place. If the program slicing takes less time than the gain in flow analysis time, then the slicing has improved the total analysis time.

Figure 1(a) gives an illustrative code example. The values in the array `A` cannot affect the number of times that the loop body will be executed. Consequently, the statements at row 1 and 6 can be removed, as illustrated in Figure 1(b). However, to derive a loop bound, an analysis must consider that the loop can be exited both if the `while` condition becomes false, or if the `if` condition becomes true. This means that these two conditions cannot be removed, and neither can any statements or variables that may affect these conditions, directly or indirectly. Therefore, since the variable `j` is used to update the `n` variable, and `n` is used in the loop condition, statements at row 2–4 cannot be removed. `MAX` is an input variable defined elsewhere in the program.

Similar to the statement at row 7, the statement at row 9 updates the `i` variable. However, the update can only be made after the loop



(a) Example code **(b) CFG** **(c) FDG** **(d) CDG** **(e) PDG** **(f) Slice**

Figure 2. PDG slicing illustration

has exited. This means the statement will not affect the number of iterations in the loop, and it can therefore be removed. This demonstrates that sometimes it is possible to remove some, but not all, occurrences of a variable.

4. Program Model

The slicing algorithms described in Sections 5 and 7 are fairly generic, and can be adapted to work on a variety of programming languages. Notably, they can handle unstructured code and pointers, thus, they are apt for the kind of low-level code that is common in embedded applications. The algorithms are capable of performing interprocedural slicing, thus, they are also applicable to code with functions and procedures.

Our current implementations of the algorithms analyze the intermediate compiler format NIC, see Section 10. However, for the purpose of explaining the algorithms, we consider programs as represented by conventional *control flow graphs* (CFG's). Each node in such a graph is decorated with either a *conditional* (boolean expression) or an *assignment*. Each program holds a number of *program variables*, whose values are updated by the assignments, and whose values are retrieved when evaluating conditionals and right-hand sides of assignments. We also allow pointer variables to appear in the conditions and assignments: these variables can be assigned, accessed, and dereferenced just as in, for instance, C. The flow graph edges constitute *program points*: each program point has unique predecessor and successor nodes, and it holds the program state, produced by its predecessor, which is used as input to its successor. Program analyses often produce information about the possible states of a program in different program points.

5. Slicing using Program Dependence Graphs

The standard algorithm for program slicing of imperative programs, originally suggested by Ottenstein and Ottenstein [31], uses the *program dependence graph* (PDG) introduced by Ferrante et al. [14]. Notably, this algorithm handles unstructured code since it operates on general CFG's: thus, it can be used also for low-level code, as long as a CFG can be produced for the code. We have used an extension of the algorithm to handle interprocedural slicing by Horwitz et al. [25]. In the following, we give a short account for this algorithm, and our implementation. As a running example, we use the example code in Figure 2(a).

5.1 Program dependence graph

For any CFG, its PDG is a directed graph, with the same set of nodes. The PDG defines the dependences between nodes (i.e., given a node n in the program, which other nodes have to be executed before n to obtain a correct result). The dependences are of two kinds: *data flow* dependences, and *control* dependences. These dependences can be described by two separate graphs: the *flow dependence graph*, (FDG), and the *control dependence graph* (CDG). The PDG is then the union of these graphs. The CFG of our example code is shown in Figure 2(b).

5.2 Data dependences

A node n_1 in a CFG is *data dependent* on the node n_2 if a variable written by an execution of n_2 may be read by an execution of n_1. The FDG captures all data dependences between nodes in the CFG. It can be calculated by a standard reaching definition analysis, see [30]. (Note, however, that this analysis must deal with pointers, see Section 5.6). The FDG for our example program is shown in Figure 2(c).

5.3 Control dependences

A node n_1 is *control dependent* on the condition node n_2 if the execution of n_2 may decide whether or not n_1 will be executed. In Figure 2(d) we show the CDG for our example program.

The *post-dominance relation* can be used to calculate control dependences. n_1 *post-dominates* n_2 if all paths from n_1 to the exit node of the CFG visits n_2 [1]. n_2 is control dependent on n_1 iff (1) there exists a directed path P from n_1 to n_2 with any n_3 in P (excluding n_1 and n_2) post-dominated by n_2, and (2) n_1 is not post-dominated by n_2 [14]. An algorithm to calculate the control dependences, based on post-dominance, is given by Gupta [18].

5.4 Calculating the slice

As mentioned, the PDG is the union of the FDG and CDG. The PDG for our running example is shown in Figure 2(e). The slice (or *backwards slice*) w.r.t. a set of nodes N in the CFG is simply the set of nodes that are backwards reachable from N in the PDG. A simple backwards search finds this set. For the purpose of WCET flow analysis, slicing is done w.r.t. sets of condition nodes, see Section 6.

In our running example, the condition nodes are found at rows 3 and 8. The computed slice w.r.t. these nodes is shown in Figure 2(f).

5.5 Interprocedural analysis

In order to perform interprocedural program slicing, we form the *system dependence graph* (SDG) from the PDG's of the different functions and procedures in the program. Basically, the SDG is formed by connecting the call sites to callees through a number of new nodes, which represent making the call, entering the called function, copying the actual arguments to local variables in the called function, returning results, etc. See [25] for details.

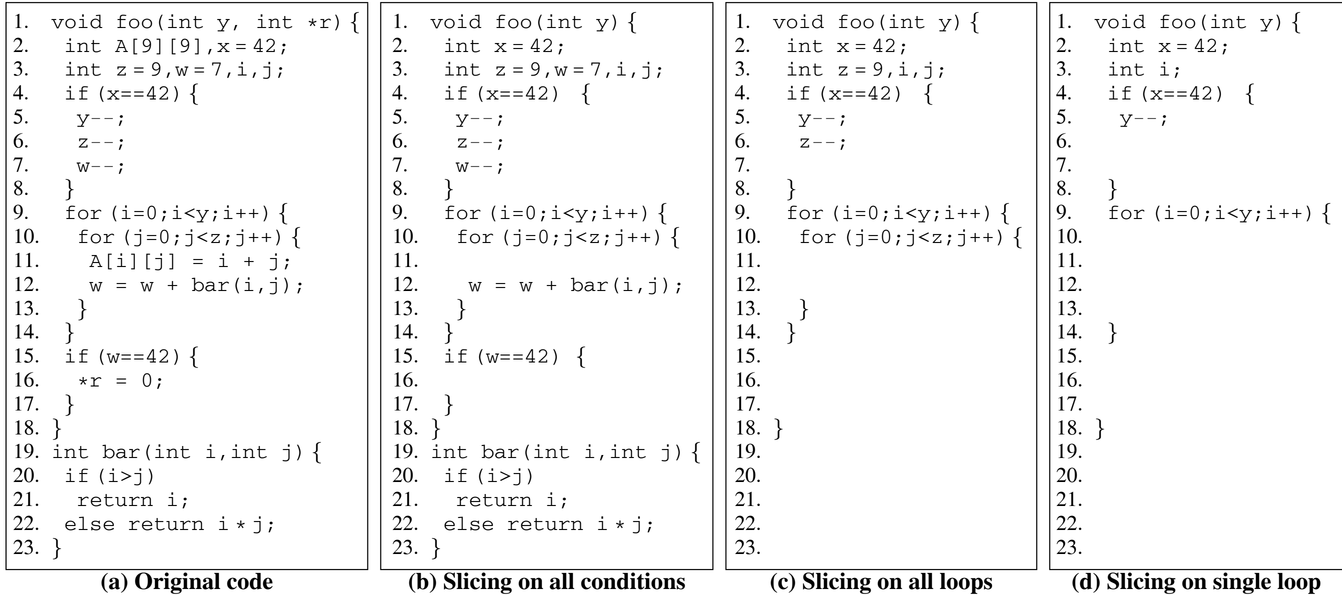


Figure 3. Program Slicing Alternatives

In [25] it is described how to perform a context-sensitive slicing, which keeps dependence information from different call sites separate. Our current implementation is context-insensitive, which may yield a coarser slicing but is simpler to implement and in general less costly.

5.6 Involving pointers

Embedded system programmers often use pointers to manipulate data and decide the outcome of conditionals [34]. Pointers, as well as the variables they point to, may affect the outcome of conditions. For example, consider the following code fragment:

```
int* p = &i; *p = 5; if(i < j)...
```

In the code, the variable `i` is assigned a value through the pointer `p`, which therefore indirectly decides the outcome of the condition.

Program slicing for realistic embedded programs must therefore be able to handle dependences through pointers. Pointers affect data dependences, since an access through a dereferenced pointer may touch different program variables depending on the current value of the pointer. Our slicing method assumes, for each program point q and pointer variable p, that there is a so-called *points-to set* which contains the set of variables possibly pointed to by p in the program point q. Using this information, the slicing algorithm can safely overestimate the data dependencies by adding a data dependence arc for each def-use chain [29] concerned by the use of the pointer.

There are a number of different pointer analyses to compute points-to sets. Our slicing method works with any of these, as long as it produces a safe overapproximation of the variables possibly pointed to: obviously, though, a more precise pointer analysis will, in general, yield a more precise slicing. Our current implementation uses the pointer analysis of Steensgaard [37]. This analysis is flow-insensitive and fast, but may produce less precise results than more sophisticated analyses. Our current implementation of the analysis is context-insensitive.

5.7 Global variables, arrays and other aggregate objects

Global variables may carry dependences between function calls, and an interprocedural analysis must take them into account. One approach is to consider them as extra function parameters, and use the method of [25] to handle them. Our current implementation uses a simpler, context- and flow-insensitive approach, where any use of a global variable is considered dependent on any definition of the same variable.

Our implementation handles arrays and other aggregate objects (e.g., structs) as single data objects. Accesses to elements `A[i]` and `A[j]` will thus be considered accesses to the same variable `A`, despite the fact that `i` and `j` may have different values.

6. Program Slicing Alternatives

As explained in Section 2, the goal of a flow analysis is to derive bounds on the number of times different parts of the program can be executed. We now show how the precision of this analysis can be traded for running time by slicing w.r.t. different kinds of conditions.

As an illustrating example, consider the C code fragment in Figure 3(a). It consists of two functions `foo()` and `bar()`. The input parameter `r` to `foo()` is a pointer. We assume that `r` cannot point to any local variable or parameter of these functions, and that the value stored through `r` in `foo()` is not needed for the analysis of the caller of `foo()`.

A flow analysis of the code in Figure 3(a) may derive iteration bounds for the different loops in rows 9 and 10, as well as bounds for the number of times the true and false exits can be taken from the `if` conditions in rows 4, 15 and 20.

In order to find a finite upper timing bound for a program, finite upper bounds for the iteration counts of all loops (and for the recursion depth of all recursive functions), are needed. In order to find such bounds, all the exit conditions of all the loops must in general be analyzed. Analyzing the remaining conditions in the program can give more precise flow information, and thus a tighter WCET estimate. However, if they are ignored then the WCET estimate will still be finite, possibly still useful, and the analysis will in general be less costly. For instance, it is not necessary to analyze the condition in row 20 in Figure 3(a) to obtain a finite WCET estimate for `foo()`. However, an analysis of the condition can give bounds how often the respective paths in `bar()` can be taken in the nested loop of `foo()`. If these bounds are lower than the loop iteration bounds, then a better WCET estimate may be

obtained, since the multiplication in the `else` branch most likely makes it more costly than the other branch to execute.

6.1 Slicing with respect to all conditions

Figure 3(b) shows the result when slicing w.r.t. all conditions for the code in Figure 3(a), which has two loop exit conditions in rows 9 and 10, and three `if` conditions in rows 4, 15 and 20. A flow analysis of this code should give the same resulting program flow constraints as for the non-sliced version.

The slicing removes both the declaration and the update of the array `A`, since it does not affect the outcome of any condition. For the same reason, the slicing also completely removes the input parameter `r`.

6.2 Slicing with respect to all loop exit conditions

Figure 3(c) illustrates the result of slicing w.r.t. all the loop exit conditions but no others. More code has been removed than by the slicing in Figure 3(b), including the call to `bar()` and the complete `bar()` function.

Note however, that even though we sliced only w.r.t. loop conditions, the resulting code still contains an `if` condition in row 4. This is because the loop conditions are data dependent on the assignments in rows 5 and 6, which in turn are control dependent on the `if` condition.

A flow analysis of the code in Figure 3(c) should produce the same loop iterations bounds as a flow analysis of the code in Figure 3(a). However, some additional bounds on the outcome of conditions would be lost. A WCET calculation would be able to calculate a finite upper timing bound, however potentially less tight. On the other hand, the flow analysis will probably run faster since it will have to analyze less code.

6.3 Slicing with respect to single loops

It is possible to slice only w.r.t. to the exit conditions of a single loop, or loop nest. The purpose is then not to use the sliced program directly for flow analysis, but rather to find the individual dependence pattern for the loop construct. This allows us to select the flow analysis method most suitable to bound the iteration count of this loop: for instance, we may have a library of pre-calculated iteration count bounds for certain common loop patterns.

Figure 3(d) shows the result of slicing only w.r.t. the loop exit condition in row 9. The slicing removes even more code than the slicing on all loops shown in Figure 3(c). This is because the outcome of the loop exit condition in row 9 is independent of the loop exit condition outcome and the data updated in the inner loop at row 10. Consequently, the inner loop can be removed as well as all occurrences of the variables `j` and `z`.

If we instead would have sliced upon the inner loop in row 10, both loops would have been kept after slicing. This is because the inner loop cannot be entered unless the outer loop is entered, and the exit condition of the inner loop is therefore control dependent on the exit condition of the outer loop.

7. A Simplified Slicing Algorithm

As explained in Section 3 it is sometimes possible to remove some, but not all occurrences, of a variable. The PDG program slicing algorithm of Section 5 is flow-sensitive and allows us to detect and represent dependences between different variable occurrences. Consequently, the algorithm can sometimes remove certain occurrences of the same variable but not others.

A alternative solution is to not differentiate between different variable occurrences. A slicing resulting from this approach is flow-insensitive, and keeps all occurrence of a variable if at least one occurrence has to be kept. For example, a slicing w.r.t. the loop condition in Figure 1(a) would not remove the statement at row 9, since the variable `i` is used in both conditions. This alternative approach may remove less statements than the PDG slicing. However, it should be easier to implement, since its dependence representation is simpler.

In order to investigate how much precision that is lost when using this simplified slicing approach, we have designed and implemented the algorithm SIMPLESLICE, which is given in Figure 4. For simplicity, we only give the intraprocedural version: it can be extended to an interprocedural analysis by standard means [30].

SIMPLESLICE distinguishes between conditional and assignment statements. For any program point q, we define $\text{STMT}(q)$ to be the succeeding statement.

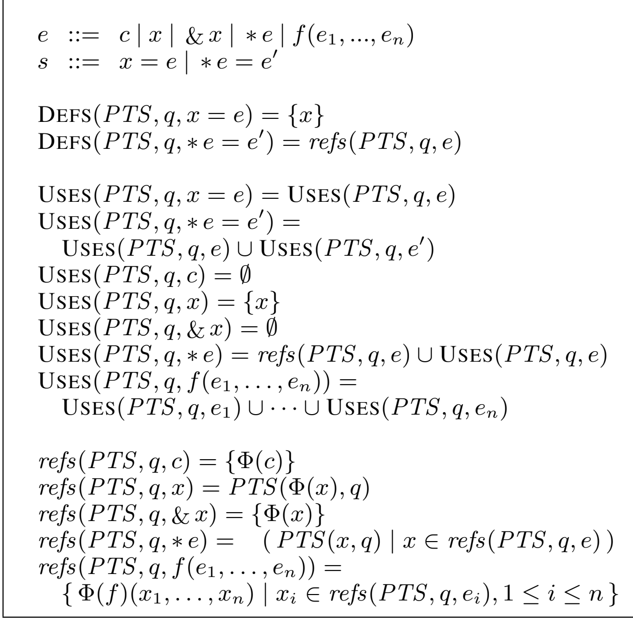
SIMPLESLICE uses the set P all statements that should not be removed, and the set N to hold all variables that has been used. In the first pass, these sets are computed. N is initialized to the set of all variables appearing in any conditional in the code, and P is initialized to the empty set. The algorithm then performs a fixpoint iteration over the assignment statements in the code, where statements possibly affecting a variable in N are added to P, and the used variables of these statements are added to N, until these sets do not grow any more. Clearly, this process must terminate since there are only a finite number of variables and assignments in a program. In the second pass, all assignment statements not in P can be safely removed from the code. The declaration of any variable that is not present in the slice can be removed.

Variables may be read or written through pointers. We assume, for each program point q and pointer variable p, that there is a points-to set $PTS(p, q)$ which contains the set of variables possibly pointed to by p in the program point q.

The DEFS function calculates a set of locations where the result of an assignment may be stored. If the left-hand side of the assignment contains dereferenced pointers, then DEFS uses PTS to calculate possible locations where the result might be stored. For example, if `x` and `i` are variables and `p` is a pointer with $PTS(\mathtt{p}, q) = \{\mathtt{a}, \mathtt{b}\}$, then $\text{DEFS}(PTS, q, \mathtt{x} = *\mathtt{p} + \mathtt{i}) = \{\mathtt{x}\}$ and $\text{DEFS}(PTS, q, *\mathtt{p} = 17) = \{\mathtt{a}, \mathtt{b}\}$.

SIMPLESLICE($CODE$, ASS, $VARS$, $CVARS$, PTS)
Input $CODE$ - the code to be sliced
  ASS - all program points preceding an assignment in the code
  $VARS$ - all variables in the code
  $CVARS$ - all variables in conditionals in the code
  PTS - a mapping from pointer variables and program points to points-to-sets
Output $SLICE$ - the modified code
0. $SLICE := CODE$
1. $N := CVARS$
2. $P := \emptyset$
3. $prev := 0$
4. **while** $|N| > prev$ **do**
5. $prev := |N|$
6.   **foreach** $q \in ASS$ **do**
7.   **foreach** $v \in \text{DEFS}(PTS, q, \text{STMT}(q))$ **do**
8.   **if** $v \in N$ **then**
9. $P := P \cup \{\text{STMT}(q)\}$
10. $N := N \cup \text{USES}(PTS, q, \text{STMT}(q))$
11. **foreach** $q \in ASS$ **do**
12.   **if** $\text{STMT}(q) \notin P$ **then**
13.   REMOVESTMT($SLICE$, $\text{STMT}(q)$)
14. **return** $SLICE$

Figure 4. SIMPLESLICE - a slicing algorithm not differentiating between different variable instances



$$
\begin{array}{lll}
e & ::= & c \mid x \mid \&x \mid *e \mid f(e_1, ..., e_n) \\
s & ::= & x = e \mid *e = e'
\end{array}
$$

$$\text{DEFS}(PTS, q, x = e) = \{x\}$$
$$\text{DEFS}(PTS, q, *e = e') = \mathit{refs}(PTS, q, e)$$

$$\text{USES}(PTS, q, x = e) = \text{USES}(PTS, q, e)$$
$$\text{USES}(PTS, q, *e = e') = \text{USES}(PTS, q, e) \cup \text{USES}(PTS, q, e')$$
$$\text{USES}(PTS, q, c) = \emptyset$$
$$\text{USES}(PTS, q, x) = \{x\}$$
$$\text{USES}(PTS, q, \&x) = \emptyset$$
$$\text{USES}(PTS, q, *e) = \mathit{refs}(PTS, q, e) \cup \text{USES}(PTS, q, e)$$
$$\text{USES}(PTS, q, f(e_1, \ldots, e_n)) = \text{USES}(PTS, q, e_1) \cup \cdots \cup \text{USES}(PTS, q, e_n)$$

$$\mathit{refs}(PTS, q, c) = \{\Phi(c)\}$$
$$\mathit{refs}(PTS, q, x) = PTS(\Phi(x), q)$$
$$\mathit{refs}(PTS, q, \&x) = \{\Phi(x)\}$$
$$\mathit{refs}(PTS, q, *e) = \bigcup (\, PTS(x, q) \mid x \in \mathit{refs}(PTS, q, e) \,)$$
$$\mathit{refs}(PTS, q, f(e_1, \ldots, e_n)) = \{\, \Phi(f)(x_1, \ldots, x_n) \mid x_i \in \mathit{refs}(PTS, q, e_i), 1 \le i \le n \,\}$$

Figure 5. Definitions of DEFS and USES

The USES function returns the set of variables possibly used (read) in a statement. These values may be used to calculate both the right-hand side, and the address where to store the result. Similarly to DEFS, it uses PTS to calculate variable uses through dereferenced pointers. For instance, for the above `x`, `i`, and `p` we get $\text{USES}(PTS, q, \mathtt{x} = 42) = \emptyset$, $\text{USES}(PTS, q, \mathtt{p} = \mathtt{p} + \mathtt{i}) = \{\mathtt{p}, \mathtt{i}\}$, and $\text{USES}(PTS, q, *\mathtt{p} = *\mathtt{p} - \mathtt{x}) = \{\mathtt{a}, \mathtt{b}, \mathtt{p}, \mathtt{x}\}$.

DEFS and USES are more formally defined, over a simple abstract language of assignments, in Figure 5. In the presentation we use s, e, x and c to denote an arbitrary statement, expression, variable and constant respectively. We use $*$ to denote the dereference operator, `&` the address-of operator, and f to denote any other, arbitrary operator in our language, e.g., addition or subtraction.

$\Phi(f)$ maps f to the domain used to represent addresses in the analysis. For example, when analysing binary code, the domain can be sets of memory addresses, and then Φ will map, e.g., $+$ to address addition. When analysing intermediate code with symbolic addresses, (like we do in our WCET analysis tool), the domain may be sets of symbolic addresses.

What is the worst-case complexity of SIMPLESLICE? In the first pass, the number of fixpoint iterations can at most equal the number of variables in the program. Each iteration goes through all program points preceding an assignment statement, and for each such program point q all the variables in $\text{DEFS}(PTS, q, \text{STMT}(q))$ are processed. This set can contain at most all variables in the program. Thus, the first pass has time complexity $O(|VARS| * |ASS| * |VARS|)$. The second pass has time complexity $O(|ASS| + |VARS|)$, which is dominated by the first pass, so the complexity of SIMPLESLICE equals the complexity of the first pass. However, the algorithm will in most cases perform better (see Section 11).

In contrast to the PDG slicing algorithm, SIMPLESLICE must always slice w.r.t. *all* conditional statements in the code. The reason is that it does not attempt to follow control dependences when growing the sets N and P. Thus, if slicing w.r.t. a subset of the conditions, there is a risk that variables or assignments, indirectly affecting some selected condition through a control dependence, will not be included in these sets. When the slicing is done w.r.t. all conditions, then such variables or assignments will always affect at

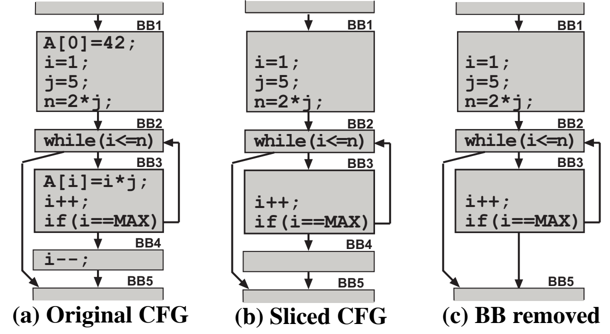


Figure 6. Removal of empty basic block

least one condition through a data dependence, and they will thus be recorded.

8. Flow Information and Code Removal

It should be noted that a sliced program only should be used in the flow analysis phase of a WCET analysis. A low-level analysis, which derives execution times for program code parts, must still analyse the non-reduced program. Consequently, to be valid, a flow analysis of a reduced program must produce flow information valid also for the original non-reduced program.

We take a general approach and consider the result of a flow analysis as upper and lower bounds on the number of times that different basic blocks or control-flow edges between basic block could be executed[1]. As an illustration consider the control-flow graph (CFG) in Figure 6(a), corresponding to the example code fragment in Figure 1(a). By giving a bound on the number of times node `BB2` can be executed (the loop header), we implicitly give a bound on the number of times the loop can be iterated.

In some cases, especially when slicing on individual loops or conditionals, we can remove all instructions in a basic block. We can then either keep the empty block, or we can remove the empty block and rewrite the CFG accordingly. For the latter approach we must be sure that we still produce flow information valid for the original non-sliced CFG.

As an illustration consider the CFG in Figure 6(b) corresponding to the sliced program in Figure 1(b). After the slicing `BB4` does not contain any instructions, and could therefore be removed, giving the simplified CFG in Figure 6(c).

A removal of node `BB4` does *not* mean that we can give flow information stating that node `BB4` never can be taken. Instead, we are only allowed to give bounds on basic blocks or edges which have not been removed. For example, a flow analysis for the reduced CFG in Figure 6(c) could produce a loop bound as an upper bound on `BB2`. This loop bound can be directly mapped back to the original CFG in Figure 6(a).

9. Identifying Inputs Affecting Program Flow

Another use of program slicing is to identify the input variables that may affect the program flow. This is useful information with a number of applications; e.g., to understand the behavior of the program or as an aid during testing. For WCET analysis, it is important to know these variables since their values in general must be constrained as much as possible, using information about the

[1] A basic block is a sequence of instructions that can be entered only at the first instruction in the sequence and exited only at the last instruction in the sequence [29].

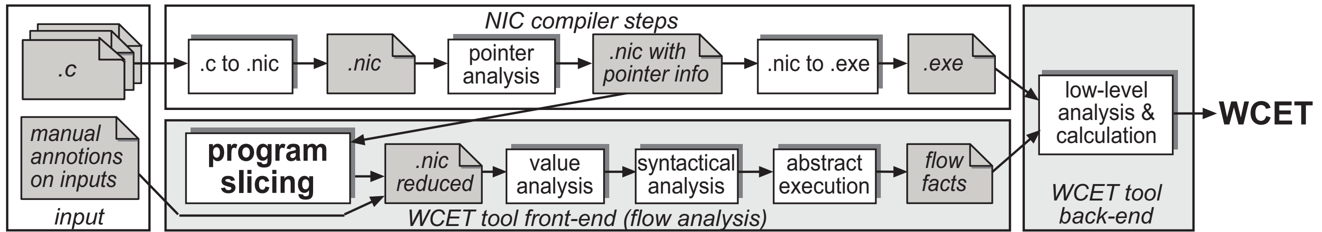


Figure 7. WCET analysis tool

environment of the program, in order to obtain tight program flow constraints from the flow analysis.

For an embedded program (or task) written in C or a similar language, the input variables may be

1. values read from the environment using primitives such as ports or memory mapped I/O,
2. parameters to `main()` or the particular function that invokes the task, and
3. data used for keeping the state of tasks between invocations or used for task communication, such as global variables or message queues.

The input variables that may affect the control flow are simply the remaining input variables, after slicing w.r.t. the conditions of the program, for which there is a path through the sliced program where the variable may be read before it is written. A simple data flow analysis can find for which variables such paths exist. If there are no such input variables in the reduced program, then the original program is a single-path program.

This analysis can also be made if we slice w.r.t. a subset of the conditions in the program, as described in Section 6. For example, if we slice w.r.t. a loop, then we can identify the inputs which control the behavior of that loop. If a loop is not controlled by any inputs, then it will always execute in the same way.

10. Our WCET Analysis Tool

The outlined program slicing method has been implemented as a step in our prototype WCET analysis research tool. The tool has a modular design, to ease replacement of analysis methods and target system platform [12].

The analysis steps of our WCET tool are depicted in Figure 7, highlighting the parts which are of particular interest for this paper. In essence, the tool architecture conforms to the general scheme for WCET analysis presented in Section 2, consisting of a flow analysis, a low-level analysis and a calculation.

The upper part of Figure 7 shows the conversion of the C code to intermediate code and executable code. We perform our analysis on NIC (New Intermediate Code), an intermediate code format designed for embedded system code analysis and compilation [33]. Performing our analyses on intermediate code allows us to easily identify variables, and other entities of interest, which are hard to identify directly from the object code. Furthermore, it gives us the opportunity to evaluate the benefits of integrating static WCET analysis with a compiler.

NIC is generated from C using a research compiler based on LCC [15]. The NIC format is able to handle complete ANSI-C. NIC can be closely connected to the C source, so that all data and control structures in the C code have direct counterparts in the NIC code. Alternatively, NIC can represent the code after various compiler optimizations, just before the compiler backend generates the resulting object code. The NIC control flow will then be close to the object code control flow, and derived flow facts can be directly mapped to the object code. Together with timing information from the low-level analysis, derived flow facts can be given as input to the calculation. A Steensgaard pointer analysis is performed on the generated NIC code. Finally, the NIC code is used to generate object code. More details on our use of NIC is given in [20].

The lower part of Figure 7 shows the steps of the flow analysis. These are described in more detail in Section 10.1. Our low-level analysis allows us to analyse and represent the effect of processor pipelining and instruction caches [10, 12], currently supporting the NECV850E and ARM9 processors. Our calculation supports three different calculation methods [8, 11, 36], each able to handle complex flow information and hardware timing dependendencies.

10.1 Flow analysis methods employed

The flow analysis phase of our WCET tool consists of several different analyses, as illustrated in Figure 7. The first analysis is the program slicing method outlined in this article, which takes the result of the NIC code with pointer information as input, and generates reduced NIC code as output to the subsequent analyses.

The second analysis, *value analysis*, is based on abstract interpretation [7] [20] and derives bounds on possible variable values (including pointers) for each node in a combined control-flow and call-graph. The abstract domain used in the current version of our WCET tool is a combination of the interval domain [7] and the congruence domain [17]. The analysis calculates an abstract state for each node in the graph, representing a set of states potentially possible at the particular node. We use *widening* to guarantee termination of the analysis, and *narrowing* to improve the precision [6].

The third analysis, *syntactical analysis*, uses pattern-matching to find common patterns in how control-flow constructs are written [21]. For example, the body of a `for(i=0;i<=100;i++)` loop construct can easily be deduced to iterate 101 times, using simple pattern matching, provided the body has no other loop exits. The syntactical analysis can be improved by using bounds on possible variable values derived by the value analysis. For example, if the value analysis derived the upper bound of `LIMIT` to be 50, the loop body of `for(i=0;i<=LIMIT;i++)` can easily be deduced to iterate at most 51 times. The syntactical analysis is currently under development and will therefore not be used, in Section 11, to evaluate our program slicing method.

The final flow analysis, labelled *abstract execution*, can be seen as a combination of abstract interpretation and symbolic execu-

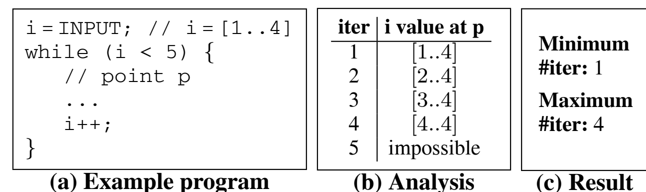


Figure 8. Example of abstract execution

tion [19, 20]. It uses abstract interpretation to derive safe bounds on variables at different points in the program. However, to derive loop bounds and other flow information, loops are "rolled out" dynamically and each iteration is analysed individually in a fashion similar to symbolic execution. The analysis also has some similarities with trace partitioning [4] and the simulation technique used in [26]. Compared to the later, however, our analysis uses a more detailed value domain.

Figure 8 gives a simple example on abstract execution using intervals. The loop in Figure 8(a) is analysed in Figure 8(b). As each iteration is analysed, the possible values of `i` are reduced until, finally, the value is impossible, and the loop analysis terminates. The result in Figure 8(c) shows the resulting lower and upper bound on the number of iterations.

By using abstract interpretation as a formal basis the abstract execution is guaranteed to analyse (i.e., abstractly execute) at least all feasible execution paths of the program. Hence, the analysis will have an execution time in most cases proportional to the worst-case execution time of the analyzed program. However, due to over-approximations in the analysis, the abstract states might sometimes include values which are not really possible in the actual execution. This means that non-feasible execution paths may be analysed by the abstract execution, sometimes leading to an overestimation of loop bounds and other flow information. This is the price we have to pay to have a safe and still very detailed flow analysis.

The result of the syntactical analysis and the abstract execution are passed as a set of *flow facts* [9] to the subsequent calculation phase, each giving constraints on the program flow for a certain piece of the analysed program (loop bounds, infeasible paths, execution dependences, etc.) [12].

11. Measurements and Evaluations

In order to evaluate the effectiveness and usefulness of our program slicing analysis we have performed a number of measurements using the benchmarks listed in Table 1. The benchmarks are taken from the Mälardalen University WCET benchmark suite [28], all being standard benchmarks for evaluating WCET tools and methods. The **#LOC** columns gives the number of lines in the C-code (including comments). The **#CFG**, **#BB**, **#Stmt**, **#Cnd**, **#LCnd**, **#LDcl**, **#GDcl** columns gives the number of control-flow graphs, basic blocks, statements, conditions (both for loops and branches), loop exit conditions, local variable declarations and global variable declarations in the NIC code, respectively.

11.1 Code size reduction

Table 2 shows the reduction in code size due to program slicing. The effect of using our alternative slicing algorithm, which does not distinguish between variable instances, is given in the **Simple slice** part. The results of slicing w.r.t. all conditionals (see Section 6.1) are given in the **Loops & conds** part. The results of slicing w.r.t. all loop exit conditions only (see Section 6.2) are given in the **Only loops** part.

The **-#BB**, **-#Stmt**, **-#Dcl**, and **-#GDcl** columns give the number of removed basic blocks, statements, local declarations, and global declarations, respectively. The **-%** columns gives the percentage of each removal, as compared with the original number. The bottom line gives the average for the programs.

We first notice that the number of removed code parts varies a lot between the analysed programs. This is natural, since the amount of code that can be removed depends on the program structure. For programs such as `bsort100` and `insertsort`, where most of the variables are used in some conditional expression, there are only a few statements which can be removed. For other programs, containing a lot of calculations where the results are not

Program	Original	Loops & conds		Only Loops	
	Time	Time	-%	Time	-%
`bsort100`	3.77	3.76	0	3.76	0
`cnt`	7.38	1.27	83	0.03	100
`cover`	4.54	1.13	75	0.01	100
`crc`	2.61	0.08	97	0.04	98
`edn`	23.4	0.11	100	0.11	100
`expint`	0.05	0.04	20	0.03	40
`fdct`	2.81	0.01	100	0.01	100
`fibcall`	0.02	0.02	0	0.03	0
`insertsort`	1.24	1.24	0	1.24	0
`jfdctint`	11.46	0.02	100	0.01	100
`lcdnum`	0.06	0.04	33	0.01	83
`statemate`	1.05	1.04	1	1.02	3
Average			**51**		**60**

Table 3. Time reduction for value analysis

used in conditional expressions, such as `crc` and `edn`, large code parts can be removed.

We see that the simple algorithm and slicing on all loops and conditionals perform similarly, while the more aggressive slicing on all loops only is able to reduce more code parts. Almost half of the statements are removed on average.

As explained in Section 9 we can use our program slicing to identify which input variables that may affect the program flow. In our benchmarks, the only inputs to the programs are the global variables. The **GDcl** column gives the number of removed global declarations. When there are no globals left in the program, it means that the program only has one execution path. For these type of programs it should be enough to run the program once to calculate the flow facts. Otherwise, the remaining global variables are candidates to be input variables, and bounds on their possible values they must therefore in general be provided in order to obtain a finite WCET estimate. More than half of the global variables are removed on average by the different slicings, which should reduce the effort to bound input variables considerably.

11.2 Flow analysis time reduction

To evaluate the effect of our slicing on subsequent flow analysis, we analysed both the original non-sliced programs and the sliced programs. We compared the execution times for the value analysis and the abstract execution (see Section 10.1), which both are time-consuming. For each benchmark we calculated flow facts in the form of loop bounds. The calculations yielded exactly the same flow facts for the non-sliced and sliced versions[2]. We have not included analysis times for the simple slicing algorithm, since it gives roughly the same times as the PDG-based method for loops and conditionals.

All measurements were performed on a 1.25 MHz PowerPC G4 processor, 1 Gb memory running Mac OS 10.4.4. We called the standard function `time()` immediately before and after the call to the particular flow analysis module, and thereafter calculated the time difference. The measured times in all tables are given in seconds.

Table 3 gives the time reduction for our value analysis. The column **Original** gives the analysis time for our original program, Table 4 gives the time reduction for our abstract execution. In the tables, the columns **Time** and **-%** give the analysis time and time savings for the sliced programs.

The gain in analysis time of a sliced program compared to an unmodified one, is approximately about 50% for the benchmarks. For some programs, (like `insertsort` and `statemate`) no real effect is visible, but for other programs (like `edn` and `fdct`), we

[2] Since only loop bounds were calculated, slicing on only loops is expected to give the same result as slicing on all conditions.

Program	Description	Code properties	#LOC	#CFG	#BB	#Stmt	#Cnd	#LCnd	#LDcl	#GDcl
bsort100	Bubblesort program.	Tests the basic loop constructs, integer comparisons, and simple array handling by sorting 100 integers.	128	2	13	29	5	4	10	1
cnt	Counts non-negative numbers in a matrix.	Nested loops, well-structured code.	267	6	22	70	5	4	20	6
cover	Program for testing many paths.	A loop containing many switch cases.	508	4	586	626	293	3	113	0
crc	Cyclic redundancy check computation on 40 bytes of data.	Complex loops, lots of decisions, loop bounds depend on function arguments, function that executes differently first time called.	128	3	29	89	9	3	37	5
edn	Finite Impulse Response (FIR) filter calculations.	A lot of vector multiplications and array handling.	285	9	46	286	12	12	141	3
expint	Series expansion for computing an exponential integral function	Inner loop that only runs once, structural WCET estimate gives heavy overestimate.	157	3	23	58	7	4	20	0
fdct	Fast Discrete Cosine Transform.	A lot of calculations based on array elements.	239	2	7	138	2	2	43	1
fibcall	Iterative Fibonacci, used to calculate fib(30).	Parameter-dependent function, single-nested loop.	72	2	7	22	2	2	8	0
insertsort	Insertion sort on a reversed array of size 10.	Input dependent nested loop with worst-case of $n^2/2$ iterations.	92	1	7	29	2	2	9	1
jfdctint	Discrete-cosine transformation on 8x8 pixel block.	Long calculation sequences, single-nested loops.	375	2	9	110	3	3	37	1
lcdnum	Read ten values, output half to LCD.	Loop with iteration-dependent flow.	64	2	56	67	26	1	14	2
statemate	Linear equations by LU decomposition.	Completely structured code. Tests effect of conditional flows.	1276	8	310	818	197	1	175	98

Table 1. Benchmark programs used in experiments

Program	Simple slice								Loops & conds								Only loops							
	-#BB	-%	-#Stmt	-%	-#LDcl	-%	-#GDcl	-%	-#BB	-%	-#Stmt	-%	-#LDcl	-%	-#GDcl	-%	-#BB	-%	-#Stmt	-%	-#LDcl	-%	-#GDcl	-%
bsort100	2	15	1	3	0	0	0	0	2	15	1	3	0	0	0	0	2	15	1	3	0	0	0	0
cnt	2	9	19	27	7	35	4	67	2	9	19	27	7	35	4	67	2	9	43	61	16	80	6	100
cover	424	72	213	34	10	9	0	-	424	72	213	34	10	9	0	-	570	97	313	50	110	97	0	-
crc	11	38	35	39	20	54	4	80	11	38	35	39	20	54	4	80	13	45	56	63	32	86	4	80
edn	0	0	202	71	118	84	3	100	0	0	202	71	118	84	3	100	0	0	202	71	118	84	3	100
expint	6	26	9	16	3	15	0	-	9	39	24	41	8	40	0	-	9	39	24	41	8	40	0	-
fdct	0	0	126	91	42	98	1	100	0	0	126	91	42	98	1	100	0	0	126	91	42	98	1	100
fibcall	0	0	8	36	4	50	0	-	0	0	8	36	4	50	0	-	0	0	8	36	4	50	0	-
insertsort	0	0	1	3	0	0	0	0	0	0	1	3	0	0	0	0	0	0	1	3	0	0	0	0
jfdcint	0	0	95	86	35	95	1	100	0	0	95	86	35	95	1	100	0	0	95	86	35	95	1	100
lcdnum	37	66	19	28	1	7	1	50	37	66	19	28	1	7	1	50	50	89	61	91	13	93	2	100
statemate	7	2	40	5	1	1	15	15	7	2	40	5	1	1	15	15	13	4	48	6	5	3	17	17
Average		**18**		**34**		**37**		**51**		**20**		**39**		**39**		**57**		**25**		**50**		**61**		**63**

Table 2. Program size reduction

Program	Original Time (s)	Loops & conds Time (s)	-%	Only Loops Time (s)	-%	Slicing Time (s)
bsort100	0.96	0.96	0	0.95	1	0
cnt	0.33	0.22	33	0.07	79	0
cover	0.69	0.60	13	0.06	91	0.26
crc	2.5	2.05	18	0.91	64	0.01
edn	7.48	1.25	83	1.22	84	0.05
expint	0.17	0.08	53	0.07	59	0
fdct	0.23	0.00	100	0.00	100	0.08
fibcall	0.03	0.02	33	0.02	33	0
insertsort	0.13	0.13	0	0.13	0	0.01
jfdctint	0.24	0.03	87	0.03	87	0.04
lcdnum	0.02	0.02	0	0.01	50	0
statemate	0.14	0.12	14	0.11	21	0.10
Average			**36**		**56**	

Table 4. Time reduction for abstract execution

see a dramatic reduction of analysis time. This has to do with the structure of the program, and the success of the slicing.

The slicing time is small compared to the time for the flow analysis. For some programs like `cover` and `statemate` we don't get any time reduction at all when considering also the slicing time. However, both of these programs has a structure that whould hardly occure in a program written by a human.

12. Conclusions and Future Work

In this paper we have shown that program slicing can be used to substantially reduce the execution time of WCET flow analysis methods. Since the method is general, and does not assume any specific program format, it can be used as a preceding stage to any flow analysis method.

Our measurements show that program slicing has a significant effect on the program size for many of our used benchmarks. On average, slicing gives a reduction of up to 45% in the number of statements and up to 58% in the number of program variables, depending on the slicing method chosen. For our flow analysis methods, this reduction in program size yields an average execution time reduction of up to 63% for value analysis, and up to 43% for abstract execution, while obtaining the same resulting flow information.

We have also shown how to slice w.r.t. a selected subset of all program conditions, and we have discussed the effects and usage of such slicings. A simplified slicing algorithm, that produces slices and reductions in flow analysis time comparable to that of the standard algorithm, has also been presented.

The benchmark programs that were used for this paper are single path programs. As a future work we plan to make studies on how slicing affects analysis times on real, industrial codes. In such a study we will also measure the effect that the loop only slicing will have on the precision of the estimated WCET. We also intend to study how the result of the slicing can be used to enhance the syntactical analysis mentioned in Section 10.1.

References

[1] Hiralal Agrawal. Dominators, Super Blocks, and Program Coverage. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–34, Portland, Oregon, 1994.

[2] ASTEC homepage, 2005. www.astec.uu.se.

[3] I. Bate and R. Reutemann. Worst-case execution time analysis for dynamic branch predictors. In *Proc. 16th Euromicro Conference of Real-Time Systems, (ECRTS'04)*, June 2004.

[4] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation; Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *LNCS*, pages 85–108. Springer-Verlag, 2002.

[5] Susanna Byhlin, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Applying static WCET analysis to automotive communication software. In *Proc. 17th Euromicro Conference of Real-Time Systems, (ECRTS'05)*, pages 249–258, July 2005.

[6] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Proc. 4th International Symposium on Programming Languages, Implementations, Logics, and Programs*, Lecture Notes in Computer Science (LNCS) 631, pages 269–295. Springer-Verlag, August 1992.

[7] 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.

[8] J. Engblom and A. Ermedahl. Pipeline Timing Analysis Using a Trace-Driven Simulator. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, Dec 1999.

[9] J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proc. 21th IEEE Real-Time Systems Symposium (RTSS'00)*, Nov 2000.

[10] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Box 337, Uppsala, Sweden, April 2002. ISBN 91-554-5228-0.

[11] A. Ermedahl, F. Stappert, and J. Engblom. Clustered Calculation of Worst-Case Execution Times. In *Proc. 6th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, (CASES'03)*, Oct 2003.

[12] 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.

[13] Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Experiences from industrial WCET analysis case studies. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis, (WCET'2005)*, pages 19–22, July 2005.

[14] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[15] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1990.

[16] J. Ganssle. Really real-time systems. In *Proc. Embedded Systems Conference San Francisco 2001*, April 2001.

[17] Philippe Granger. Static Analysis of Arithmetical Congruences. *International Journal of Computer Mathematics*, pages 165–199, 1989.

[18] Rajiv Gupta. Generalized Dominators and Post-Dominators. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 246–257, New York, NY, USA, 1992. ACM Press.

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

[20] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Towards a flow analysis for embedded system C programs. In *Proc. 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005)*, February 2005.

[21] Jan Gustafsson, Björn Lisper, Christer Sandberg, and Nerina Bermudo. A tool for automatic flow analysis of C-programs for WCET calculation. In *Proc. 8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, January 2003.

[22] C. Healy, Mikael Sjödin, V. Rustagi, and David Whalley. Bounding Loop Iterations for Timing Analysis. In *Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, June 1998.

[23] C. Healy and D. Whalley. Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*, June 1999.

[24] N. Holsti, T. Långbacka, and S. Saarinen. Worst-Case Execution-Time Analysis for Digital Signal Processors. In *Proc. EUSIPCO 2000 Conference (X European Signal Processing Conference)*, 2000.

[25] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[26] T. Lundqvist and P. Stenström. Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, June 1998.

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

[28] Mälardalen University WCET project homepage, 2006. www.mrtc.mdh.se/projects/wcet.

[29] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997. ISBN: 1-55860-320-4.

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

[31] Karl J. Ottenstein and Linda M. Ottenstein. The Program Dependence Graph in a Software Development Environment. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, 1984.

[32] M. Rodriguez, N. Silva, J. Esteves, L. Henriques, D. Costa, N. Holsti, and K. Hjortnaes. Challenges in Calculating the WCET of a Complex On-board Satellite Application. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, 2003.

[33] J. Runeson and S. Nyström. Retargetable Graph-coloring Register Allocation for Irregular Architectures. In *Proc. 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES 03)*, 2003.

[34] C. Sandberg. Inspection of Industrial Code for Syntactical Loop Analysis. In *Proc. 4th International Workshop on Worst-Case Execution Time Analysis, (WCET'2004)*, June 2004.

[35] 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.

[36] 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.

[37] B. Steensgaard. Points-to Analysis in Almost Linear Time. *Proc. in*

23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Jan 1996.

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

[39] Tidorum. Bound-T tool homepage, 2006. `www.tidorum.fi/bound-t/`.

[40] Vinnova homepage, 2006. `www.vinnova.se`.

[41] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[42] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A Brief Survey of Program Slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.