Abstraction of Assembler Programs for Symbolic Worst Case Execution Time Analysis

Tobias Schuele
Tobias.Schuele@informatik.uni-kl.de

Klaus Schneider
Klaus.Schneider@informatik.uni-kl.de

Reactive Systems Group, Department of Computer Science, University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany

ABSTRACT

Various techniques have been proposed to determine the worst case execution time of real–time systems. For most of these approaches, it is not necessary to capture the complete semantics of the system. Instead, it suffices to analyze an abstract model provided that it reflects the system's execution time correctly. To this end, we present an abstraction technique based on program slicing that can be used to simplify software systems at the level of assembler programs. The key idea is to determine a minimal set of instructions such that the control flow of the program is maintained. This abstraction is essential for reducing the runtime of the analysis algorithms, in particular, when symbolic methods are used to perform a complete state space exploration.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; C.4 [**Performance of Systems**]: Measurement techniques, modeling techniques

General Terms

Performance, Algorithms, Verification

Keywords

Real–time systems, worst case execution time, abstraction, program slicing, assembler programs, symbolic simulation

1. INTRODUCTION

The tight analysis of reaction times is a major criterion in the design of embedded systems. In particular, safety critical applications often require that certain tasks must be completed before a strict deadline. To this end, a variety of methods to estimate the *worst case execution time (WCET)* [23] of real-time systems have been proposed. State of the art approaches use different techniques like abstract interpretation [7, 8, 26], symbolic execution [14, 16], special restrictions on loops [9], computer algebra [1], and integer linear programming [11]. There are even frameworks that take into account pipeline effects and cache behavior [8, 12, 13, 20, 26].

Usually, one distinguishes between two approaches: the *high–level* and the *low–level* WCET analysis [6]. High–level analysis is applied to an architecture independent description of the system to obtain approximate timing estimates in early design phases. For example, in hardware/software codesign, WCET analysis provides the designer with information to decide which parts of the system should be implemented in hardware. At this stage, the primary goal is to quickly get a rough estimate of the worst case execution time without particular demands on accuracy and tightness.

In contrast, low–level analysis is performed in late design phases. Thus, it depends on the hardware/software partitioning and on the chosen architecture. At this stage, tight and safe estimates are mandatory to ensure that all requirements on the timing are met. For this reason, WCET analysis must be performed at least at the level of assembler programs. Assembler programs do not only take into account the instruction set architecture of the target, but also provide a unifying framework for different design flows which is independent of high–level languages and tools.

The major problem of a tight WCET analysis is that the maximal number of computation steps, in particular, the number of iterations of loops, may heavily depend on the input data. For this reason, most approaches require that the programmer specifies an upper bound on the number of loop iterations. For nontrivial programs, this is an error-prone task and can lead to rather pessimistic WCET estimates [6].

To solve these problems, symbolic methods have been proposed that compute tight bounds on the execution time [15, 25]. In contrast to [1, 14, 16], these methods perform a complete state space exploration using implicit set representations. However, symbolic methods can be very time consuming since they explore all possible computations.

In this paper, we present a novel technique to speed up symbolic WCET analysis. Using this technique, it is possible to derive an abstract model from a given program that can be analyzed much more efficiently. The idea behind our approach is that we are not interested in the complete semantics of a program, but merely in its worst case execution time. In other words, we can eliminate instructions that do not affect the program's execution time.

In general, the elimination of parts of a program is called *program slicing* [27]. Using program slicing, we can determine those instructions that do not affect the control flow of the program. For that purpose, we introduce the notion of relevant registers. A register is called relevant if it eventually affects a branch. In this way, we are able to determine a minimal set of instructions that are necessary to maintain the control flow of a program.

Our approach offers the following advantages:

- Fully automatic, i.e., no manual interaction required.
- Independence of the design flow by considering assembler programs.
- Accuracy (the number of executed instructions is guaranteed to be the same for the original and the abstracted program).
- Easily adaptable to different architectures by using generic frameworks for representing and analyzing programs.

On the one hand, our abstraction technique is related to code optimization techniques used in compilers [19]. On the other hand, there are some fundamental differences: Many methods such as dead–code elimination aim at improving the execution time of a program without changing its semantics. In contrast, our method *must not* change the execution time, but it may alter the semantics.

In the next section, we sketch the symbolic approach to WCET analysis. Then, we describe our abstraction technique (Section 3) and present experimental results (Section 4). We conclude with a summary and directions for future work (Section 5).

2. PRELIMINARIES

In this section, we briefly describe how the worst case execution time of assembler programs can be determined by means of symbolic state space exploration. We start with the translation of assembler programs to transition systems which is important for our abstraction procedure. For more detailed information, the reader is referred to [25].

Transition systems are a framework for modeling finite and infinite state processes. In general, a transition system consists of a set of states and a transition relation. The states represent the configurations of a process and the transition relation describes possible computations. In timed transition systems [15], each transition is labeled with a weight that represents the time units required to take the transition.

Given an assembler program, the states can be interpreted as variable assignments that represent the current values of the processor's registers. As the program proceeds with its execution, it changes some of the register contents and therefore, we have a new assignment at the next point of time. Hence, we can represent the transition relation of a program by a formula over a set of variables $\mathcal{V} \cup \mathcal{V}'$ where $\mathcal{V}$ and $\mathcal{V}'$ are the register contents of the current and the next state, respectively.

Assume that for each register we have an associated pair of variables r and r' that denote the current and the next register value, respectively. In particular, let pc and pc' be two variables that represent the program counter. Moreover, let $P = I_1, \ldots, I_n$ be the program to be modeled such that instruction I_i is located at memory address $\mathsf{A}(I_i)$. Then, a symbolic description of the transition relation $\mathcal{R}$ can be obtained as follows where $\mathsf{S}(I_i)$ defines the semantics of an instruction I_i:

$$\mathcal{R} :\equiv \bigvee_{i=1}^{n} ((pc = \mathsf{A}(I_i)) \wedge \mathsf{S}(I_i))$$

In order to execute a program symbolically, we have to compute the successors for a set of states. As usual in symbolic methods, this can be performed by intersection, existential quantification, and variable substitution [5]. Since all inputs are considered at once during image computation, we can traverse the state space in a breadth first manner where at each step all successor states are explored simultaneously.

Besides the transition relation, we also need a set of initial states $\mathcal{I}$ and a set of final states $\mathcal{F}$. The initial states specify the first instruction to be executed and the initial register contents where preconditions can be used to restrict the set of possible inputs. Final states are those states where the program counter points to the last instruction (e.g., *return from subroutine*).

The worst case execution time of a program is given by the longest path between $\mathcal{I}$ and $\mathcal{F}$. For ordinary transition systems, the length of a path is simply the number of transitions. For timed transition systems, the length of a path is the sum of its weights. Note that the length of a path is only well-defined if the transition relation is acyclic. Otherwise, there exists at least one infinite path which means that the program does not terminate for all input values. In this case, the worst case execution time is infinite.

In the sequel, we consider the MIPS32 instruction set architecture which defines a well-structured and powerful instruction set [18]. Table 1 shows the semantics of some selected instructions.[1] The function C ensures that registers, which are not affected by an instruction, remain unchanged:

$$\mathsf{C}(R) :\equiv \bigwedge_{r \in W} (r' = r) \text{ with } W = \mathcal{V} \backslash (R \cup \{pc\})$$

As an example, Figure 1 shows the translation of a MIPS assembler program to a transition system where `$i` denotes register r_i.

while $y > 0$ **do** $t := x \oplus y;$ $y := 2 \times (x \wedge y);$ $x := t;$ **end**	I_1 : `AND $2, $3, $4` I_2 : `XOR $3, $3, $4` I_3 : `SLL $4, $2, 1` I_4 : `BNE $4, $0, -3`

$$\begin{aligned}
\mathcal{R} &:\equiv \textstyle\bigvee_{i=1}^{4} ((pc = \mathsf{A}(I_i)) \wedge \mathsf{S}(I_i)) \\
\mathcal{I} &:\equiv (pc = \mathsf{A}(I_1)) \wedge (0 \leq r_3 < 2^{32}) \wedge (0 \leq r_4 < 2^{32}) \\
\mathcal{F} &:\equiv (pc > \mathsf{A}(I_4))
\end{aligned}$$

$$\begin{aligned}
\mathsf{S}(I_1) &= (r_2' = r_3 \wedge r_4) \wedge (pc' = pc + 4) \wedge \mathsf{C}(\{r_2\}) \\
\mathsf{S}(I_2) &= (r_3' = r_3 \oplus r_4) \wedge (pc' = pc + 4) \wedge \mathsf{C}(\{r_3\}) \\
\mathsf{S}(I_3) &= (r_4' = r_2 \times 2) \wedge (pc' = pc + 4) \wedge \mathsf{C}(\{r_4\}) \\
\mathsf{S}(I_4) &= ((r_4 \neq 0) \Rightarrow (pc' = pc + 4 \times (-3)) \mid \\
&\qquad (pc' = pc + 4)) \wedge \mathsf{C}(\emptyset)
\end{aligned}$$

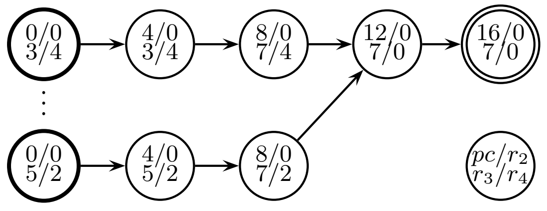


Figure 1: Sample program and transition system

In practice, we need a suitable logic as the basic formalism for representing transition systems by means of the above equations. In particular, the logic should be decidable to obtain automatic tools. In [25] we used Presburger arithmetic [22] as the base logic to capture the semantics of assembler programs at a high level of abstraction. Presburger arithmetic is a decidable subset of the theory of natural numbers where multiplication is not allowed as a single operation. However, multiplication can be accomplished iteratively by shifting and addition.

[1]Conditional expressions are denoted by $x \Rightarrow y \mid z$.

Table 1: Semantics of selected MIPS32 instructions

Instruction (I)		Description	Semantics ($\mathsf{S}(I)$)
LW	rt, c(rs)	load word	$(r'_t = M_{r_s+c}) \wedge (pc' = pc + 4) \wedge \mathsf{C}(\{r_t\})$
SW	rt, c(rs)	store word	$(M'_{r_s+c} = r_t) \wedge (pc' = pc + 4) \wedge \mathsf{C}(\emptyset)$
ADDI	rt, rs, c	add immediate	$(r'_t = r_s + c) \wedge (pc' = pc + 4) \wedge \mathsf{C}(\{r_t\})$
SUB	rd, rs, rt	subtract	$(r'_d = r_s - r_t) \wedge (pc' = pc + 4) \wedge \mathsf{C}(\{r_d\})$
SLT	rd, rs, rt	set on less than	$((r_s < r_t) \Rightarrow (r'_d = 1) \mid (r'_d = 0)) \wedge (pc' = pc + 4) \wedge \mathsf{C}(\{r_d\})$
AND	rd, rs, rt	and	$(r'_d = r_s \wedge r_t) \wedge (pc' = pc + 4) \wedge \mathsf{C}(\{r_d\})$
SLL	rd, rt, sa	shift left logical	$(r'_d = r_t \times 2^{sa}) \wedge (pc' = pc + 4) \wedge \mathsf{C}(\{r_d\})$
BEQ	rs, rt, c	branch on equal	$((r_s = r_t) \Rightarrow (pc' = pc + 4 \times c) \mid (pc' = pc + 4)) \wedge \mathsf{C}(\emptyset)$
NOP		no operation	$(pc' = pc + 4) \wedge \mathsf{C}(\emptyset)$

In contrast to the original definition, we interpret the logic over the integers rather than over the natural numbers. Furthermore, we employ logical operations on numbers since these are common to all instruction set architectures and frequently used in assembler programs. From a theoretical point of view, one can model every program as a transition system using Presburger arithmetic, since one can simulate register machines [17] in this way. From a practical point of view, the formalization of most instruction sets is straightforward using Presburger arithmetic.

It is well-known that every Presburger formula can be translated to a finite automaton that encodes its models [2, 4, 28]. Finite automata are an efficient data structure for storing and manipulating large sets during symbolic simulation. As there exists for every finite automaton an equivalent minimal one, automata can be used as canonical representations for Presburger formulas. This is analogous to the use of binary decision diagrams [3] as canonical normal forms for propositional logic.

3. ABSTRACTION

Using the techniques described in the previous section, we are able to compute tight bounds on the worst case execution time of assembler programs. However, this can be very time consuming and therefore, we present an abstraction technique that can be used to simplify the programs prior to WCET analysis. As mentioned previously, the key idea of our method is to identify and eliminate those instructions that do not affect the control flow of a program.

Consider for example the program shown in Figure 2 which multiplies two natural numbers x and y. Since the control flow, i.e., the loop condition and the if–statement, only depends on the value of y, we do not need to compute the product p in order to determine its execution time. As a consequence, computing the new value for x at each iteration is also dispensable.

```
$p := 0$;
**while** $y \neq 0$ **do**
    **if** $y$ mod $2 = 1$ **then**
        $p = p + x$;
    **end**;
    $x := x \cdot 2$;
    $y := y \,/\, 2$;
**end**;
```

Figure 2: Multiplication of natural numbers

Let us now consider the assembly code for the above algorithm (Figure 3). For the moment, we assume without loss of generality that all instructions are executed in a single cycle. Then, the instructions that compute p and x can be replaced with NOPs without affecting the execution time. In Section 3.3, we will present a more general technique that does not replace redundant instructions with NOPs, but simplifies their semantics such that the execution time of an instruction is retained. In this way, we are able to model multi-cycle instructions and to take into account pipeline hazards.

```
     MOVE  $3, $0      // p := 0
L1:  BNE   $2, $0, L2  // y != 0 => L2
     J     $31         // return
L2:  ANDI  $4, $2, 1   // r4 := y mod 2
     BEQ   $4, $0, L3  // r4 = 0 => L3
     ADDU  $3, $3, $1  // p := p + x
L3:  SLL   $1, $1, 1   // x := x * 2
     SRL   $2, $2, 1   // y := y / 2
     J     L1          // => L1
```

Figure 3: Assembler program for Figure 2

It is important to note that the size of a program's transition relation and the state sets do not primarily depend on the number of instructions, but on the amount of information they encode. In particular, symbolic WCET analysis can be performed much more efficiently if the transition relation contains only necessary information.

3.1 Program Representation

Before we describe our abstraction algorithm in detail, we need the notion of Kripke structures to model the control flow of assembler programs.

DEFINITION 1 (KRIPKE STRUCTURE). *Let V be a finite set of variables. A Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{R}, \mathcal{L})$ is a transition system where $\mathcal{S}$ is the set of states, $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation, and $\mathcal{L} : \mathcal{S} \to 2^V$ is a labeling function that maps each state to a set of variables.*

Given a program $P = I_1, \ldots, I_n$, we construct a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{R}, \mathcal{L})$ such that each state represents exactly one instruction, i.e., $S = \{s_1, \ldots s_n\}$. The transition relation[2] is defined as follows:

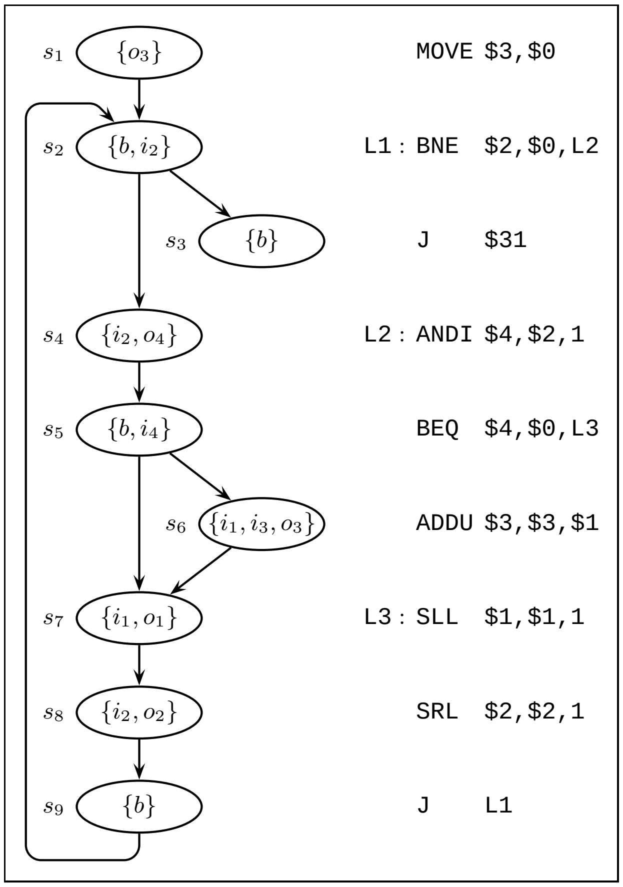
- $(s_i, s_{i+1}) \in \mathcal{R} \Leftrightarrow I_i$ is not a jump
- $(s_i, s_j) \in \mathcal{R} \Leftrightarrow I_i$ is a branch or a jump and I_j is the target instruction

[2]Note that a transition system as described in Section 2 represents the complete semantics of a program. In contrast, a Kripke structure as defined above only represents those parts of a program that are relevant for the abstraction algorithm.

Let $V = \{b, i_1, \ldots, i_{31}, o_1, \ldots, o_{31}\}$ be the set of variables, then we have for the labeling function:

- $b \in \mathcal{L}(s_i) \Leftrightarrow I_i$ is a branch or a jump
- $i_k \in \mathcal{L}(s_i) \Leftrightarrow I_i$ reads register r_k (input)
- $o_k \in \mathcal{L}(s_i) \Leftrightarrow I_i$ writes register r_k (output)

Figure 4 shows the Kripke structure for our sample program. In the MIPS instruction set architecture, register r_0 is always zero and can hence be treated as a constant [18].



s_1 $\{o_3\}$ MOVE $3,$0
s_2 $\{b, i_2\}$ L1 : BNE $2,$0,L2
s_3 $\{b\}$ J $31
s_4 $\{i_2, o_4\}$ L2 : ANDI $4,$2,1
s_5 $\{b, i_4\}$ BEQ $4,$0,L3
s_6 $\{i_1, i_3, o_3\}$ ADDU $3,$3,$1
s_7 $\{i_1, o_1\}$ L3 : SLL $1,$1,1
s_8 $\{i_2, o_2\}$ SRL $2,$2,1
s_9 $\{b\}$ J L1

Figure 4: Kripke structure for Figure 3

3.2 Computation of Relevant Registers

As the next step, we have to identify those computations that maintain the control flow of a program. To this end, we determine for each register a set of instructions for which the register is *relevant*. A register r_k is relevant for an instruction I_i if

(1) r_k is an operand of I_i and I_i is a branch,
(2) r_k is an operand of I_i and the results of I_i are relevant for one of its successors,
(3) r_k is not written by I_i but it is relevant for one of its successors.

For example, register r_4 is relevant for instruction I_5 since r_4 is an operand of I_5 and I_5 is a branch, i.e., the outcome of the branch depends on the content of r_4.

In the following, we use the μ–calculus [10, 21] to compute the set of registers that are relevant for an instruction. Given a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{R}, \mathcal{L})$, a variable v with $v \in V$ holds in a state $s \in S$ iff $v \in \mathcal{L}(s)$. This is naturally extended to the Boolean operators. The formula $\Diamond\varphi$ holds in a state $s \in S$ iff there exists an $s' \in S$ such that $(s, s') \in \mathcal{R}$ and φ holds in s'. The set of states where a formula φ holds is usually denoted as $[\![\varphi]\!]_{\mathcal{K}}$.

The rules (1) – (3) can then be expressed by a system of equations $R_1, \ldots, R_{31}$ where R_k represents the set of instructions for which register r_k is relevant:

$$\begin{aligned} R_1 &\overset{\mu}{=} i_1 \wedge (b \vee \textstyle\bigvee_{i=1}^{31}(o_i \wedge \Diamond R_i)) \vee (\neg o_1 \wedge \Diamond R_1) \\ &\vdots \\ R_{31} &\overset{\mu}{=} i_{31} \wedge (b \vee \textstyle\bigvee_{i=1}^{31}(o_i \wedge \Diamond R_i)) \vee (\neg o_{31} \wedge \Diamond R_{31}) \end{aligned}$$

Since we are interested in an optimal solution, we compute the least solutions for R_k. Thus, $[\![R_k]\!]_{\mathcal{K}}$ denotes the minimal set of states (instructions) for which register r_k is relevant. The solution of the equation system can be computed using Tarski's fixpoint theorem in time $\mathcal{O}(n^2)$ where n is the number of instructions [24].

More precisely, for each equation we have a set that contains the current solution. At the beginning, all the sets are empty. During the fixpoint iteration the right-hand sides of the equations are evaluated and the new solutions are stored in the corresponding sets. The evaluation of a $\Diamond$ operator is simply accomplished by determining the set of predecessor states.

PROPOSITION 1. *The fixpoint iteration for computing relevant registers by means of the given equation system terminates.*

Proof According to Tarski's fixpoint theorem, it suffices to show that the right-hand sides of the equations are monotonic. Since none of the R_i's occurs complemented in any of the equations, monotonicity is guaranteed. $\square$

For the program of Figure 3, we obtain the following solution:

$$\begin{aligned} [\![R_1]\!]_{\mathcal{K}} &= \emptyset \\ [\![R_2]\!]_{\mathcal{K}} &= \{s_1, s_2, s_4, s_5, s_6, s_7, s_8, s_9\} \\ [\![R_3]\!]_{\mathcal{K}} &= \emptyset \\ [\![R_4]\!]_{\mathcal{K}} &= \{s_5\} \\ [\![R_5]\!]_{\mathcal{K}} &= \emptyset \\ &\vdots \\ [\![R_{31}]\!]_{\mathcal{K}} &= \emptyset \end{aligned}$$

3.3 Elimination of Redundant Instructions

Now we are able to identify redundant instructions and to replace them with NOPs. More precisely, an instruction I_i can be replaced with a NOP iff I_i is not a branch (jump) and its result is not relevant to any successor instructions:

$$N = \neg b \wedge \neg \textstyle\bigvee_{i=1}^{31}(o_i \wedge \Diamond R_i)$$

Figure 5 shows the simplified assembler program for the algorithm of Figure 2 with $N = \{s_1, s_6, s_7\}$.

The above techniques aim at reducing the complexity of a program's transition relation to speed up symbolic WCET analysis. However, replacing redundant instructions with NOPs is not an optimal solution for that purpose. This is due to the fact that NOPs apply the identity function to each register which is superfluous for irrelevant registers.

Moreover, replacing an instruction with a NOP is only possible if we presuppose that all instructions are executed in the same amount of time. As mentioned previously, a more general solution is to modify the instruction semantics, such that only relevant registers are considered.

Table 2: Results and runtimes of WCET analysis

Benchmark	8 Bits			16 Bits			24 Bits		
	WCET	Time [s]		WCET	Time [s]		WCET	Time [s]	
		w/o abs.	with abs.		w/o abs.	with abs.		w/o abs.	with abs.
SquareRoot	97	0.25	0.23	1537	16.24	15.77	24577	826.04	806.57
RussMult	60	> 1h	0.19	116	> 1h	1.70	172	> 1h	7.80
Booth	84	> 1h	0.58	160	> 1h	6.06	236	> 1h	30.12
Fibonacci	1530	1115.59	2.88	393210	> 1h	3331.45	–	> 1h	> 1h
DCT	716	–	0.36	716	–	0.36	716	–	0.36
MatrixMult	402755	–	265.61	402755	–	265.61	402755	–	265.61

```
      NOP
L1 :  BNE   $2, $0, L2  // y != 0 => L2
      J     $31         // return
L2 :  ANDI  $4, $2, 1   // r4 := y mod 2
      BEQ   $4, $0, L3  // r4 = 0 => L3
      NOP
L3 :  NOP
      SRL   $2, $2, 1   // y := y / 2
      J     L1          // => L1
```

Figure 5: Assembler program after abstraction

Let X_i and X'_i be the set of irrelevant registers for instruction I_i and its successors, respectively, with

$$X_i = \{r_k \mid s_i \notin [\![R_k]\!]_{\mathcal{K}}\}$$
$$X'_i = \{r'_k \mid \neg\exists s_j \in \mathcal{S} \,.\, (s_i, s_j) \in \mathcal{R} \wedge s_j \in [\![R_k]\!]_{\mathcal{K}}\}.$$

Then, the simplified transition relation is obtained as follows (cf. Section 2):

$$\mathcal{R} :\equiv \bigvee_{i=1}^{n} ((pc = \mathsf{A}(I_i)) \wedge (\exists X_i, X'_i \,.\, \mathsf{S}(I_i)))$$

Existentially quantifying over the variables in X_i and X'_i means that *irrelevant registers can change their contents arbitrarily* during a transition. As a result, the transition relation is simplified even more than with NOPs. In addition, this approach is also applicable to timed transition systems [15], since it does not change the transition weights that represent the time units required to take a transition.

For example, consider the fourth instruction of the program shown in Figure 3:

I_4 : `ANDI $4, $2, 1`

Since r_2 is the only relevant register for I_4 (s_4 is only contained in R_2), we have $X_4 = \{r_1, r_3, \dots, r_{31}\}$. For instruction I_5, the successor of I_4, the registers r_2 and r_4 are relevant and hence, we get $X'_4 = \{r'_1, r'_3, r'_5, \dots, r'_{31}\}$. Thus, we have for the corresponding part of the transition relation:

$$(pc = \mathsf{A}(I_4)) \wedge (\exists r_1, r_3, \dots, r_{31}, r'_1, r'_3, r'_5, \dots, r'_{31} \,.\, \mathsf{S}(I_4))$$

4. EXPERIMENTAL RESULTS

To evaluate our approach, we implemented the algorithms and applied them to some benchmarks. All experiments were performed on an AMD Opteron processor with 2 GHz clock frequency. The benchmarks were written in C and compiled[3] with the GNU C-compiler to obtain assembly code for the MIPS R3000 family.

[3]Code optimization was enabled using option -O2.

Table 2 shows the results for different inputs given as preconditions of the form $0 \leq x < 2^n$ and $-2^{n/2} \leq x < 2^{n/2}$, respectively. The worst case execution times are given as number of executed instructions. For all the benchmarks, abstraction was performed in less than one second which is negligible in most cases compared to the time for WCET analysis.

For the first benchmark program (square root), abstraction did not yield a significant improvement. This is not a problem, since it can be analyzed in acceptable time without abstraction. For the RussMult and the Booth benchmark, the situation is different. For these programs, WCET analysis could not be performed in less than one hour without abstraction. Using our abstraction technique, the runtimes are reduced to less than one minute.

The runtimes for the Fibonacci benchmark could be reduced from 1300 to 6 seconds for 8 bit wide numbers. Nevertheless, it is hardly tractable for larger inputs since its execution time is exponential in the input size.

The execution times of the last two benchmarks are independent of the input data, and hence, the results are the same for different bitwidths. For these programs abstraction is essential to obtain tight bounds, since our algorithms do not support multiplication as a single operation. This is due to the fact that Presburger arithmetic would otherwise be undecidable. However, using abstraction the multiplication operations were eliminated and the programs could be analyzed in less than one and twelve minutes, respectively.

Note that multiplication operations can be replaced with successive shift operations and additions. In this case, WCET analysis is possible without abstraction. However, this increases the number of executed instructions and thus requires timed transition systems to avoid overestimation.

5. SUMMARY AND CONCLUSION

We presented an abstraction technique based on program slicing that can be used to simplify assembler programs prior to WCET analysis. Abstraction is important for symbolic methods that perform a complete state space exploration to obtain tight bounds on the worst case execution time. Tight estimates are essential for core routines which are frequently called and contribute to a large part to the total execution time, e.g., interrupt service routines of real–time operating systems and inner loops.

Our approach is based on the observation that the high complexity of symbolic methods is largely due to computations that are irrelevant with respect to a program's control flow. Such computations can be eliminated without affecting the number of executed instructions. To this end, a minimal set of instructions is determined that are necessary to maintain the control flow. Experimental results show that abstraction significantly reduces the runtime of WCET analysis. For some programs, abstraction is mandatory to prevent an exponential blow–up during symbolic simulation.

As a major advantage, our method is fully automatic and does not require manual interaction or annotation of the programs to be analyzed. Moreover, it is independent of the design flow by considering assembler programs and it is easily adaptable to various instruction set architectures. However, it does not aim at determining cycle accurate timing estimates. In particular, we do not yet consider superscalar execution and cache memories. Since the latter can have significant impact on the worst case execution time, various approaches have been proposed to incorporate cache behavior into WCET analysis [8, 12, 20]. We believe that the best results are achieved by a combination of different methods. To this end, our approach can be extended such that not only the control flow of a program, but also its memory behavior is preserved. In other words, the abstraction rules can be modified such that load and store instructions are not eliminated.

As another approach, one could use our method to determine the number of times a loop is executed. Recall, that symbolic methods yield tight estimates even if the number of iterations depends on the input data. Then, cycle accurate techniques for WCET analysis can be used to estimate the execution time of the loop body. In this way, it is possible to obtain precise bounds on the worst case execution time of data dependent loops.

6. REFERENCES

[1] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *Workshop on Real-Time Programming*, Palma, Spain, 2000.

[2] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Trees in Algebra and Programming (CAAP)*, volume 1059 of *LNCS*, pages 30–43, Linköping, Sweden, 1996. Springer.

[3] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] J. Büchi. On a decision method in restricted second order arithmetic. In E. Nagel, editor, *International Congress on Logic, Methodology and Philosophy of Science*, pages 1–12, Stanford, CA, 1960. Stanford University Press.

[5] O. Coudert, C. Berthet, and J. Madre. Verification of synchronous sequential machines using symbolic execution. In *Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 365–373, Grenoble, France, June 1989. Springer.

[6] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Towards industry strength worst-case execution time analysis. ASTEC Technical Report 99/02, Uppsala University, Sweden, 1999.

[7] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *International European Conference on Parallel Processing (EuroPar)*, volume 1300 of *LNCS*, pages 1298–1307, Passau, Germany, 1997. Springer.

[8] C. Ferdinand and R. Wilhelm. Fast and efficient cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2/3), 1999.

[9] C. Healy, R. van Engelen, and D. Whalley. A general approach for the tight timing predictions of non-rectangular loops. In *Real-Time Technology and Applications Symposium*, 1999.

[10] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, December 1983.

[11] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Design Automation Conference (DAC)*, pages 456–461, 1995.

[12] Y.-T. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer, 1999.

[13] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, 1995.

[14] Y. Liu and G. Gomez. Automatic accurate time-bound analysis for high-level languages. In *Languages, Compilers and Tools for Embedded Systems (LCTES)*, 1998.

[15] G. Logothetis and K. Schneider. Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In *Design, Automation and Test in Europe (DATE)*, pages 196–203, Munich, Germany, March 2003. IEEE Computer Society.

[16] T. Lundqvist and P. Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In *Languages, Compilers and Tools for Embedded Systems (LCTES)*, 1998.

[17] M. Minsky. *Computation: Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, 1967.

[18] MIPS Technologies. Website, 2003. http://www.mips.com.

[19] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1997.

[20] F. Mueller, D. Whalley, and M. G. Harmon. Predicting instruction cache behavior. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1994.

[21] V. Pratt. A decidable μ-calculus. In *Symposium on Foundations of Computer Science (FOCS)*, pages 421–427, New York, 1981. IEEE Computer Society.

[22] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In F. Leja, editor, *Sprawozdanie z I Kongresu Matematyków Krajów Słowiańskich*, pages 92–101, Warszawa, Skład Głowny, 1929.

[23] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(1):159–176, 1989.

[24] K. Schneider. *Verification of Reactive Systems – Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[25] T. Schüle and K. Schneider. Exact runtime analysis using automata-based symbolic simulation. In *Formal Methods and Models for Codesign (MEMOCODE)*, Mont Saint-Michel, France, 2003. IEEE Computer Society.

[26] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separate cache and path analyses. *Real-Time Systems*, 18(2/3), May 2000.

[27] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[28] P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *LNCS*, pages 1–19, Berlin, March 2000. Springer.