Exact Runtime Analysis Using Automata–Based Symbolic Simulation

Tobias Schüle and Klaus Schneider
University of Kaiserslautern, Department of Computer Science, Reactive Systems Group
P. O. Box 3049, 67653 Kaiserslautern, Germany
{Tobias.Schuele,Klaus.Schneider}@informatik.uni-kl.de
http://ag-rs-www.informatik.uni-kl.de

Abstract

In this paper, we present a technique for determining tight bounds on the execution time of assembler programs. Thus, our method is independent of the design flow, but takes into account the target architecture to obtain accurate estimates. The key idea is to compute the maximal number of executed instructions by means of symbolic simulation. To this end, we utilize a slight extension of Presburger arithmetic that can be translated to finite automata. Finite automata are an efficient data structure for symbolically traversing the state space of a program.

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] have been proposed. Usually, one distinguishes between two approaches: the high–level and the low–level WCET analysis [9].

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 co–design WCET analysis helps the designer 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. High–level analysis can be performed by determining path information [22] such as unfeasible computations or bounds on the maximal number of loop iterations. Clearly, high–level WCET analysis is undecidable when infinite data types are used, and therefore only limited automation can be achieved. State of the art approaches use abstract interpretation [10], symbolic execution [17, 15], special restrictions on loops [11], and computer algebra [1].

In contrast, low–level analysis is performed in late design phases. Thus, it depends on the hardware/software partitioning as well as on the chosen architecture (microcontroller). At this stage, tight and safe estimates are mandatory to ensure that all requirements on the timing are met. A straightforward approach to low–level WCET analysis is to compute the execution time of each basic block\footnote{Basic blocks are maximal sequences of instructions that do not contain branches.} and to add these up afterwards. However, simply adding up the execution times of the basic blocks yields pessimistic bounds. The major problem of a tight WCET analysis is that the maximal number of computation steps, e.g., in a loop, may heavily depend on the input data.

Another important issue concerning WCET analysis is the increasing complexity of embedded systems. There is a growing trend towards distributed and parallel systems which are notoriously hard to implement. On the background of safety critical applications and their economical impact, it is therefore surprising that the design of embedded real-time systems still suffers from a lack of seamlessly integrated design tools. One reason for this is that embedded systems are usually heterogeneous, i.e., their development often requires the design of software at different levels of abstraction as well as specialized hardware designs. The increasing size and complexity of software naturally raised the need for new description languages like message sequence charts, SystemC, statecharts, synchronous languages, and others. However, the introduction of new tools and methods has made the design flow even more heterogeneous which complicates WCET analysis. For this reason, WCET analysis techniques on the level of machine code provide a unifying basis for software descriptions, since all the different tools finally generate assembler programs.

Recently, we presented a new approach to WCET analysis which is based on translating programs given in synchronous languages like Esterel [2] or Quartz [24, 25] to
an equivalent finite state transition system [16]. The key idea is then to use symbolic methods for a complete state space exploration where all possible inputs are considered at once. During this state space exploration, the minimal and maximal numbers of transitions are counted.

In this paper, we follow a similar approach to [16], but do not presuppose an implementation given in a synchronous language. Instead, we take into account the target architecture to open our tools towards other design flows and to obtain an accurate estimation of the worst case execution time. This is achieved by considering assembler programs. Using symbolic simulation of assembler programs, we are able to consider all possible computations and to determine tight WCET bounds. The estimates are given at a logical level, since we compute the number of executed assembler instructions. For the sake of simplicity, we do not yet take into account pipeline and cache effects, even though this is possible.

Our approach takes advantage of established techniques for efficiently manipulating large systems using implicit set representations. These have been originally developed for representing finite state transition systems by means of binary decision diagrams (BDDs) [4]. BDDs provide a canonical representation of formulas in propositional logic and have been used for many applications, in particular for model checking [7], equivalence checking [8], and other forms of state space exploration. However, using propositional logic has the drawback that only finite data types can be considered. In addition, these have to be encoded by Boolean variables to be representable by BDDs. However, the large state space of the obtained data flow is one of the main sources of the state space explosion problem.

Due to these difficulties and restrictions, we use Presburger arithmetic instead of propositional logic to capture the semantics of assembler programs at a higher level of abstraction. Presburger arithmetic has also been used for many other applications, e.g. for symbolic model checking [6] and for reachability analysis of extended finite state machines (EFSMs) [14]. Presburger arithmetic can be translated to finite automata [3, 12, 28] which equips us with an efficient data structure for storing and manipulating possibly infinite sets during symbolic simulation. In order to achieve a compact representation of automata, we have chosen a semi-symbolic representation where sets of transitions between automaton states are encoded with BDDs. To summarize, our approach offers the following advantages:

- tight estimates by handling semantic dependencies
- no overestimation even for data dependent loops
- independence of the design flow
- architecture specific (processor, compiler, etc.)
- fully automated, i.e. no manual annotations required

In the next section, we define XPres, an extension of Presburger arithmetic. We need this logic to formally define the semantics of assembler programs. Then, we sketch the translation of XPres formulas to automata (Section 3) and consider some details of our implementation (Section 4) that are important to obtain industrial strength tools. In Section 5, we describe how XPres is used to translate assembler programs to transition systems, and show how WCET analysis is performed by symbolic simulation. Finally, we present experimental results (Section 6) and conclude with a summary and directions for future work (Sections 7).

2. Extended Presburger Arithmetic XPres

As already mentioned, the basic formalism of our approach is an extension of Presburger arithmetic [21]. Presburger arithmetic is a decidable subset of the first-order theory of the natural numbers where multiplication and division are not allowed. In contrast to the original definition of Presburger arithmetic, we interpret the logic over the integers $\mathbb{Z}$ rather than over the natural numbers $\mathbb{N}$. Furthermore, we introduce logical operations on numbers since these are common to all instruction set architectures and frequently used in assembler programs. The syntax of extended Presburger arithmetic is defined as follows:

**Definition 1 (Syntax of Ext. Presburger Arithmetic)** Let $\mathcal{V} = \mathcal{V}_b \cup \mathcal{V}_i$ be a finite set of variables with $\mathcal{V}_b \cap \mathcal{V}_i = \emptyset$ where $\mathcal{V}_b$ denotes a set of Boolean and $\mathcal{V}_i$ a set of integer variables, respectively. Then, the set of extended Presburger terms $\text{XTerm}_\mathcal{V}$ is the smallest set satisfying the following rules:

- $\mathbb{Z} \subseteq \text{XTerm}_\mathcal{V}$
- $\mathcal{V}_i \subseteq \text{XTerm}_\mathcal{V}$
- $\tau + \pi, \tau - \pi \in \text{XTerm}_\mathcal{V}$ provided that $\tau, \pi \in \text{XTerm}_\mathcal{V}$
- $c \tau \in \text{XTerm}_\mathcal{V}$ provided that $c \in \mathbb{Z}$ and $\tau \in \text{XTerm}_\mathcal{V}$
- $\neg \tau$ provided that $\tau \in \text{XTerm}_\mathcal{V}$
- $\tau \land \pi \in \text{XTerm}_\mathcal{V}$ provided that $\tau, \pi \in \text{XTerm}_\mathcal{V}$

The set of extended Presburger formulas $\text{XPres}_\mathcal{V}$ is the smallest set satisfying the following rules:

- $\mathcal{V}_b \subseteq \text{XPres}_\mathcal{V}$
- $\tau \leq \pi \in \text{XPres}_\mathcal{V}$ provided that $\tau, \pi \in \text{XTerm}_\mathcal{V}$
- $\neg \varphi \in \text{XPres}_\mathcal{V}$, $\varphi \land \psi \in \text{XPres}_\mathcal{V}$ provided that $\varphi, \psi \in \text{XPres}_\mathcal{V}$
- $\exists v. \varphi \in \text{XPres}_\mathcal{V}$ provided that $\varphi \in \text{XPres}_\mathcal{V}$ and $v \in \mathcal{V}$

In order to explain the semantics of XPres, we need some notation on bitvector arithmetic. First of all, we have to define a particular encoding of integers by strings over $\{0, 1\}$. 

In the next section, we define XPres, an extension of Presburger arithmetic. We need this logic to formally define the semantics of assembler programs. Then, we sketch the translation of XPres formulas to automata (Section 3) and consider some details of our implementation (Section 4) that are important to obtain industrial strength tools. In Section 5, we describe how XPres is used to translate assembler programs to transition systems, and show how WCET analysis is performed by symbolic simulation. Finally, we present experimental results (Section 6) and conclude with a summary and directions for future work (Sections 7).
In the following, we use the two’s complement encoding with \( b_i \in \{0, 1\} \):

\[
\text{enc}(b_n \ldots b_0) := -b_n 2^n + \sum_{i=0}^{n-1} b_i 2^i
\]

Recall that the most significant bit \( b_n \) determines the sign: \( b_n = 1 \) yields negative numbers, and \( b_n = 0 \) yields positive numbers or zero. Moreover, a well-known matter of fact (that we use later on) is the sign extension lemma which means that the following equation is valid for any bitvector \( b_n \ldots b_0 \):

\[
\text{enc}(b_n \ldots b_0) = \text{enc}(b_n \ldots b_n b_n \ldots b_0) \quad \text{for every integer } k
\]

For this reason, there are infinitely many bitstrings \( b \) such that \( \text{enc}(b) = c \) holds for any \( c \in \mathbb{Z} \). Hence, the encoding is not injective, but surjective, which suffices for our purposes. The value of a Boolean variable can be simply encoded by the leading bit, i.e., the sign.

The semantics is defined by means of a variable assignment \( \varsigma : \mathcal{V} \rightarrow \mathbb{Z} \). To this end, we introduce an operation for modifying variable assignments: \( \varsigma[x/c](x) := c \) and \( \varsigma[x/c](y) := \varsigma(y) \) for every variable \( y \) different from \( x \).

**Definition 2 (Semantics of Ext. Presburger Arithmetic)**

Given a variable assignment \( \varsigma : \mathcal{V} \rightarrow \mathbb{Z} \) for a finite set of variables \( \mathcal{V} = \mathcal{V}_\mathbb{Z} \cup \mathcal{V}_B \), we define its extension \( \varsigma^\ast : \text{XTerm}_\mathcal{V} \rightarrow \mathbb{Z} \) as follows:

\[
\begin{align*}
\varsigma^\ast(v) &:= \varsigma(v) & \text{for any } v \in \mathcal{V}_\mathbb{Z} \\
\varsigma^\ast(c) &:= c & \text{for any } c \in \mathbb{Z} \\
\varsigma^\ast(\tau + \pi) &:= \varsigma^\ast(\tau) + \varsigma^\ast(\pi) \\
\varsigma^\ast(\tau - \pi) &:= \varsigma^\ast(\tau) - \varsigma^\ast(\pi) \\
\varsigma^\ast(\tau \cdot \pi) &:= c \cdot \varsigma^\ast(\tau) \\
\varsigma^\ast(\tau \div \pi) &:= \text{enc}(b_n \ldots b_0) \text{ for any bitvecotor } b_n \ldots b_0 \\
&\quad \text{with } \text{enc}(b_n \ldots b_0) = \varsigma^\ast(\tau) \\
\varsigma^\ast(\tau \land \pi) &:= \text{enc}((a_n \land b_n) \ldots (a_0 \land b_0)) \text{ for all bitvectors } a_n \ldots a_0 \text{ and } b_n \ldots b_0 \text{ with } \text{enc}(a_n \ldots a_0) = \varsigma^\ast(\tau) \text{ and } \text{enc}(b_n \ldots b_0) = \varsigma^\ast(\pi)
\end{align*}
\]

Given a function \( \varsigma^\ast : \text{XTerm}_\mathcal{V} \rightarrow \mathbb{Z} \), we define a function \( \varsigma^\circ : \text{XPres}_\mathcal{V} \rightarrow \mathbb{B} \) as follows:

\[
\begin{align*}
\varsigma^\circ(v) &:= \begin{cases} 1 & \text{if } \varsigma^\ast(v) < 0 \\
0 & \text{if } \varsigma^\ast(v) \geq 0 \end{cases} & \text{for } v \in \mathcal{V}_B \\
\varsigma^\circ(\tau \leq \pi) &:= 1 & \text{for any } \tau, \pi \in \mathcal{V} \\
\varsigma^\circ(\neg \varphi) &:= \begin{cases} 1 & \text{if } \varsigma^\circ(\varphi) = 0 \\
0 & \text{if } \varsigma^\circ(\varphi) = 1 \end{cases} \\
\varsigma^\circ(\varphi \land \psi) &:= \begin{cases} 1 & \text{if } \varsigma^\circ(\varphi) = 1 \text{ and } \varsigma^\circ(\psi) = 0 \\
0 & \text{otherwise} \end{cases} \\
\varsigma^\circ(\exists \varphi) &:= 1 & \text{iff there is a } c \in \mathbb{Z} \text{ such that } \varsigma^\circ[\sigma/c](\varphi) = 1
\end{align*}
\]

In the following, let \( \llbracket \varphi \rrbracket := \{ \varsigma : \mathcal{V} \rightarrow \mathbb{Z} \mid \exists w \in \text{Lang}(\mathcal{A}) \forall v_i \in \mathcal{V}. \varsigma(v_i) = \text{enc}(w_{i,n} \ldots w_{i,0}) \} \) denote the set of assignments that satisfy a formula \( \varphi \in \text{XPres}_\mathcal{V} \).

### 3. Translating XPres to Automata

It is well-known that Presburger arithmetic is decidable [21]. In particular, one can translate every Presburger formula to a finite automaton \( \mathcal{A}_\mathbf{P} \) that encodes the models \( \llbracket \varphi \rrbracket \) of \( \varphi \) (see also [5, 3, 28]). As there exists for every finite automaton an equivalent deterministic one, and even more a canonical (minimal) one, we can use automata as canonical representations for \( \llbracket \varphi \rrbracket \). This is very much in the same spirit as binary decision diagrams (BDDs) are used as canonical normal forms for propositional logic.

The relationship between finite automata and sets of variable assignments \( \varsigma : \mathcal{V} \rightarrow \mathbb{Z} \) is established via our encoding of integers by bitvectors. For a finite set of variables \( \mathcal{V} = \{v_1, \ldots, v_m\} \), we consider nonempty finite words over the alphabet \( \mathbb{B}^m \) and interpret them as variable assignments. More precisely, a word \( w \) over the alphabet \( \mathbb{B}^m \) with \( w_{i,j} \in \mathbb{B} \)

\[
w = \left( \begin{array}{c}
w_{1,n} \\
\vdots \\
w_{m,n}
\end{array} \right) \end{array} \right) \end{array}
\]

is interpreted as the variable assignment \( \varsigma_w : \mathcal{V} \rightarrow \mathbb{Z} \) with \( \varsigma_w(v_i) := \text{enc}(w_{i,n} \ldots w_{i,0}) \) for every variable \( v_i \in \mathcal{V} \). Using this encoding scheme, the \( i \)-th row is a bitvector encoding of the value of the \( i \)-th variable, and the \( j \)-th column is read by an automaton in its \( j \)-th processing step. Hence, the number of variables is finite and fixed, whereas their bitwidth is also finite, but arbitrarily large.

For example, if \( \mathcal{V} = \{v_1, v_2, v_3\} \) is a set of integer variables, the assignment \( \varsigma_w \) with \( \varsigma_w(v_1) = -3 \), \( \varsigma_w(v_2) = 9 \), \( \varsigma_w(v_3) = 6 \) is represented by the following word \( w \) over the alphabet \( \mathbb{B}^3 \):

\[
w = \left( \begin{array}{cccc}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{array} \right)
\]

The relationship between sets of assignments and finite automata is as follows:

**Definition 3 (Representing Sets of Assignments)** Let \( \mathcal{V} \) be a finite set of variables with \( \mathcal{V} = \{v_1, \ldots, v_m\} \). Any finite automaton \( \mathcal{A} \) over the alphabet \( \mathbb{B}^m \) with language \( \text{Lang}(\mathcal{A}) \) represents the following set of assignments:

\[
\text{Models}_\mathcal{V}(\mathcal{A}) := \{ \varsigma : \mathcal{V} \rightarrow \mathbb{Z} \mid \exists w \in \text{Lang}(\mathcal{A}) \forall v_i \in \mathcal{V}. \varsigma(v_i) = \text{enc}(w_{i,n} \ldots w_{i,0}) \}
\]

As a subtle matter of fact, we have to distinguish between language equivalence and model equivalence: Given two automata \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \) with \( \text{Lang}(\mathcal{A}_1) = \text{Lang}(\mathcal{A}_2) \), we also have \( \text{Models}_\mathcal{V}(\mathcal{A}_1) = \text{Models}_\mathcal{V}(\mathcal{A}_2) \), but not vice versa.
This is due to the sign extension, since it may be the case that an automaton accepts one particular bitvector \( b \) that represents an integer encoding \( \text{enc}(b) \), but not all of the bitvectors that encode \( \text{enc}(b) \). For this reason, we define a closure operation on automata in the following way:

**Definition 4 (Sign Extension Closure)** Let \( \mathcal{A} \) be an automaton with \( \mathcal{A} = (Q, B^m, \Delta, q^0, F) \) where \( Q \) is the set of states, \( B^m \) is the alphabet, \( \Delta \subseteq Q \times B^m \times Q \) is the transition relation, \( q^0 \) is the initial state, and \( F \subseteq Q \) is the set of final states. We define \( \text{cl}(\mathcal{A}) := (Q, B^m, \Delta', q^0, F) \) where \( \Delta' \) is the least transition relation with the following properties:

\[- \Delta \subseteq \Delta' \]
\[- \text{if } (q_0, b, q_1) \in \Delta \text{ and } (q_1, b, q_2) \in \Delta, \text{ then we also have } (q_0, b, q_2) \in \Delta' \]
\[- \text{if } (q_0, b, q_1) \in \Delta, \text{ then we also have } (q_1, b, q_1) \in \Delta' \]

The sign extension closure ensures that two different words which represent the same assignment are either both accepted or both rejected by an automaton. Obviously, the closure of an automaton can be computed by successively adding transitions to \( \Delta \). We then have the following lemma that is important for our construction procedure:

**Lemma 1 (Sign Extension Closure)** The following properties hold for the sign extension closure:

\[- \text{cl}(\text{cl}(\mathcal{A})) = \text{cl}(\mathcal{A}) \]
\[- \text{Models}_{SV}(\text{cl}(\mathcal{A})) = \text{Models}_{SV}(\mathcal{A}) \]
\[- \text{Models}_{SV}(\mathcal{A}_1) = \text{Models}_{SV}(\mathcal{A}_2) \text{ iff } \text{Lang}(\text{cl}(\mathcal{A}_1)) = \text{Lang}(\text{cl}(\mathcal{A}_2)) \]

As a corollary, checking \( \text{Models}_{SV}(\mathcal{A}_1) = \text{Models}_{SV}(\mathcal{A}_2) \) for automata \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \) is decidable, since this problem can be reduced via the closure operation to language equivalence.

A decision procedure for Presburger arithmetic is obtained by translating a given formula \( \varphi \in X\text{Pres}_SV \) to an automaton \( \mathcal{A}_\varphi \) such that \( \llbracket \varphi \rrbracket = \text{Models}_{SV}(\mathcal{A}_\varphi) \). Hence, the formula \( \varphi \) is valid iff the automaton \( \mathcal{A}_\varphi \) accepts any word, and the formula \( \varphi \) is unsatisfiable iff \( \text{Lang}(\mathcal{A}_\varphi) \) is empty.

**Theorem 1 (Translating X\text{Pres}_SV to Automata)** Given a finite set of variables \( \mathcal{V} \), there is an algorithm that computes for any formula \( \varphi \in X\text{Pres}_SV \) an automaton \( \mathcal{A}_\varphi \) such that \( \llbracket \varphi \rrbracket = \text{Models}_{SV}(\mathcal{A}_\varphi) \).

By Definition 1, a term can contain arithmetic and bitvector operators at the same time. This complicates the construction of automata for expressions which consist of both types of operators. In contrast, the translation from purely arithmetic or bitvector type expressions to automata is much easier. For this reason, we separate the arithmetic parts of an expression from its bitvector parts by means of the following rules, where RH \( \in \{=, \neq, <, \leq, >, \geq \} \) and RH \( \in \{\land, \lor, \oplus, \ominus, \otimes, \div \} \):

\[- (\tau_1 \pm \tau_2) \circ \tau_3 \circ \pi := \exists u.((u = \tau_1 \pm \tau_2) \land (u \circ \tau_3 \circ \pi)) \]
\[- (\tau_1 \circ \tau_2) \pm \tau_3 \circ \pi := \exists u.((u = \tau_1 \circ \tau_2) \land (u \pm \tau_3 \circ \pi)) \]

For example, the equation \((x + y) \oplus z = c \) is translated to the formula \( \exists u.((u = x + y) \land (u \oplus z = c)) \). If these rules are recursively applied to an expression, one finally obtains a formula whose basic equations and inequations are either of completely arithmetic or of completely bitvector type.

In the following, we describe the construction of automata for both types of formulas. For the sake of simplicity, we restrict ourselves to equations. In principle, inequations can be handled similarly even though the details are more complicated.

The construction of automata for pure bitvector equations is straightforward. Given an initial state \( q^0 \), an accepting state \( q^a \), and a sink state \( q^s \), the automaton evaluates the equation at each step according the current input letter. If the equation is satisfied for the given bitvector, the automaton moves to the accepting state \( q^a \). Otherwise, it moves to the sink state \( q^s \) which cannot be left. In this way, it is guaranteed that all the letters of a word satisfy the equation. As an example, Figure 1 shows the automaton for the equation \( x \land y = z \). The initial state is used to prevent the automaton from accepting the empty word.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{automaton.png}
\caption{Automaton representing bitwise AND}
\end{figure}

The construction of automata for Boolean variables is also simple: It suffices to test the sign of the corresponding bitvector. The construction of automata for arithmetic equations is somewhat more complex. In the following, we adopt the method given in [28]. As a first step, the equation to be encoded is brought to the form \( \sum_{i=1}^m a_i v_i = c \) with \( m = \|\mathcal{V}\| \), \( v_i \in \mathcal{V} \), and \( a_i, c \in \mathbb{Z} \). The idea is that each state of the automaton corresponds to the value of the left–hand side of the equation for the bits read so far. The only accepting state corresponds to the equation’s right–hand side. Thus, the automaton for an equation is defined by the tuple \( \mathcal{A} = (Q \cup \{q^0\}, \mathbb{B}^m, \Delta, q^0, \{e\}) \) whose states are a finite set of integers, i.e. \( Q \subseteq \mathbb{Z} \), including a special initial state \( q^0 \). The transition relation \( \Delta : Q \times \{q^0\} \times \mathbb{B}^m \rightarrow Q \) is defined as follows with \( a \in \mathbb{Z}^m \), \( b \in \mathbb{B}^m \), and \( a \cdot b \) is defined as follows with \( a \in \mathbb{Z}^m \) and \( b \in \mathbb{B}^m \):
\[ \Delta(q^0, b) = -a \cdot b \]
\[ \Delta(q, b) = 2q + a \cdot b \text{ for } q \neq q^0 \]

Note that the transitions from the initial state read the most significant bits which have a negative value in two’s complement notation. Figure 2 shows a sample automaton which accepts the solutions to the equation \( x + y = z \). States from which there is no path to an accepting state can be collapsed to a single sink state such that the resulting automaton becomes complete.

\[ \begin{align*}
0 & \quad \begin{cases} 
(0, 0), (0, 1), (1, 0) \\
(1, 1)
\end{cases} \\
-1 & \quad \begin{cases} 
(0, 0), (0, 1), (1, 1) \\
(1, 0)
\end{cases}
\end{align*} \]

Figure 2. Automaton representing addition

Once we have constructed the basic automata, we can combine them using the following definitions: Given two automata \( A_{\varphi} = (Q_{\varphi}, \mathbb{B}^m, \Delta_{\varphi}, q^0_{\varphi}, F_{\varphi}) \) and \( A_\psi = (Q_\psi, \mathbb{B}^m, \Delta_\psi, q^0_\psi, F_\psi) \), we have:

\[ A_{\varphi \wedge \psi} := (Q_{\varphi} \times Q_\psi, \mathbb{B}^m, \Delta_{\varphi \wedge \psi}, (q^0_{\varphi}, q^0_\psi), F_{\varphi} \times F_\psi) \]

\[ A_{\varphi \vee \psi} := \exists (Q_{\varphi} \times Q_\psi, \mathbb{B}^m, \Delta_{\varphi \vee \psi}, (q^0_{\varphi}, q^0_\psi), F_{\varphi} \times F_\psi) \text{ where } ((p_{\varphi}, p_\psi), b, (q_{\varphi}, q_\psi)) \in \Delta \text{ iff } (p_{\varphi}, b, q_{\varphi}) \in \Delta_{\varphi} \text{ and } (p_\psi, b, q_\psi) \in \Delta_\psi \]

The Boolean operations follow the usual definitions of automata theory, as e.g. described in [13]. Existential quantification can be performed by erasing one or more positions in the bitvectors that constitute the alphabet. In contrast to automata theory, as e.g. described in [13], existential quantification can be performed by erasing one or more positions in the bitvectors that constitute the alphabet. In contrast to automata theory, as e.g. described in [13].

\[ \begin{align*}
f_1 & = x\bar{y}z \lor \bar{x}yz \lor x\bar{y}z \\
f_2 & = \bar{x}yz \lor x\bar{y}z \lor xyz \\
f_3 & = \bar{x}yz \lor x\bar{y}z \lor xyz \\
f_4 & = xy\bar{z}
\end{align*} \]

For each state \( q \in \{q_0, 0, -1\} \) and for each transition, there is exactly one BDD representing a set of letters.

\[ \begin{align*}
q_0 & \quad 0 \quad f_1 \quad -1 \quad f_2 \\
0 & \quad \quad f_1 \quad -1 \quad f_2 \\
-1 & \quad -1 \quad f_1 \quad \quad 0 \quad f_2
\end{align*} \]

Figure 3. Semi–symbolic representation of the automaton shown in Figure 2

\[ \begin{align*}
\Delta(q^0, b) = -a \cdot b \\
\Delta(q, b) = 2q + a \cdot b \text{ for } q \neq q^0
\end{align*} \]
The operations on the chosen data structure are implemented as follows: Given complete and deterministic automata \( A_\varphi \) and \( A_\psi \) with \( A_\varphi = (Q_\varphi, V, (\delta_\varphi, I_\varphi), q_0^\varphi, F_\varphi) \) and \( A_\psi = (Q_\psi, V, (\delta_\psi, I_\psi), q_0^\psi, F_\psi) \), the Boolean operations follow immediately from the above definitions with \( \circ \in \{ \land, \lor, \neg, \rightarrow, \Rightarrow \} \):

\[
A_{\varphi \circ \psi} := (Q_\varphi \times Q_\psi, V, (\delta_\varphi, I_\varphi), q_0^\varphi, F_\varphi)
\]

with \( \delta = \{( (q_\varphi, q_\psi), (q_\varphi', q_\psi') ) \mid (q_\varphi, q_\psi) \in \delta_\varphi \land (q_\varphi', q_\psi') \in \delta_\psi \} \) and \( I(I_\varphi \times I_\psi) = (I_\varphi(q_\varphi) \land I_\psi(q_\psi)) \).

Since existential quantification of a variable \( i \) is performed by erasing the \( i \)-th position of the alphabet, we abstract the corresponding variable \( w_i \) in the associated BDD:

\[
A_{\exists w_i} := (Q_\varphi, V, (\delta_\varphi, I_\varphi), q_0^\varphi, F_\varphi)
\]

with \( \delta = \{( (q_\varphi, q_\psi), (q_\varphi', q_\psi') ) \mid (q_\varphi, q_\psi) \in \delta_\varphi \land (q_\varphi', q_\psi') \in \delta_\psi \} \).

As mentioned previously, we have to perform a determinization step after each projection operation. To keep the size of the resulting automaton small, we construct only reachable states and omit transitions labeled with a zero BDD. In addition, the automata are minimized regularly.

### 5. WCET Analysis

In the previous sections, we described the base logic XPres and its translation to finite automata. In this section, we show how assembler programs can be translated to equivalent transition systems. To this end, we introduce the notion of integer Kripke structures which serve as the basis for the WCET analysis algorithm.

**Definition 5 (Integer Kripke Structure (IKS))** An integer Kripke structure (IKS) over a finite set of variables \( V \) is a transition system \( K = (S, I, R) \) where \( S \) is the possibly infinite set of states, \( I \subseteq S \) are the initial states, and \( R \subseteq S \times S \) is the transition relation. Every state \( \xi \in S \) is a variable assignment \( \xi : V \rightarrow \mathbb{Z} \). In addition, it is required that the set of initial states \( I \) and the transition relation \( R \) are definable in XPres, i.e., they must be regular sets.

According to the above definition, a state of an IKS is a variable assignment for the variables \( V \). Regarding the modeling of an assembler program, an assignment describes the current values of the processor’s registers and the used memory locations. 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 (cf. Figure 4). Hence, we can represent the transition relation of a program by a Presburger formula \( \text{XPres}_{\mathcal{V}, \mathcal{V}', \mathcal{V}''} \) where \( \mathcal{V} \) and \( \mathcal{V}' \) are the register contents of the current and the next state, respectively. Consequently, the transition relation is regular and can be represented by a finite automaton. Note that from a theoretical point of view, we can model every program with an IKS, since the operations of register machines [18] are expressible in Presburger arithmetic.

The translation of an assembler program to an IKS is straightforward, provided that we have a definition of the semantics of the processor’s instruction set. Assume that for each register \( r \) we have an associated pair of variables \( r \) and \( r' \). In particular, let \( pc \) and \( pc' \) denote the program counter for the current and next instruction, respectively. Moreover, let \( P = I_0, \ldots, I_n \) be the program to be modeled, i.e., a sequence of instructions such that \( I_k \) is the instruction located at memory address \( k \). Then, the transition relation for \( P \) can be obtained as follows where \( S(I) \) defines the semantics of an instruction \( I \) (cf. Table 1):

\[
R := \bigvee_{k \in \{0, \ldots, n\}} (pc = k) \land S(I_k)
\]

The set of initial states \( I \) specifies the initial register contents and the address of the first instruction to be executed. To obtain tight bounds on the execution time, it is often required to specify preconditions for the program. Suppose for example that one of the inputs is given by an analog/digital converter which produces values in the range between 0 and \( 2^8 - 1 \). In contrast to other approaches to WCET analysis, we can define the initial state set using this information to avoid overestimation.

Preconditions are also useful in compositional methods where the system is divided into parts which are analyzed separately. The overall execution time can be obtained by adding up the results for the subsystems. As mentioned in the introduction, this yields an overly pessimistic estimation if dependencies between the subsystems are neglected. However, using symbolic simulation, we can determine the set of possible output values for one subsystem and use this set as input constraints for other subsystems.

In the following, we consider the MIPS32 architecture which defines a simple yet powerful instruction set [19]. Table 1 shows the semantics of some selected instructions. Note that memory locations are modeled by a set of variables \( M_i \).² The following function ensures that registers, which are not affected by an instruction, remain unchanged:

\[
C(R) := \bigwedge_{r \in W} (r' = r) \land W = V \setminus (R \cup \{pc\})
\]

In order to execute a program symbolically, we have to compute the successor states for a set of states \( Q \), i.e., the image of the transition relation with respect to \( Q \):

\[
\text{Img}_R(Q) := \{ \xi' \mid \exists \xi \in Q \land (\xi, \xi') \in R \}
\]

²Our semi-symbolic representation of automata allows us to deal with large sets of variables, since transitions are encoded by BDDs.
As usual in symbolic methods, image computation can be performed by intersection, existential quantification (projection), and variable substitution [8]. 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. Given an IKS $K = (S, I, R)$ and a set of final states $F$, the worst case execution time of a program modeled by a transition relation $R$ can be computed by the following algorithm:

\begin{verbatim}
function WCET :
n := 0;
Q := I;
while $Q \not\subseteq F$ do
  $Q := \text{img}_R(Q);$
  n := n + 1;
end:
return n;
end;
\end{verbatim}

Usually, final states are those states where the program counter points to the last instruction (e.g. return from subroutine). As an example for the construction of an IKS from an assembler program, Figure 4 shows the core of a von Neumann adder, its translation to Presburger arithmetic, and a part of the corresponding Kripke structure.

It can happen that the above algorithm does not terminate, in particular for programs whose execution times depend on the input values. In practice, this is not a restriction since we can constrain the initial register contents to their minimal and maximal values, respectively. Additionally, we can stop the algorithm when the time budget given to the program is exceeded as proposed in [10].

### 6. Experimental Results

To evaluate our approach, we implemented and integrated the algorithms in our Equinox toolbox [26] and applied them to some benchmarks. All experiments were performed on a Pentium 4 processor with 2 GHz clock frequency and 256 MB RAM. For the representation and manipulation of BDDs we used the CUDD package [27]. The benchmarks were written in C and compiled with the GNU C-compiler to obtain assembly code for the MIPS R3000 processor family (code optimization was enabled using option -O2).
Table 2. Results and runtimes of WCET analysis

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>8 Bits</th>
<th></th>
<th>12 Bits</th>
<th></th>
<th>16 Bits</th>
<th></th>
<th>20 Bits</th>
<th></th>
<th>24 Bits</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WCET</td>
<td>Time</td>
<td>WCET</td>
<td>Time</td>
<td>WCET</td>
<td>Time</td>
<td>WCET</td>
<td>Time</td>
<td>WCET</td>
<td>Time</td>
</tr>
<tr>
<td>vonNeumann</td>
<td>38</td>
<td>0.43</td>
<td>54</td>
<td>1.75</td>
<td>70</td>
<td>6.19</td>
<td>86</td>
<td>17.82</td>
<td>102</td>
<td>43.53</td>
</tr>
<tr>
<td>IntRoot</td>
<td>97</td>
<td>0.45</td>
<td>385</td>
<td>3.44</td>
<td>1537</td>
<td>24.51</td>
<td>6145</td>
<td>166.15</td>
<td>24577</td>
<td>1044.59</td>
</tr>
<tr>
<td>RussMult</td>
<td>60</td>
<td>0.35</td>
<td>88</td>
<td>1.01</td>
<td>116</td>
<td>2.44</td>
<td>144</td>
<td>5.10</td>
<td>172</td>
<td>9.34</td>
</tr>
<tr>
<td>Booth</td>
<td>84</td>
<td>0.92</td>
<td>122</td>
<td>3.02</td>
<td>160</td>
<td>7.88</td>
<td>198</td>
<td>17.70</td>
<td>236</td>
<td>34.19</td>
</tr>
<tr>
<td>ErGu97</td>
<td>216</td>
<td>233.14</td>
<td>216</td>
<td>254.33</td>
<td>216</td>
<td>273.23</td>
<td>216</td>
<td>296.02</td>
<td>216</td>
<td>308.43</td>
</tr>
<tr>
<td>Collatz</td>
<td>813</td>
<td>114.47</td>
<td>-</td>
<td>&gt;1h</td>
<td>-</td>
<td>&gt;1h</td>
<td>-</td>
<td>&gt;1h</td>
<td>-</td>
<td>&gt;1h</td>
</tr>
<tr>
<td>GCD</td>
<td>1531</td>
<td>110.05</td>
<td>-</td>
<td>&gt;1h</td>
<td>-</td>
<td>&gt;1h</td>
<td>-</td>
<td>&gt;1h</td>
<td>-</td>
<td>&gt;1h</td>
</tr>
</tbody>
</table>

Table 2 shows the results for different bitwidths which are 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 (WCET) are given in number of executed instructions. The runtimes of our analysis algorithm are given in seconds. The first benchmark program (von Neumann adder) could be analyzed without any modifications of the assembly code in less than one minute. For the IntRoot benchmark, the runtimes are worse, but still acceptable.

The RussMult and Booth benchmarks were simplified to reduce the runtime of the WCET algorithm. The idea behind this simplification is that we are not interested in the results of the algorithms, but merely in their worst case execution times. In other words, we can omit some computations provided that the algorithm’s execution time remains unaffected. For the two benchmarks, we simply replaced those instructions that compute the result, but do not affect the termination condition of the loops with NOPs.

The benchmark ErGu97 was taken from [10] and consists of two interdependent loops. Since the number of iterations of the loops is bounded, the runtimes of our algorithm are nearly constant. The last two benchmarks could not be simplified in a straightforward manner and are hardly tractable with symbolic simulation since the automata become exponentially large. Note that for all the benchmarks except Collatz and IntRoot, an exhaustive analysis using explicit techniques would require to simulate the algorithms for $2^{24}$ input values in the case of 24 bit numbers.

In a second set of experiments, we determined the amount of memory which is needed by the benchmark programs. For that purpose, we measured the size of the automata which represent the transition relations of the benchmark programs (Table 3). The first two columns specify the number of automaton states and the number of transitions, respectively. The third column shows the total number of BDD nodes which are used to encode the transitions. Moreover, we determined the runtimes for constructing the automata (fourth column, given in seconds). As can be seen, the benchmark programs can be quickly translated to finite automata which provide a succinct representation of the transition relations. It also follows from Table 3 that the complexity of WCET analysis mainly depends on the size of the state sets and not on the size of the transition relation.

Table 3. Benchmark transition relations

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>States</th>
<th>Trans.</th>
<th>Nodes</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>vonNeumann</td>
<td>10</td>
<td>29</td>
<td>195</td>
<td>1.06</td>
</tr>
<tr>
<td>IntRoot</td>
<td>19</td>
<td>62</td>
<td>628</td>
<td>0.11</td>
</tr>
<tr>
<td>RussMult</td>
<td>14</td>
<td>45</td>
<td>301</td>
<td>0.10</td>
</tr>
<tr>
<td>Booth</td>
<td>27</td>
<td>88</td>
<td>588</td>
<td>0.35</td>
</tr>
<tr>
<td>ErGu97</td>
<td>42</td>
<td>146</td>
<td>734</td>
<td>1.30</td>
</tr>
<tr>
<td>Collatz</td>
<td>23</td>
<td>79</td>
<td>620</td>
<td>0.12</td>
</tr>
<tr>
<td>GCD</td>
<td>19</td>
<td>62</td>
<td>398</td>
<td>0.91</td>
</tr>
</tbody>
</table>

For the tractable benchmarks, Table 4 shows the size of the largest automaton constructed during WCET analysis. The last column shows the total number of bytes which are used to represent this automaton. These results indicate, that memory consumption is not a crucial concern for our approach unless the program under consideration causes an exponential blow-up such as Collatz and GCD. Our experiments have also shown that most of the time is spent for automaton minimization, and hence, efficient minimization algorithms as proposed in [12] are beneficial.

Table 4. Memory consumption (24 Bits)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>States</th>
<th>Trans.</th>
<th>Nodes</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>vonNeumann</td>
<td>425</td>
<td>1464</td>
<td>21</td>
<td>23760</td>
</tr>
<tr>
<td>IntRoot</td>
<td>100</td>
<td>227</td>
<td>106</td>
<td>5328</td>
</tr>
<tr>
<td>RussMult</td>
<td>119</td>
<td>361</td>
<td>77</td>
<td>7008</td>
</tr>
<tr>
<td>Booth</td>
<td>181</td>
<td>606</td>
<td>155</td>
<td>12176</td>
</tr>
<tr>
<td>ErGu97</td>
<td>851</td>
<td>2540</td>
<td>338</td>
<td>46048</td>
</tr>
</tbody>
</table>
We presented an exact technique for analyzing the worst case execution time of assembler programs. It determines the maximal number of executed instructions for all possible computations by means of symbolic simulation. This is accomplished by translating the program to an integer Kripke structure which can be represented by formulas in Presburger arithmetic. Using Presburger arithmetic, we are able to model most instructions directly such as addition and bitvector operations which belong to the basic instructions of all processors and micro–controllers. The translation of Presburger arithmetic to finite automata gives us an efficient means for symbolically traversing the state space of the program.

Our method is particularly well–suited for algorithms in real–time systems whose execution times highly depend on the input values and are thus hard to estimate accurately using conventional methods. Tight estimates of worst case execution times are especially important for core routines which are frequently called and contribute a large part to the total execution time, e.g. interrupt service routines of real–time operating systems. Moreover, our approach neither imposes any restrictions on the structure of the programs nor requires the user to annotate the programs with information about the expected behavior.

However, it does not aim at analyzing large systems which consist of thousands of lines of code. We believe that a complete and accurate WCET analysis of large systems can only be achieved by combining different methods for each level of abstraction. In such a framework, our approach can be applied to critical modules whose execution times are data dependent and have significant impact on the whole system.

Currently, our algorithms do not support multiplication as a single operation which is due to the fact that Presburger arithmetic would otherwise be undecidable. However, we plan to incorporate multiplication into our tools by reduction to successive additions. To this end, we have to extend our algorithms such that intermediate computations do not contribute to the execution time. As a more general approach, one could also label each transition with a natural number that represents the time units required to take the transition.

Furthermore, it is advantageous to simplify the transition relations in order to reduce the runtimes of our algorithms. For this reason, we are working on compiler techniques that replace instructions which do not affect the execution time with NOPs. This can be achieved by a kind of dead-code elimination in order to remove redundant computations [20]. In addition, we do not yet consider pipeline hazards, cache misses, branch prediction, and arithmetic overflows which is part of our future work.


