Efficient Analysis of Pipeline Models for WCET Computation

Stephan Wilhelm
AbsInt GmbH and Saarland University
Saarbrücken, Germany
sw@absint.com

April 17, 2005

Abstract

Worst-case execution time (WCET) predicion for modern CPU's cannot make local assumptions about the impact of input information on the global worst-case because of the existence of *timing anomalies*. Therefore, a large subset of the reachable states of the CPU must be considered. As the number of states grows, WCET prediction can become infeasible because of the increase in computation time and memory consumption. This paper presents a solution for this problem by defining WCET computation in terms of operations on binary decision diagrams (BDD's).

1 Introduction

Finding the worst-case execution time (WCET) for all tasks of a software is an important requirement in the design of hard real-time systems. Because the execution time depends on the underlying processor hardware, WCET computation requires a detailed analysis of the hardware behavior for the analyzed task. For CPU's using modern techniques for reducing the average execution time, such as caches, pipelined execution, branch prediction, speculative execution, and out-of-order execution, the WCET cannot be obtained by measurements because it is usually not possible to determine the worst-case inputs manually.

A proven approach for obtaining tight upper bounds of the WCET has been presented in [8]. It employs several semantics-based static program analyses on the assembly level control flow graph (CFG) of the input program. First, the *value analysis* computes the address ranges for instructions accessing memory. In a second step, an integrated *cache-* and *pipeline-analysis* predicts the cache behavior [7] and the behavior of the program on the processor pipeline [13]. The result of the pipeline analysis is the WCET for each basic block from which a subsequent *path analysis* [12] computes the global worst-case path.

Pipeline analysis computes sets of pipeline states that can occur at any point in the program. Imprecisions in its input information, arising from unknown memory accesses or unknown cache behavior (may be cache hit or miss), cause situations where each pipeline state can have several successor states. Unfortunately, it has been proven that for CPU's, using modern techniques for reducing the average execution time, it is not possible to decide locally which element from the input set triggers the global worst-case behavior. E. g. a cache *hit* might contribute to the global worst-case. Such cases has been termed *timing anomalies* [9]. Because of the presence of timing anomalies, pipeline analysis must consider all possible successor states. For complex pipelines with large state spaces, the analysis can become infeasible because of the increase in memory consumption and computation time [13]. This problem is known as *state explosion* and it is also a well known phenomenon in the area of model checking. The use of ordered binary decision diagrams (OBDD's) [4] for symbolic set operations has significantly reduced the state explosion problem for model checking and the size of systems that have been successfully verified by model checking has increased ever since [5]. The key idea of this paper is to define pipeline analysis in terms of BDD [1] operations, similar to symbolic model checking. It can be expected, that this will reduce runtime and memory consumption of pipeline analyses, making the analysis of complex pipelines feasible, even for large programs.

[1] The terms BDD and OBDD are used interchangeably in this text.

2 Finite state machines

Processor pipelines can be regarded as finite state machines (FSM's) and pipeline analysis can be defined as a computation on sets of states of the FSM for the analysed pipeline. The efficiency of the presented approach for pipeline analysis relies on the BDD-based representation of FSM's which is introduced in this section.

Definition 1 A *Finite State Machine (FSM) M* is a triple, (Q, I, T), where Q is the set of states, I is the set of input values, and $T = (Q \times I \times Q)$ is the transition relation.

Each set of FSM states $A \subseteq Q$ can be associated to its *characteristic function* $\mathbf{A} : Q \rightarrow \{0, 1\}$; $\mathbf{A}(x) = 1 \Leftrightarrow x \in A$. In the same way, the transition relation T can be associated to the function $\mathbf{T} : Q \times I \times Q \rightarrow \{0, 1\}$; $\mathbf{T}(x, i, y) = 1 \Leftrightarrow (x, i, y) \in T$. It is common practice to represent FSM state sets and the FSM transition relation by their characteristic functions encoded as BDD's. This representation has the advantage of compactly representing a large number of commonly encountered functions. Useful operations such as negation, conjunction and existential quantification can be efficiently performed using BDD's.

A *hardware design* consists of a set of interconnected latches and gates. A design with n latches and m input wires is characterized by an associated FSM with state space $Q = \{0, 1\}^n$ and input space $I = \{0, 1\}^m$. The transition relation is defined by the corresponding logic. For such models, the variables of BDD's for encoding the characteristic functions of states sets represent the latches of the design.

Definition 2 Given a FSM (Q, I, T) and a set of states $A \subseteq Q$. The *image* of A, $\mathrm{Img}(A) \subseteq Q$, is the set of states that is reachable from A under T.

Image computation is the core operation of symbolic model checking algorithms [10]. Section 4 shows that it can also be used for dataflow analysis of pipeline models. Image computation can become infeasible for large designs if the transition relation is given as a single BDD [3] but there are efficient algorithms for image computation that avoid building the complete transition relation by exploiting the fact that the FSM transition relation can be factored into the transition relations of the involved latches [10].

```
reg [0:3] cycles;
reg [0:1] instr;
reg [0:1] delay;

initial cycles = 0;
initial instr = 0;
initial delay = 0;

always @(clk_first) begin
   if (cycles == 7)
      cycles = 0;
   else
      cycles = cycles + 1;
end

always @(clk_second) begin
    if (delay == 0)
        delay = get_delay();
    else
        delay = delay - 1;
end

always @(clk_third) begin
    if (delay == 0)
        instr = get_next_instr();
end
```

Figure 1: Example Verilog code for simple FSM.

3 Specifying pipeline models

Hardware description languages like VHDL or Verilog have been designed for writing concise descriptions of hardware designs in terms of latches and update logic. It has been shown that such specifications can be compiled into (timed) finite state machines [6]. The *VIS* system for model checking and synthesis of hardware designs supports a substantial subset of Verilog extended by an expression for specifying non-deterministic behavior. Specifications in Verilog are compiled using the *vl2mv* compiler and the resulting description of the system as a finite automaton can be used for CTL modelchecking and reachability analysis [11].

Low-level HDL specifications, including detailed models of pipeline states and the corresponding logic, are readily available for many CPU's and can be compiled into finite automata by vl2mv. The resulting automata are often too large for most kinds of analyses but the problem can be overcome by applying suitable abstractions to the original description. Automatic abstrac-

tion from HDL models is a field of ongoing research [2]. HDL's also support behavioral descriptions of hardware which can be used to describe the (timing) behavior of a design as specified by the manual. Such descriptions are usually more compact and the resulting automata are smaller.

Figure 1 shows an example of Verilog code for a FSM with a two bit delay counter. Note that the declaration `reg[k:l]` denotes a set of $(l-k)+1$ variables of the FSM state with value 0 or 1. Let us assume that this FSM is a simplified pipeline model [2]. It has an instruction pointer, `instr`, for 3 instructions and a delay counter, `delay`, that is intialized with the delay for each instruction. The `cycle` counter counts execution cycles of basic blocks with at most 7 cycles. Execution of one cycle is done in three steps, indicated by the signals `clk_first`, `clk_second` and `clk_third`. Whenever the delay counter reaches 0 and the signal `clk_second` is active, a new value for delay is read using the function `get_delay()`. Similarly, the new value for `instr` is obtained from the function `get_next_instr()` when `delay` is 0 and the signal `clk_third` is active.

When generating a FSM for this description, the input functions are modeled as non-deterministic transitions. Thus, image computation for a state where `delay` is 0 will yield the 3 possible successor states where delay takes the values 0, 1 or 2.

4 Pipeline analysis

Given a hardware model by an FSM, pipeline analysis performs a fixed point iteration on the domain $P(Q)$ of pipeline states. The least fixed point (LFP) is the solution to the data flow problem containing all FSM states that are reachable for a given program point and also containing the WCET state. Note that the FSM state comprises a counter for execution cycles of basic blocks (the number of execution cycles per basic block is clearly finite). The WCET for each basic block B is found by selecting the state with the highest value for the execution cycle counter from all states where the last instruction belonging to B has finished.

4.1 Transfer functions

The transfer functions for pipeline analysis compute the next states for each FSM transition in all current states.

[2]Although the FSM of figure 1 is not a pipeline model, it is sufficient for illustrating the principles of the presented approach for pipeline analysis.

instruction	delay
0	1
1	2
2	{0,1}

Table 1: Example program input for analyzing the model from figure 1.

In general, this is an image computation with the restriction that program analysis is only interested in the set of reachable states under the *concrete* inputs of the program. Image computation as defined in section 2 determines the set of reachable states for *all* possible inputs. Let A_0 be the set of initial states of the FSM (Q, I, T). Then, the following fixed point calculation computes the set of reachable FSM states:

$$A_{k+1} = A_k \cup \mathrm{Img}(A_k)$$

The problem of encoding concrete inputs of the analyzed program can be solved by constructing BDD's for the states where inputs are read and BDD's for the concrete inputs themselves. Remember that the BDD variables for a hardware design are the latches of the design. Let $\cdot$ denote the conjunction of BDD variables and $\neg$ is the negation of a variable. For the example of figure 1, the state where the delay for instruction 1 is read by the function `get_delay()` can be encoded as follows:

$$\mathbf{J}_1 = \texttt{instr<0>} \cdot \neg\texttt{instr<1>} \cdot \neg\texttt{delay<0>} \cdot \neg\texttt{delay<1>} \cdot \texttt{clk\_second}$$

This is the state where the instruction pointer is 1, the value of the delay counter is 0 and the signal `clk_second` is active. Table 1 specifies that the delay for instruction 1 is 2. This concrete input can be encoded as follows:

$$\mathbf{C}_1 = \neg\texttt{delay<0>} \cdot \texttt{delay<1>}$$

The BDD's $\mathbf{J}_1$ and $\mathbf{C}_1$ can be regarded as the characteristic functions of the state sets J_1 and C_1 where the variables have the values encoded in the BDD's. For a set A_k of FSM states, the next states for the concrete input at this program point are the computed by the following formula:

$$A_{k+1,1} = (\mathrm{Img}(A_k \cap J_1)) \cap C_1$$

For n concrete inputs, the next states for the set of states where *no* concrete input information is required

is calculated as follows:

$$A_{k+1,-} = \mathrm{Img}(A_k \setminus \bigcup_{0 \le l \le n} J_l)$$

Finally, the fixed point iteration for pipeline analysis under n concrete input informations from the input program can be computed as:

$$A_{k+1} = (\bigcup_{0 \le l \le n} A_{k+1,l}) \cup A_{k+1,-}$$

Please note that imprecise input information can also easily be encoded as a BDD. E. g. the delay information for instruction 2 in Table 1 is either 0 or 1. The BDD for this input is simply $\mathbf{C}_2 = \neg$`delay<1>`. The sequence in which instructions are analyzed is also an input to the model of figure 1. This input can be determined from the program's CFG and encoded in the same way as the delay input.

The success of BDD based algorithms depends on the size of the involved BDD's, which is very sensitive to the ordering of the BDD variables. Finding a minimum sized BDD for a given logic function is algorithmically intractable. However, there are many heuristics for finding good variable orderings [10]. A good variable ordering must only be found once for each pipeline model.

5 Work in progress

A prototype of the presented approach for pipeline analysis is currently being implemented in aiT for ARM7 [1]. Future targets comprise the Infineon Tricore 2 CPU and the Motorola Power PC family of processors (MPC5xx and MPC755). The MPC755 is the most challenging and interesting target because of the huge state space of the pipeline model. We expect to achieve a significant reduction of computation time and memory consumption for this processor.

6 Conclusion

We have shown that the application of well-known techniques for handling large state sets from the area of model checking to the program semantics-based analysis of pipeline models, can help to handle the increasing complexity of modern processor hardware for WCET computation. For large state sets, BDD based algorithms are more space efficient and faster than implementations using an explicit representation of pipeline states.

Furthermore, we have established a connection between pipeline analysis implementation and pipeline specifications written in Verilog or VHDL. Generating the pipeline analysis from the same specification used for hardware synthesis is faster and less error-prone than the difficult way of manual implementation. Finally, the important task of verification of the analysis is also simplified for analyses generated from HDL specifications.

References

[1] http://www.absint.com/aiT/.

[2] http://www.avacs.org.

[3] A. Aziz, S. Tasiran, and R.K. Brayton. BDD Variable Ordering for Interacting Finite State Machines. In *31st ACM/IEEE Design Automation Conference (DAC)*, San Diego, CA, 1994.

[4] R. Bryant. Graph based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, 1986.

[5] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. IEEE Comp. Soc. Press, 1990.

[6] S.-T. Cheng. Compiling Verilog into Automata, 1994.

[7] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

[8] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of EMSOFT 2001, LNCS 2211*, 2001.

[9] T. Lundquist and P. Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, 1999.

[10] R. Ranjan, A. Aziz, R. Brayton, B. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification, 1995.

[11] The VIS Group. VIS user's manual.

[12] H. Theiling. ILP-based Interprocedural Path Analysis. In *Proceedings of the Workshop on Embedded Software*, Grenoble, France, 2002.

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