

Automatic and Optimized Generation of Compiled High-Speed RTL Simulators

Alexey Kupriyanov, Frank Hannig, and Jürgen Teich
Department of Computer Science 12, Hardware-Software-Co-Design,
University of Erlangen-Nuremberg, Germany,
{kupriyanov, hannig, teich}@cs.fau.de
URL: <http://www12.informatik.uni-erlangen.de>

Abstract

In this paper we focus on the derivation of optimal code when generating high-speed event-driven compiled simulators for processor architectures described on register transfer level (RTL). The simulators' generation is part of a framework, which aims at architecture and compiler co-generation for special purpose processors. The main contribution of this paper is an efficient algorithm to generate optimal if-then-else structures in order to perform the update cycle during the event-driven simulation process. Our approach guarantees that during one simulation cycle a possible change of each register content is checked exactly once and that each register is updated at most once. Additionally, the proposed technique minimizes the code size of the generated simulator. The simulator's superior performance compared to an existing commercial simulator is shown. Finally, we demonstrate the pertinence of our approach by simulating a MIPS processor.

1 Introduction

Today, due to the increasing complexity of processor architectures, 70-80% of the development cycle is spent in validation. Here, beside formal verification methodologies, simulation based validation plays an important role. Furthermore, cycle-accurate and bit-true simulations are necessary for the debug process when mapping applications to a not physically available architecture. Moreover, in the domain of architecture/compiler co-design for application-specific instruction set processors (ASIPs), fast estimation

and simulation methodologies are essential to explore the enormous design space of possible architecture/compiler co-designs.

In the following, we list some significant approaches aiming at architecture simulation. Today's simulation techniques include the flexible but often very slow interpretive simulation and faster compiled simulation. The simulation can be performed at the different levels of abstraction starting from gate level, register transfer level, up to instruction-set (IS) level. The lower the abstraction level used, the higher simulation flexibility is achieved but at the same time the simulation speed dramatically slows down. Nowadays, the IS simulators are widely used in the domain of application specific instruction set processors because of their extremely high simulation speed.

In [4], a fast retargetable simulation technique improves traditional static compiled simulation by the utilization of the host machine resources.

FastSim [10] and Embra [12] simulators use dynamic binary translation and result caching to improve simulation performance. Embra provides a high performance but it is restricted to the simulation of only the MIPS R3000/R4000 [1] architectures.

The architecture description language LISA [9] is the basis for a retargetable compiled approach aiming at the generation of fast simulators for microprocessors even with complex pipeline structures. LISA is also used as entry language in the MaxCore framework [3] which automatically generates fast and accurate processor simulation models.

In the BUILDABONG framework ([11], [5], [6]), a cycle-accurate model of the register transfer architecture using the formalism of Abstract State Machines

(ASMs) is generated. Here, a convenient debugging environment is generated but the functional simulation is not optimized and can be performed only for relatively simple architectures. The problem of this approach is that the simulation engine is based on a library operating on bitstrings which is relatively slow.

This paper is based on our work presented in [7], where we proposed a mixed register-transfer level compiled simulation technique where the simulator is automatically extracted from a RTL description and the application program is compiled prior to simulator-run. In this paper we present a novel strategy in order to derive optimal simulators. Optimal in terms of a minimal number of comparisons, re-evaluations, and as secondary goal also the code size is minimized.

The rest of the paper is structured as follows: In Section 2, the basic concepts when generating high-speed optimized event-driven bit-true and cycle-accurate compiled simulators are described. Here, a given graphical architecture description is transformed into a graph representation. The main focus of the paper is on the *If-Then-Else trees extraction algorithm*. In Section 4, experiments and a case study when simulating a MIPS processor are presented and the results of our simulation approach are compared to an existing commercial simulator. Finally in Section 5, the paper is concluded with a summary of our achievements and an outlook of future work directions.

2 Efficient RTL Simulation

As our objective is a cycle-accurate and pipeline-accurate bit-true simulation of a given computer architecture, the architecture's behavior have to be simulated at the RTL level. Such a behavior may be described by a set of guarded register transfer patterns [8]. From a hardware-oriented point of view, register transfers (RT) take place during the execution of each machine cycle. During a RT, input values are read from a set of storage elements (registers, memory cells, etc.), a computation is performed on these values, and the result is assigned to a destination, which is again a set of storage elements.

Often, it can be seen that not all regions of the given circuit are involved in its entire functional process during any significant time interval. For example, in such architectures as FPGAs or coarse-grained

```

get_sensitivity_update_mappings_list( $G$ )
1  IN:     $G(V, E)$ 
2  OUT:  sensitivity-update-mappings list  $L$ 
3  BEGIN
4      integer  $i \leftarrow 0$ ;
5      FOR all sequential elements  $v \in V_r$  DO
6           $i \leftarrow i + 1$ ;  $v_{r_i} \leftarrow v$ ;
7           $\{u_{r_1}, \dots, u_{r_n}\} \leftarrow \text{DFS}(G, v)$ ;
8           $L_i \leftarrow (v_{r_i}, \{u_{r_1}, \dots, u_{r_n}\})$ ;
9      ENDFOR
10     RETURN  $L$ ;
11     END

```

Figure 1. Algorithm to extract a sensitivity-update-mappings list L from the RTL netlist given as a netgraph.

reconfigurable arrays some inactive cells do not perform any signal changes from cycle to cycle, especially those parts which are responsible for the reconfiguration process. That is why, it is very reasonable not to recompute or *update* these regions in each simulation cycle.

Definition 2.1 (Sensitivity-update-mapping) A sensitivity-update-mapping defines a set of sequential elements $U = \{u_1, \dots, u_n\}$ of the circuit which must be recomputed if a sequential element v_r has changed its value compared to the previous simulation cycle. It can be represented by the following notation: $(v_r \mapsto U)$ or $(v_r \mapsto \{u_1, \dots, u_n\})$.

A given RTL netlist can be seen as a hypergraph, where the elements and nets are vertices and edges, respectively. This hypergraph can be transformed into a graph by the introduction of a *netgraph* concept.

Definition 2.2 (Netgraph) A netgraph $G = (V, E)$, $E \subseteq V \times V$ is a directed graph containing two disjoint sets of vertices $V = V_r \cup V_c$, representing the sequential elements or registers V_r and combinational elements V_c of a given netlist $N = (V, F)$. Netlist interconnections are represented by directed edges $e = (v_1, v_2) \in E$.

From a given RTL circuit represented as a netgraph a list of *sensitivity-update-mappings* can be found according to the algorithm in Fig. 1. This algorithm

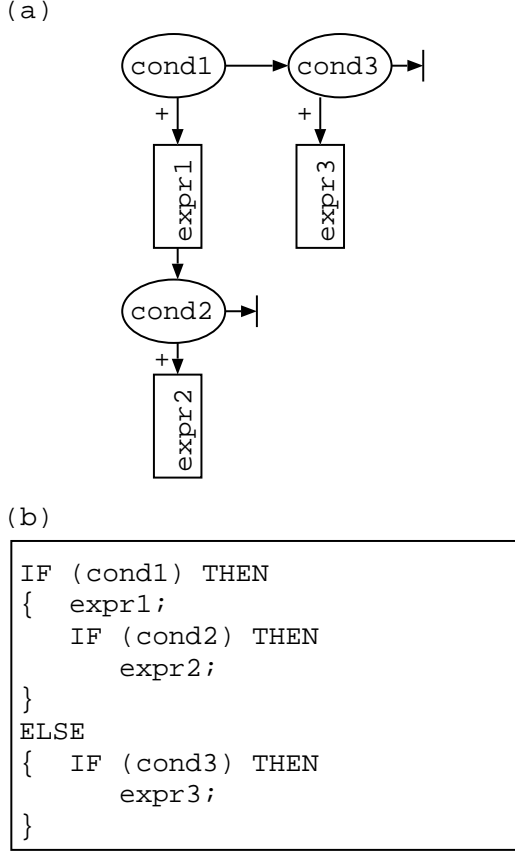


Figure 2. ITE tree (a) and appropriate IF-THEN-ELSE code (b).

determines the initial *sensitivity-update-mappings* as follows: if there exist a directed path from one register v_{r_1} to an other register v_{r_2} and on the path between these registers are no other sequential elements, then if the value of v_{r_1} changes, v_{r_2} has to be updated by evaluating the path of combinational elements in between. We can write: $L = \{\bigcup_i L_i\}$ with $L_i = (v_{r_i}, \{u_{r_1}, \dots, u_{r_n}\})$. A set of such initial sensitivity-update-mappings and evaluation paths can be achieved by a search algorithm like *depth-first search* (DFS).

In [7], we presented a graph decomposition algorithm. The algorithm transforms a graph representing a given RTL circuit into subgraphs, denoting the minimal subsets of sequential elements which have to be reevaluated during each simulation cycle. The conditions of the sensitivity-update routine in the generated simulator can be directly extracted from the set of subgraphs. Each subgraph corresponds to one IF-THEN

code structure with a complex condition structure to check the set of input registers and with expressions to update all possible output registers in the current subgraph. The whole sensitivity-update routine is a set of such independent IF-THEN structures. In spite of the decomposition algorithm guarantees that each register is updated at most once during each simulation cycle, it only minimizes the number of comparisons in the conditions using a heuristic. So, during one simulation cycle, some registers are still analyzed whether they changed with respect to the previous cycle more than once which leads to inefficient simulation. In this context, we present in the following a new method to generate efficient simulation code by representing the sensitivity-update routines by a set of sophisticated nested IF-THEN-ELSE structures. To extract such IF-THEN-ELSE code structures which enable efficient RTL simulation with the minimal number of register updates and register comparisons.

We represent an IF-THEN-ELSE code structure as a binary tree, consisting of two kinds of nodes: condition and expression nodes.

Definition 2.3 (ITE tree) An IF-THEN-ELSE tree (ITE tree) is a binary tree where every node is either of type condition or expression. Every condition node's left subtree represents the expressions of an appropriate IF-THEN-ELSE code structure when the condition is true, and every right subtree represents the expressions that must be evaluated if the condition is false. The root node is always a conditional node.

In Fig. 2 an example of an ITE tree and the appropriate IF-THEN-ELSE code is presented. The subset of expression nodes is shown as rectangles and the subset of conditional nodes is presented by ellipses. To generate an appropriate IF-THEN-ELSE code all of the nodes of the given ITE tree must be traversed, this can be done by using a DFS algorithm. The conditions read from the root condition nodes or from the condition nodes direct parent of which is an expression node, must be placed into the IF-part of the IF-THEN-ELSE code structure, and all of the other conditions are placed into the ELSE-part of the code.

To generate efficient IF-THEN-ELSE-code any path of the ITE tree must not contain the nodes with the same expressions. Such ITE tree we call *optimal*.

```

extract_ite_trees(G)
1  IN:    G(V, E)
2  OUT:   Set of If-Then-Else trees  $T_i$ 
3  BEGIN
4  //Find sensitivity-update-mappings list L
5    //list element  $L_i = (v_{r_i} \mapsto U_i), i = 1, \dots, N$ 
6     $L \leftarrow \text{get\_sensitivity\_update\_mappings\_list}(G)$ ;
7  //Break sensitivity-update-mappings list into groups  $S_i, i = 1, \dots, l$ 
8    // $L = S_1 \cup S_2 \cup \dots \cup S_l : s_1 \in S_i : \nexists s_2 \in S_j : s_2 = s_1, i \neq j$ ;
9     $S = \{S_1, \dots, S_l\} \leftarrow \text{decompose\_sensitivity\_update\_mappings\_list}(L)$ ;
10   integer  $i \leftarrow 0$ ;
11   FOR all sensitivity-update-mappings groups DO
12      $i \leftarrow i + 1$ ;
13     //sorting mappings in descending order (criteria is  $|U_j|$  for each mapping )
14      $\text{sort\_mappings\_in\_group}(S_i)$ ;
15     //Create an If-Then-Else Tree for each group independently
16      $T_i \leftarrow \text{group\_to\_tree}(S_i)$ ;
17   ENDFOR
18 END

```

Figure 3. If-Then-Else optimal trees extraction algorithm

In a technical implementation of a compiled RTL simulator some registers would have to be updated more than once during the same simulation cycle. Moreover, in order to determine a new event in the simulation system some registers must be compared with respect to their value in a previous simulation cycle more than once too. These facts make the overall simulation process inefficient. In order to avoid such multiple updates and comparisons, in the next section we present an optimal ITE trees extraction algorithm to enable high speed simulations.

3 If-Then-Else Trees Extraction Algorithm

Now, we propose a novel algorithm, we call *ITE trees extraction algorithm* whose purpose is, (i), to represent all RTs of a given RTL architecture as a set of optimal ITE trees which denote the minimal subsets of sequential elements which have to be recomputed during one simulation cycle and (ii), to define the minimal number of registers compare operations in conditions of the generated simulator code for RT updates. In Fig. 3, an algorithm to derive an optimal set of ITE trees T_i is given as pseudo-code. For the sake of clarity, in Fig. 1, Fig. 4, and Fig. 5 the additional algo-

rithms used by the main algorithm are presented.

The worst-case running time of the IF-THEN-ELSE trees extraction algorithm is $O(|V|^2)$. Since, this complexity influences only the generation of the simulator it can be tolerated. This algorithm is applied only during the generation process of the simulator and doesn't effect the simulation speed.

First, the algorithm performs a DFS¹ in order to extract the unique set of *sensitivity-update-mappings* such that, each left hand side of them has only one register (see the algorithm in Fig. 1). This set is placed into list L . Then, the list L is broken into independent groups by applying a decomposition algorithm as described in Fig. 4 such that any two groups do not contain the same mapping. The grouping of the list L allows to terminate same subtrees in all paths of the ITE tree, which, actually, doesn't effect the simulation speed, but significantly minimizes the size of the code of the generated simulator. The grouping also decreases the complexity of the algorithm. In the next phase of the ITE trees extraction algorithm the mappings are sorted in descending order. The criteria of sorting is the number of elements in the right-hand

¹The DFS is modified such that, if a sequential node v is found no outgoing edges of v traversed anymore.

decompose_sensitivity_update_mappings_list(L)

```
1  IN:    L
2  OUT:   Set of sensitivity-update-mapping groups S
3  BEGIN
4  //Build adjacency matrix  $M(|V_r| \times |V_r|)$ 
5      $m_{ij} \leftarrow \begin{cases} 1, & \text{if } \exists L_i : (v_{r_i} \mapsto u_{r_j}), \\ 0, & \text{otherwise;} \end{cases}$ 
6  //Sorting the rows of M in descending order (criteria is a sum of the elements in each row)
7      $sum\_in\_rows[1..|V_r|] \leftarrow \text{get\_sum\_in\_rows}(M)$ ;
8      $\text{sort\_rows}(M, sum\_in\_rows)$ ;
9  //Arrange the sensitivity-update-mappings in independent groups
10 List[] S;
11 integer grIdx  $\leftarrow -1$ ;
12 FOR i = 1 TO ( $|V_r| - 1$ ) DO
13     IF ( $\text{!row\_is\_marked}(i, M)$ ) THEN
14         grIdx  $\leftarrow grIdx + 1$ ;
15          $S[grIdx] \leftarrow \emptyset$ ;  $S[grIdx].add\_mapping(i)$ ;
16         FOR j = 1 TO ( $|V_r|$ ) DO
17             IF ( $M[i, j] == 1$ ) THEN
18                 FOR ii = i + 1 TO ( $|V_r|$ ) DO
19                     IF ( $\text{!row\_is\_marked}(ii, M)$ ) THEN
20                          $mark\_row(ii, M)$ ;  $S[grIdx].add\_mapping(ii)$ ;
21                     ENDIF
22                 ENDFOR
23             ENDIF
24         ENDFOR
25     ENDIF
26 ENDFOR
27 RETURN S;
28 END
```

Figure 4. Decompose the list of sensitivity-update-mappings algorithm

side of the mapping ($|U|$) (see Definition 2.1). And finally, all sorted sensitivity-update-mapping groups are sequentially converted to ITE trees. This is done in the `group_to_tree(S_i)` algorithm described in Fig. 5. The process of tree formation is done in such manner that in all subtrees of the tree the expression and condition nodes will be included only once. This fact provides an optimality of the algorithm, because during one simulation cycle only one branch of the ITE tree or only one branch of the IF-ELSE code can be executed. The kernel of the algorithm in Fig. 5 is the construction of left subtrees of the ITE tree (see lines 7-18). The conditional nodes are inserted into a subtree at line 11 by extracting the left-hand side of the sensitivity-

update-mapping. For the same subtree all conditional nodes are different as they are extracted from different sensitivity-update-mappings with different indices n (as we know, there are no mappings with the same left-hand part in a given list of sensitivity-update-mappings). In line 14 of the algorithm in Fig. 5 the expression node is formed by the subtraction of the set of the registers in right-hand side of the current mapping with the maximal set of registers in right-hand side (the very first mapping in each group). This allows to avoid multiple updates of the registers in one branch or subtree of the ITE tree. Hence, the converted ITE tree has a minimal number of conditions and update expressions, which shows its optimality.

group_to_tree(S_i)

```
1  IN:    sensitivity-update-mappings group  $S_i$ 
2  OUT:   If-Then-Else tree  $T_i$ 
3  BEGIN
4    Tree  $T_i \leftarrow \text{init\_tree}()$ ;
5     $T_i.\text{setRootNode}(S_i.\text{get\_mapping}(1).\text{getVr}())$ ;
6    FOR  $m = 1$  TO  $(|S_i| - 1)$  DO
7      Set  $\text{leftSubTree} \leftarrow \emptyset$ ;
8      IF  $(m \neq 1)$  THEN  $T_i.\text{appendRightChildNode}(S_i.\text{get\_mapping}(m).\text{getVr}())$ ;
9       $\text{leftSubTree.add}(S_i.\text{get\_mapping}(m).\text{getU}())$ ;
10     FOR  $n = (m + 1)$  TO  $|S_i|$  DO
11        $\text{cond} \leftarrow S_i.\text{get\_mapping}(n).\text{getVr}()$ ;
12       Set  $\text{tmpU}_1 \leftarrow S_i.\text{get\_mapping}(m).\text{getU}()$ ;
13       Set  $\text{tmpU}_2 \leftarrow S_i.\text{get\_mapping}(n).\text{getU}()$ ;
14       Set  $\text{expr} \leftarrow \text{tmpU}_2 \setminus \text{tmpU}_1$ ;
15       IF  $(\text{expr} \neq \emptyset)$  THEN  $\text{leftSubTree.add}(\text{cond}, \text{expr})$ ;
16     ENDFOR
17     Pointer  $\text{tmpTreePos} \leftarrow T_i.\text{get\_current\_pos}()$ ;
18      $T_i.\text{appendSubTree}(\text{leftSubTree})$ ;
19      $T_i.\text{set\_current\_pos}(\text{tmpTreePos})$ ;
20   ENDFOR
21   RETURN  $T_i$ ;
22 END
```

Figure 5. Convert sensitivity-update-mappings group into ITE tree algorithm

And therefore, the IF-THEN-ELSE code is generated according to deep-first traversing of the ITE trees and will perform the most possible efficient register update. The comparisons will be done only once for the registers that must be analyzed. The number of updates for each register will be at most one during one simulation cycle.

3.1 Example of ITE Trees Extraction

In the following, we will illustrate the main steps of the ITE trees extraction algorithm with a simple example. Let be a set of sensitivity-update-mappings given:

$$\begin{array}{ll} \{r_1\} \mapsto \{r_5, r_6, r_7, r_8\} & \{r_6\} \mapsto \{r_1, r_2\} \\ \{r_2\} \mapsto \{r_6, r_7\} & \{r_7\} \mapsto \{r_3\} \\ \{r_3\} \mapsto \{r_7, r_8\} & \{r_8\} \mapsto \{r_4\} \\ \{r_4\} \mapsto \{r_8\} & \{r_9\} \mapsto \{r_3, r_4\} \\ \{r_5\} \mapsto \{r_7\} & \end{array}$$

We can refer each sensitivity-update-mapping by the name of the register in the left-hand side

because in a left-hand side of a mapping there is only one register according to definition 2.1, and in the list of sensitivity-update-mappings there are no mappings with the same left-hand side. Then, we group the mappings from the given list of sensitivity-update-mappings by the `decompose_sensitivity_update_mappings_list(L)` algorithm. Basically, this algorithm unite in one group only those mapping whose sets of registers in a right-hand side are the subsets of the set of the registers in a right-hand side of another mapping in the same group. For instance, mapping r_2 in a right-hand side has the set of registers $\{r_6, r_7\}$ which is a subset of the set of registers in the right-hand side $\{r_5, r_6, r_7, r_8\}$ of the mapping r_1 . Thus, the mappings r_1 and r_2 are put in the same group by the algorithm. In such manner all other mappings are analyzed and put into the certain groups. As the result of applying the sensitivity-update-mappings list grouping algorithm described in Fig. 4 we obtain 3 groups:

Sensitivity-update-mappings list

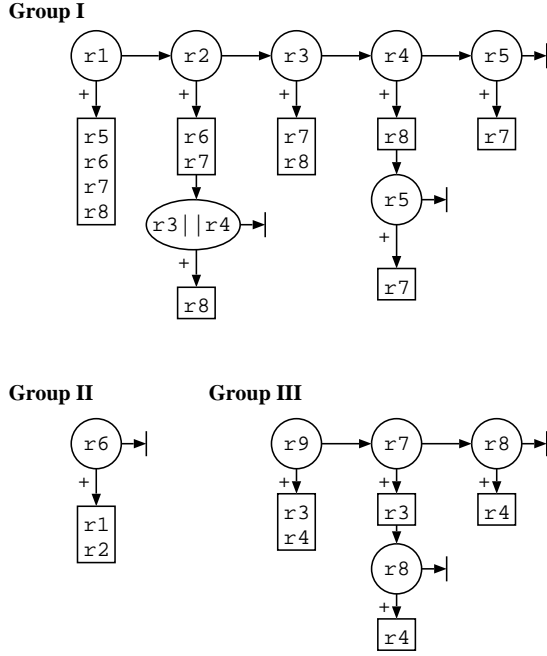
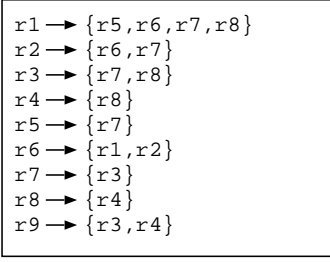


Figure 6. ITE trees extraction algorithm results.

- Group I: $(r_1, r_2, r_3, r_4, r_5)$
- Group II: (r_6)
- Group III: (r_7, r_8, r_9)

Sorting the sensitivity-update-mappings in each group with criteria of the number of registers in the right-hand side of sensitivity-update-mappings will reorder the mappings in the third group:

- Group I: $(r_1, r_2, r_3, r_4, r_5)$
- Group II: (r_6)
- Group III: (r_9, r_7, r_8)

In Group III, since mapping r_9 has two registers in

right-hand side and other mappings r_7, r_8 have there only one register, the mapping r_9 is put on the first position.

And finally, an algorithm reading a group of mappings and generating an appropriate ITE tree can be applied to obtain an ITE optimal tree for each group of sensitivity-update-mappings.

In the following, a part of the generated compiled simulator is shown for the ITE trees represented in Fig. 6.

```

IF (r1) THEN update(r5,r6,r7,r8);
ELSE
  IF (r2) THEN
    { update(r6,r7);
      IF (r3||r4) THEN update(r8);
    }
  ELSE
    IF (r3) THEN update(r7,r8);
    ELSE
      IF (r4) THEN
        { update(r8);
          IF (r5) THEN update(r7);
        }
      ELSE
        IF (r5) THEN update(r7);
IF (r6) THEN update(r1,r2);
IF (r9) THEN update(r3,r4);
ELSE
  IF (r7) THEN
    { update(r3);
      IF (r8) THEN update(r4);
    }
  ELSE
    IF (r8) THEN update(r4);

```

4 Experimental Results and Case Study

We performed a number of tests to measure the number of comparisons and the number of updates by using the proposed IF-THEN-ELSE trees extraction algorithm. Applying our new algorithm, leads to a reduction of comparisons up to 30% and the same optimal number of register updates compared with the results of the decomposition algorithm, presented in [7].

Table 1. Simulation results of the MIPS processor

MIPS R3000	SUN (GMP)	SUN (Integer)	SUN Scirocco (VHDL)	Linux (GMP)	Linux (Integer)
Simulation speed, CPS	253 807	3 448 280	9 940	540 541	9 090 920

As a realistic case study, a simulator for the MIPS R3000 32-bit processor has been generated and evaluated, too. Table 1 shows the simulation performance of the MIPS processor using our approach compared to the existing commercial RTL Scirocco simulator (in event-driven mode) (Synopsys) [2]. In the table, the performance results were obtained using (i) a Sun workstation running Solaris 2.9 with a 900 MHz UltraSPARC-III processor and (ii), a Pentium IV at 2.5 GHz with 1 GB RAM running Linux SuSe 8.0. A notable improvement in simulation speed of almost two magnitudes compared to Scirocco is shown. Up to 9 mill. machine cycles per second were demonstrated here at RTL which is equally fast or faster than most compiled instruction-level simulators.

5 Conclusions

In this paper we proposed a new approach for the derivation of optimal code in order to generate high-speed event-driven compiled simulators for processor architectures described on register transfer level. Namely, an efficient algorithm to generate optimal *IF-THEN-ELSE* structures in order to perform the update cycle during the event-driven simulation process is presented. The algorithm guarantees that during one simulation cycle a possible change of each register content is checked exactly once and that each register is updated at most once. The simulator's superior performance compared to an existing commercial simulator is shown. Finally, we demonstrate the pertinence of our approach by simulating a MIPS processor.

References

[1] Mips. <http://mips.com/>.

[2] Synopsys. <http://synopsys.com/>.

[3] Axys design automation.
<http://www.axysdesign.com>.

- [4] J. Z. Daniel. A retargetable, ultra-fast instruction set simulator. In *Proceedings on the European Design and Test Conference*, 1999.
- [5] D. Fischer, J. Teich, M. Thies, and R. Weper. Efficient architecture/compiler co-exploration for asips. In *ACM SIG Proceedings International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2002)*, pages 27–34, Grenoble, France, 2002.
- [6] D. Fischer, J. Teich, M. Thies, and R. Weper. BUILD-ABONG: A framework for architecture/compiler co-exploration for ASIPs. *Journal for Circuits, Systems, and Computers, Special Issue: Application Specific Hardware Design*, pages 353–375, 2003.
- [7] A. Kupriyanov, F. Hannig, and J. Teich. High-Speed Event-Driven RTL Compiled Simulation. In *Proceedings of the 4th International Samos Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS 2004)*, Island of Samos, Greece, July 2004.
- [8] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997.
- [9] S. Pees, A. Hoffmann, and H. Meyr. Retargeting of compiled simulators for digital signal processors using a machine description language. In *Proceedings Design Automation and Test in Europe (DATE'2000)*, Paris, March 2000.
- [10] E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memorization. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 283–294, 1998.
- [11] J. Teich, P. Kutter, and R. Weper. Description and simulation of microprocessor instruction sets using asms. In *International Workshop on Abstract State Machines*, Lecture Notes on Computer Science (LNCS), pages 266–286. Springer, 2000.
- [12] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.