Fast Dynamic Analysis of Complex HW/SW-Systems based on Abstract State Machine Models

Giuseppe Del Castillo
Heinz Nixdorf Institut
Department of Mathematics
& Computer Science
University of Paderborn
Fürstenallee 11, 33102 Paderborn, Germany

Wolfram Hardt
C-LAB: Cooperative Computing
& Communication Laboratory,
Siemens Nixdorf Informationssysteme AG
& University of Paderborn
Fürstenallee 11, 33102 Paderborn, Germany

Abstract

High level design decisions as HW/SW-partitioning and instrumenting of building blocks can be supported efficiently by detailed analysis of dynamic instruction usage. In many cases the instruction usage is specific to the application domain in view. We present a very fast analysis approach based on high level system models. Complex application characteristics, e.g., the average number of not interrupted instructions can be determined. This is much more than execution of, e.g., C-programs can provide.

1 Introduction

Today's systems are highly complex and composed out of several building blocks (BB), each block possibly consisting of a HW-part and a SW-part. Thus the complexity of a single BB is reasonably high (e.g., processor cores, microcontrollers, floating point and memory units). The performance of a BB is determined by the HW constituting this block and by the SW executed on this block. As each programmable BB has its own instruction set, the evaluation of performance trade-offs is not trivial. Furthermore, the instruction set available to the system programmer results from the instruction sets of all building blocks. For optimization of a HW/SW-BB as well as of a complete system, detailed evaluation of instruction usage is needed. This system level optimization does not focus in the implementation of a single instruction because this is subject of the HW-design phase of a BB. Most systems are optimized for a special application domain. System optimizations reflects the domain specific characteristics, e.g., deep pipelines are only efficient if large basic blocks are to be computed. To determine these domain specific characteristics fast dynamic analysis of the instruction mix of the SW-part of a single BB as well as of the complete HW/SW-system remains crucial. Based on this analysis data, important design decisions are made, e.g., concerning HW/SW-partitioning, and the complexity of BBs. Hardt and Rosenstiel have pointed out that detailed analysis data, e.g., on memory access and dynamic instruction usage are important for performance driven HW/SW-partitioning [9]. Of course, this cannot be determined statically, as the control structure is in general data-dependent. For evaluation co-simulation can be used. Several approaches to co-simulation have been proposed, e.g., [10, 3, 4]. One major problem is the design complexity of HW/SW-systems, which leads to very long simulation times. Other approaches provide executable HW/SW-implementations for dynamic analysis, e.g., [8]. This is much faster than simulation but the instrumentation of the executable implementation is very specialized to one single target system. In this paper, we propose a different approach for system level analysis of HW/SW-systems. An abstract model is provided for analysis of dynamic instruction usage. This model is based on *abstract state machines* (ASMs) and is theoretically well founded. The instrumentation of different instruction sets and also of several instructions with the same functionality but different execution times becomes very easy. The main features of this approach are easy determination of the dynamic instruction mix (e.g. the average length of not interrupted instructions or the length of instruction sequences without data dependencies) by very high simulation speed compared to traditional simulation approaches. This leads to a very valuable support for the evaluation of design decisions. Besides this, comfortable debugging features are provided by the ASM simulator.

In this paper, we present a short overview of the basic notions of ASMs, a first case study, and some experimental results to demonstrate the improvements of our approach.

2 Basic Concepts of Abstract State Machines

In this section we introduce the notions of ASMs needed in this paper, as implemented in the ASM-SL specification language [5] (the reader interested in a deeper study of ASMs should consult Gurevich's definition of ASMs in [7]). We first describe the computational model underlying ASMs, and then their syntax and semantics.

2.1 Computational Model

Computations Abstract state machines define a state-based computational model, where computations (*runs*) are finite or infinite sequences of states $\{S_i\}$, obtained from a given *initial state* S_0 by repeatedly executing *transitions*. Such runs can be intuitively visualized as

$$S_0 \xrightarrow{\delta_1} S_1 \xrightarrow{\delta_2} S_2 \ \ldots \ \xrightarrow{\delta_n} S_n \ \ldots$$

where the S_i are the states and the δ_i the transitions.

States The *states* are algebras over a given *signature* Σ (or Σ-*algebras* for short). A signature Σ consists of a set of *basic types* and a set of *function names*, each function name f coming with a fixed arity n and type $T_1 \ldots T_n \to T$, where the T_i and T are basic types (written $f : T_1 \ldots T_n \to T$, or simply $f : T$ if $n = 0$). A Σ-algebra (or state) S consists of: *(i)* a nonempty set $\mathcal{T}^S$ for each basic type T (the *carrier sets* of S), and *(ii)* a function

$$\mathbf{f}_S : \mathcal{T}_1^S \times \ldots \times \mathcal{T}_n^S \to \mathcal{T}^S$$

for each function name $f : T_1 \ldots T_n \to T$ in Σ (the *interpretation* of the function name f in S). Some function names in Σ are declared as *static* (indicating that they have the same interpretation in each computation state), while other are declared as *dynamic* (indicating that their interpretation may be altered by the transitions). Any signature Σ must contain a basic type *BOOL*, static nullary function names (constants) *true* : *BOOL*, *false* : *BOOL*, and the usual boolean operations ($\wedge$, $\vee$, etc.).[1] Finally, there is a special constant *undef* : T for any basic type T except *BOOL*. When no ambiguity arises we omit explicit mention of the state S (e.g., we write $\mathcal{T}$ instead of $\mathcal{T}^S$ for the carrier sets, and $\mathbf{f}$ instead of $\mathbf{f}_S$ for static functions, as they never change in the course of a computation).

Transitions Transitions transform a state S into its successor state S' by changing the interpretation of some dynamic function names on a finite number of points.

More precisely, the transition transforming S into S' results from firing a finite *update set* Δ at S, where the *updates* are of the form $(f, \overline{x}, y)$, where $f : T_1 \ldots T_n \to T$ is a dynamic function name, $\overline{x} \in \mathcal{T}_1 \times \ldots \times \mathcal{T}_n$, and $y \in \mathcal{T}$. The state S' resulting from firing Δ at S is such that the carrier sets are unchanged (i.e., $\mathcal{T}^{S'} = \mathcal{T}^S$ for each basic type T), and, for each function name f:

$$\mathbf{f}_{S'}(\overline{x}) \ = \ \begin{cases} y & \text{if } (f, \overline{x}, y) \in \Delta \\ \mathbf{f}_S(\overline{x}) & \text{otherwise.} \end{cases}$$

[1] A function $f : T_1 \ldots T_n \to$ *BOOL* is also called a *relation* over $T_1 \ldots T_n$.

The update set Δ—which depends on the state S—is determined by evaluating in S a distinguished *transition rule* P, called the *program*.[2] Note that the above definition is only applicable if Δ does not contain any two updates $(f, \overline{x}, y)$ and $(f, \overline{x}, y')$ with $y \neq y'$ (i.e., if Δ is *consistent*).

2.2 Language

Terms *Terms* (over the given signature Σ) are used to refer to elements of the carrier sets (the admissible *values*), and usually denoted by the letter t. Each term t has a type T (written $t : T$). The syntax of terms is defined recursively: if $f : T_1 \ldots T_n \to T$ is a function name in Σ, and t_i is a term of type T_i (for $i = 1, \ldots, n$), then $f(t_1, \ldots, t_n)$ is a term of type T.[3] The meaning of a term t (of type T) in a state S is a value $S(t) \in \mathcal{T}$ defined by

$$S(f(t_1, \ldots, t_n)) \ = \ \mathbf{f}_S(S(t_1), \ldots, S(t_n)).$$

Transition rules While terms denote values, transition rules (*rules* for short) denote *update sets*, and are used to define the dynamic behaviour of an ASM: the meaning of a rule R in a state S is an update set $\Delta_S(R)$.

The *program* P is a distinguished rule which determines the ASM runs: each state S_{i+1} ($i \geq 0$) is obtained by firing the update set $\Delta_{S_i}(P)$ at S_i. Visually:

$$S_0 \xrightarrow{\Delta_{S_0}(P)} S_1 \xrightarrow{\Delta_{S_1}(P)} S_2 \ \ldots \ \xrightarrow{\Delta_{S_{n-1}}(P)} S_n \ \ldots$$

The syntax and semantics of rules is as follows.

Update rule The *update* rule has the syntax

$$R \ ::= \ f(t_1, \ldots, t_n) := t$$

where $f : T_1 \ldots T_n \to T$ is a dynamic function name in Σ, $t_i : T_i$ for $i = 1, \ldots, n$, and $t : T$. Such an update rule produces a single update:

$$\Delta_S(\,R\,) \ = \ \{\, (f, (S(t_1), \ldots, S(t_n)), S(t)) \,\}.$$

Intuitively, the terms t_i and t are evaluated—in the state S—to values $x_i = S(t_i)$, $y = S(t)$; then, the interpretation of f on $(x_1, \ldots, x_n)$ is changed to y.

Block rule The *block* rule

$$R \ ::= \ R_1 \ \ldots \ R_n$$

combines the effects of more transition rules:

$$\Delta_S(\,R\,) = \ \textstyle\bigcup_{i=1}^{n} \Delta_S(\,R_i\,).$$

The execution of a block rule corresponds to *simultaneous* execution of its subrules.[4]

[2] In this way, abstract state machines—which can be considered, in a first approximation, as given by the program P together with an initial state S_0—model discrete dynamic systems.

[3] If $n = 0$ the parentheses are omitted, i.e. we write f instead of $f()$.

[4] For example, a block rule a := b, b := a exchanges a and b. Note also that the use of block rules may lead to inconsistent update sets.

Conditional rule The *conditional* rule has the syntax

$$R ::= \texttt{if}\ G\ \texttt{then}\ R_T\ \texttt{else}\ R_F$$

where G is a boolean term. Its meaning is, obviously:

$$\Delta_S(\,R\,) \;=\; \begin{cases} \Delta_S(\,R_T\,) & \text{if } S(G) = \textbf{true} \\ \Delta_S(\,R_F\,) & \text{otherwise.} \end{cases}$$

The ASM-SL Environment

The basic ASM constructions described above are part of the ASM-SL specification language [5], which also contains features for defining types, functions, and transitions, and a set of predefined types (booleans, integers, etc.) and generic data structures (tuples, lists, sets, etc.), which help to model a wide range of systems in a concise way. ASM-SL is the basis of a tool environment, called "The ASM Workbench", developed at Paderborn University by the first author, which supports syntax- and type-checking of ASM specifications as well as their simulation and debugging. The ASM model of the case study discussed in this paper has been checked and executed using that tool.

3 Case Study: a VLIW Instruction Set

In this case study an instruction set of a VLIW processor based on ASMs is presented. This instruction set is used as a kind of "abstract assembler code" for the zCPU, a VLIW processor used as control unit in the SIMD parallel architecture APE100 developed at INFN[5] [1]. The zCPU processor itself has already been modelled at the RT-level by means of ASMs in [2]. In the APE100 architecture, the abstract assembler code is mapped to executable VLIW code for the zCPU by a code generator. However, the instruction set under study could be implemented also by a traditional pipelined architecture, or by a superscalar processor, or whatever: in this case study we want to abstract from particular implementations, and concentrate instead on the study of the application domain specific usage of instructions, by means of an ASM-based methodology which can be easily generalized to other instruction sets.

The instruction format The instruction set under study—as most RISC instruction sets—consists of register-register arithmetic-logical instructions, load and store instructions for memory access, and branch instructions: we distinguish between two groups of instructions, the arithmetic-logical ones (MAC instructions for short) and all the other (IOC instructions). This groups can also be understood as different building blocks providing specialized instructions.

The instruction set is modelled by a data type `INSTR`, and auxiliary data types `JUMP_COND`, representing the possible branch conditions to be tested[6], `REG`, for register addresses, `DISP`, for the displacement field in load/store instructions.

[5] The Italian National Institute for Nuclear Physics.

[6] Note that we abstract from the concrete representation of such conditions by using symbolic names (constructors) with obvious meanings.

```
datatype JUMP_COND ==
  { TRUE, FALSE, EQ, NE, LT, LE, GT, GE }
datatype REG  == { R : INT }
datatype DISP == { Disp : INT }
datatype INSTR ==
{ // arithmetic-logic instructions (MAC)
  OR  : REG REG REG,  AND : REG REG REG,
  // ... all MAC instructions have this format
  // ... except ZERO, FF (1 op) and CMP (2 ops)
  ZERO : REG,  FF : REG,  CMP : REG REG,
  // I/O, branch and special instructions (IOC)
  LD   : REG REG DISP,
  // ... LDA, LDPA and ST have the same format
  JUMP : REG DISP JUMP_COND, HALT }
```

Resources The resources needed for the execution of instructions are: the program memory[7] (`instr` in the ASM model) and the program counter (`PMA`—Program Memory Address), the data memory (`mem`), the general-purpose registers (`reg`), the condition code flags (`Neg`, `Zero`, `Divz`):

```
static function instr : INT -> INSTR ==
  // (the program)
dynamic function PMA  : INT
  initially 0
dynamic function mem : INT -> INT
  initially    // (initial memory configuration)
dynamic function reg : REG -> INT
  initially { }  // (registers initially undef.)
dynamic function Neg   initially false
dynamic function Zero  initially false
dynamic function Divz  initially false
```

Arithmetic-logical instructions (MAC) The arithmetic-logical instructions are processed by a transition rule `MATH_RULE` which performs a case distinction and, according to the current instruction, fires the particular transition rule needed to execute that instruction:

```
transition MATH_RULE ==
  case instr (PMA) of
    AND (RR, R1, R2)   : DO_AND (RR, R1, R2) ;
    OR (RR, R1, R2)    : DO_OR (RR, R1, R2) ;
    // ... the same for other MAC instructions
  end
```

Each instruction is modelled by a transition rule, e.g.:[8]

```
transition DO_OR (RR, R1, R2) ==
  LogicalInstr (RR, or_fun (reg (R1), reg (R2)))
transition DO_AND (RR, R1, R2) ==
  LogicalInstr (RR, and_fun (reg (R1), reg (R2)))
```

Note the use of the static functions `or_fun`, `and_fun`, etc.: there is one such static function for each available arithmetic operation, such that the functional aspects of the specification are separated from the operational behaviour (state transitions), described for instance—in the case of logical operations like `AND`, `OR`—by the common rule:

```
transition LogicalInstr (RR, value) == block
  reg (RR) := value
  Neg      := (value < 0)
  Zero     := (value = 0)
end
```

[7] In this example, two separate memories are used for the program and for the data, such that instructions and data can be accessed simultaneously.

[8] We show only two examples, rules for other instructions are similar.

Load, store and branch instructions (IOC) Similarly as for MAC instructions, the main rule is a case distinction:

```
transition IOC_RULE ==
  case instr (PMA) of
    LD (RD, RA, disp) : DO_LD (RD, RA, disp) ;
    ST (RD, RA, disp) : DO_ST (RD, RA, disp) ;
    // ...
```

Modelling load/store instructions is very simple, for instance, the rules for the instructions `LD`, `ST` are:

```
transition DO_LD (RD, RA, disp) ==
  reg (RD) := mem (reg (RA) + disp_addr (disp))
transition DO_ST (RD, RA, disp) ==
  mem (reg(RA) + disp_addr (disp)) := reg (RD)
```

where the static function `disp_addr` extracts the address contained in the displacement field of the instruction.

Modelling branches is slightly more complicated, as it implies testing the branch condition against the flags:

```
static function eval_cond (cond, N, Z) ==
  case cond of
    TRUE  : true ;        FALSE : false ;
    EQ    : Z = true ;    NE    : Z = false ;
    LE    : N = true or Z = true ;   // ...
  end
transition DO_JUMP (RA, disp, cond) ==
  if eval_cond (cond, Neg, Zero)
  then PMA := reg (RA) + disp_addr (disp)
  else PMA := PMA + 1 end
```

The program counter (PMA) Finally, a rule is needed to increment the program counter in normal situations, i.e. when the current instruction is not a branch:

```
transition INCR_PMA ==
  if not (is_jump_instruction (instr (PMA)))
     and (instr (PMA) <> HALT)
  then PMA := PMA + 1 end
```

(where `is_jump_instruction` is a static function returning true whenever its argument is a `JUMP` instruction).

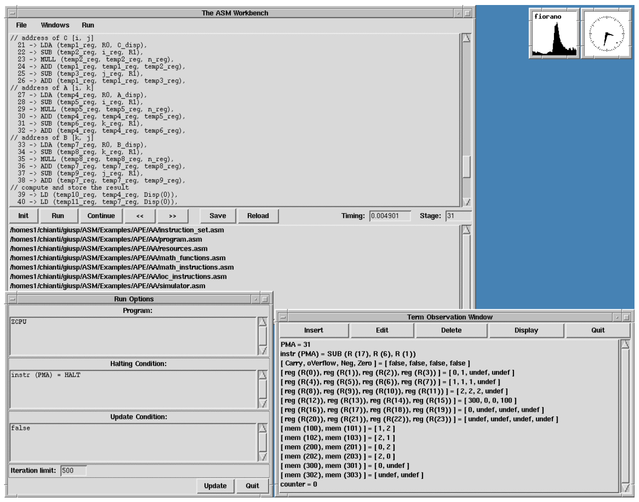
The instruction set model The executable model of the instruction set is then simply obtained by putting all the pieces together into the following rule (the ASM *program*):

```
transition ZCPU ==
  block  MATH_RULE  IOC_RULE  INCR_PMA  end
```

The whole executable model of the instruction set is quite compact (ca. 450 lines of ASM-SL code) and was obtained very quickly from an existing specification on paper[9].

4 Simulation and Experimental Results

The ASM model of the instruction set has been tested using *The ASM Workbench*[10] on a simple test program which



```
// address of C [i, j]
  21 -> LDA (temp1_reg, R0, C_disp),
  22 -> SUB (temp2_reg, i_reg, R1),
  23 -> MULL (temp2_reg, temp2_reg, n_reg),
  24 -> ADD (temp1_reg, temp1_reg, temp2_reg),
  25 -> SUB (temp3_reg, j_reg, R1),
  26 -> ADD (temp1_reg, temp1_reg, temp3_reg),
// address of A [i, k]
  27 -> LDA (temp4_reg, R0, A_disp),
  28 -> SUB (temp5_reg, i_reg, R1),
  29 -> MULL (temp5_reg, temp5_reg, n_reg),
  30 -> ADD (temp4_reg, temp4_reg, temp5_reg),
  31 -> SUB (temp6_reg, k_reg, R1),
  32 -> ADD (temp4_reg, temp4_reg, temp6_reg),
// address of B [k, j]
  33 -> LDA (temp7_reg, R0, B_disp),
  34 -> SUB (temp8_reg, k_reg, R1),
  35 -> MULL (temp8_reg, temp8_reg, n_reg),
  36 -> ADD (temp7_reg, temp7_reg, temp8_reg),
  37 -> SUB (temp9_reg, j_reg, R1),
  38 -> ADD (temp7_reg, temp7_reg, temp9_reg),
// compute and store the result
  39 -> LD (temp10_reg, temp4_reg, Disp(0)),
  40 -> LD (temp11_reg, temp7_reg, Disp(0)),
```

Timing: 0.004901 Stage: 31

```
/homes1/chianti/giusp/ASM/Examples/APE/AA/instruction_set.asm
/homes1/chianti/giusp/ASM/Examples/APE/AA/program.asm
/homes1/chianti/giusp/ASM/Examples/APE/AA/resources.asm
/homes1/chianti/giusp/ASM/Examples/APE/AA/math_functions.asm
/homes1/chianti/giusp/ASM/Examples/APE/AA/math_instructions.asm
/homes1/chianti/giusp/ASM/Examples/APE/AA/ioc_instructions.asm
/homes1/chianti/giusp/ASM/Examples/APE/AA/simulator.asm
```

Program: ZCPU

Halting Condition: instr (PMA) = HALT

Update Condition: false

Iteration limit: 500

```
PMA = 31
instr (PMA) = SUB (R (17), R (6), R (1))
[ Carry, oVerflow, Neg, Zero ] = [ false, false, false, false ]
[ reg (R(0)), reg (R(1)), reg (R(2)), reg (R(3)) ] = [ 0, 1, undef, undef ]
[ reg (R(4)), reg (R(5)), reg (R(6)), reg (R(7)) ] = [ 1, 1, 1, undef ]
[ reg (R(8)), reg (R(9)), reg (R(10)), reg (R(11)) ] = [ 2, 2, 2, undef ]
[ reg (R(12)), reg (R(13)), reg (R(14)), reg (R(15)) ] = [ 300, 0, 0, 100 ]
[ reg (R(16)), reg (R(17)), reg (R(18)), reg (R(19)) ] = [ 0, undef, undef, undef ]
[ reg (R(20)), reg (R(21)), reg (R(22)), reg (R(23)) ] = [ undef, undef, undef, undef ]
[ mem (100), mem (101) ] = [ 1, 2 ]
[ mem (102), mem (103) ] = [ 2, 1 ]
[ mem (200), mem (201) ] = [ 0, 2 ]
[ mem (202), mem (203) ] = [ 2, 0 ]
[ mem (300), mem (301) ] = [ 0, undef ]
[ mem (302), mem (303) ] = [ undef, undef ]
counter = 0
```

Figure 1. The ASM Workbench

multiplies matrices of arbitrary size, hand-compiled from a high-level algorithm through a simple compilation scheme.

Instrumenting the model

In order to collect any information of interest about instruction usage in the course of the simulation, we had to "instrument" the instruction set model: this can be done easily, as ASMs are a general-purpose specification and modelling language, and not an hardware description language. In fact, one simply extends the model by the necessary functions and transitions which do the bookkeeping, which can be freely mixed with those constituting the actual system model. For instance, if we want to know the number of jump instructions that our test program goes through during its execution, we just introduce a dynamic function `counter` and a transition `COUNT_JUMPS`:

```
dynamic function counter initially 0
transition COUNT_JUMPS ==
  if (is_jump_instruction (instr (PMA)))
  then counter := counter + 1 end
```

and then include `COUNT_JUMPS` in the block constituting the ASM program to make the counting effective:

```
transition ZCPU == block
  MATH_RULE  IOC_RULE  INCR_PMA   // system model
  COUNT_JUMPS                     // bookkeeping
end
```

From this example it should result clear how any other interesting measurements on instruction usage can be defined and incorporated into the simulation.

Instruction usage measurements

The results collected by simulating the execution of the matrix multiplication program, for different sizes of the matrices A and B (A of size $m \times p$, B of size $p \times n$, written $m \times p \times n$ for short), are presented in the following table. As

[9] About three days of work, including developing from scratch the example program used for the experiments.

[10] A snapshot of the tool is shown in Figure 1, where a fragment of the test program is visible in the browser window (the upper left window) and some observable quantities of interest (like the program counter `PMA` and the contents of some relevant registers and memory locations) can be seen in the *term observation window* (bottom right). The other visible window (*run options*) contains simulation status information, e.g. the program to be executed, in this case consisting of the rule `ZCPU`, and an halting condition.

expected, the proportions of the executed instructions relative to different instruction groups are quite stable, except for branches, which become less influent as the sizes grow.

	$2 \times 2 \times 2$		$2 \times 3 \times 2$		$3 \times 3 \times 3$		$5 \times 5 \times 5$	
No. of Inst.	abs	rel	abs	rel	abs	rel	abs	rel
Logical	1	0.3	1	0.2	1	0.1	1	0.0
Additive	155	51.3	215	51.9	478	52.5	2066	53.1
Multipl.	36	11.9	52	12.6	117	12.8	525	13.5
Division	0	0.0	0	0.0	0	0.0	0	0.0
MAC instr.	192	63.5	268	4.7	596	65.4	2592	66.6
Load	63	20.9	87	21.0	188	20.6	810	20.8
Store	12	4.0	16	3.9	36	4.0	150	3.8
Branch	35	11.6	43	10.4	91	10.0	341	8.8
IOC instr.	110	36.5	146	5.3	315	34.6	1301	33.4
Total	302	100%	414	100%	911	100%	3893	100%

Probably more interesting is a refined dynamic analysis of the branch behaviour: here we separate taken branches from not taken ones, and introduce another interesting measure, namely the average number of instructions executed between two taken branches, including of course not taken branches (called *average distance* in the table):[11]

	$2\times2\times2$		$2\times3\times2$		$3\times2\times3$		$3\times3\times3$	
No. of Inst.	abs	rel	abs	rel	abs	rel	abs	rel
Taken	21	60.0	25	58.1	43	58.9	52	57.1
Not taken	14	40.0	18	41.9	30	40.1	39	42.9
Total	35	100%	43	100%	73	100%	91	100%
Average distance	13.38		15.56		14.33		16.52	

The comparison between the cases $2 \times 3 \times 2$ and $3 \times 2 \times 3$ is particularly interesting: although the matrices to be multiplied have the same size, in the latter case the average branch distance is noticeably worse: this is due to the fact that the innermost loop (the shortest one) is iterated more times. This is a typical case where the efficiency of the code could be improved by unfolding the loop.

What we want to suggest here is that the proposed technique (simulation of abstract executable models), possibly in combination with a prototype code generator[12], can be used to comparatively test the effectiveness of code optimization techniques (on the compiler side) as well as of the instruction set (on the architecture side).

5 Conclusions

In this paper we presented a novel approach to high-level analysis based on abstract state machines, a formal method with a rigorous mathematical semantics (but still easy to understand and to use for practitioners without a particular training in formal methods). A case study demonstrated the fast and easy determination of dynamic instruction usage information which is important for architecture design of HW/SW-systems. The presented analysis approach provides much more information than, e.g., the execution of a C-program which is derived from the ASM model basis.

Further work will concentrate on the adaption of design space exploration. Moreover, ASMs could be used for verification (see for instance [6], where the theorem prover KIV is used to verify the correctness of an ASM-based model of the DLX architecture).

[11] These measurements give, already at the level of abstraction of the instruction set, some hint about the performance of the programs when executed on a pipelined implementation of the instruction set (of course more taken branches and shorter average branch distance imply a worse pipeline performance).

[12] Which must be dedicated to the entire target system.

References

[1] A. Bartoloni et al. *The Software of the APE100 Processor* and *A Hardware Implementation of the APE100 Architecture*. International Journal of Modern Physics, C 4 (1993), p. 955.

[2] E. Börger, G. Del Castillo. *A formal method for provably correct composition of a real-life processor out of basic components*. In: B. Werner (Ed.), Proc. of ICECCS'95, Ft. Lauderdale, Florida, pp. 145-148.

[3] J. Buck, S. Ha, A. Lee, D.G. Messerschmidt. *Ptolemy: a Framework for Simulation and Prototyping Heterogeneous Systems*. International Journal of Computer Simulation, Special issue on Simulation Software Development, Jan. 1994.

[4] R. Camposano, J. Wilberg. *Embedded System Design*. Design Automation for Embedded Systems, 1995, vol. 1, no. 1, pp. 5-50.

[5] G. Del Castillo. *ASM-SL, a Specification Language based on Gurevich's Abstract State Machines: Introduction and Tutorial*. Universität-GH Paderborn, technical report, to appear.

[6] M. Giese, D. Kempe, A. Schönegge. *KIV zur Verifikation von ASM-Spezifikationen am Beispiel der DLX-Pipelining Architektur*. Universität Karlsruhe, Interner Bericht Nr. 16/97, 1997.

[7] Y. Gurevich. *Evolving Algebras 1993: Lipari Guide*. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[8] W. Hardt, W. Rosenstiel. *Speed-Up Estimation for HW/SW-Systems*. Proc. of CODES/CACHE '96.

[9] W. Hardt, W. Rosenstiel. *Prototyping of Tightly Coupled Hardware/Software-Systems*. Design Automation for Embedded Systems, vol. 2, no. 1 (1997).

[10] J. Henkel, Th. Benner, R. Ernst, W. Ye, N. Serafimov, G. Glawe. *COSYMA: A Software–Oriented Approach to Hardware/Software Codesign*. Journal of Computer and Software Engineering, 1994, vol. 2, no. 3.

More details about the techniques and models described in this paper can be found at:
`http://www.uni-paderborn.de/cs/giusp.html`