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

Abstract

High level design decisions as HW/SW-partitioning and
instrumenting of building blocks can be supported effi-
ciently by detailed analysis of dynamic instruction usage.
In many cases the instruction usage is specific to the ap-
plication domain in view. We present a very fast analysis
approach based on high level system models. Complex ap-
plication 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 con-
sisting 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 per-
formance of a BB is determined by the HW constituting this
block and by the SW executed on this block. As each pro-
grammable 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 optimi-
ization of a HW/SW-BB as well as of a complete system,
detailed evaluation of instruction usage is needed. This sys-
tem level optimization does not focus in the implementa-
tion of a single instruction because this is subject of the HW-
design phase of a BB. Most systems are optimized for a spe-
cial 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 de-
terminate 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 cru-
cial. Based on this analysis data, important design deci-
sions 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 complex-
ity of HW/SW-systems, which leads to very long simula-
tion 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 differ-
ent 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 de-
termination of the dynamic instruction mix (e.g. the aver-
age 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 specifica-
tion 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 \(\delta_i\) the transitions.

States The states are algebras over a given signature \(\Sigma\) (or \(\Sigma\)-algebras for short). A signature \(\Sigma\) consists of a set of basic types and a set of function names, each function name \(f\) coming with a fixed arity and type \(T_1, \ldots, T_n \rightarrow T\), where the \(T_i\) and \(T\) are basic types (written \(f : T_1 \times \ldots \times T_n \rightarrow T\), or simply \(f : T\) if \(n = 0\)). A \(\Sigma\)-algebra (or state) \(S\) consists of:

(i) a nonempty set \(T^S\) for each basic type \(T\) (the carrier sets of \(S\)), and

(ii) a function \(f_S : T_1^S \times \ldots \times T_n^S \rightarrow T^S\)

for each function name \(f : T_1 \ldots T_n \rightarrow T\) in \(\Sigma\) (the interpretation of the function name \(f\) in \(S\)). Some function names in \(\Sigma\) 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 \(\Sigma\) must contain a basic type \(BOOL\), static nullary function names (constants) \texttt{true : BOOL, false : BOOL}, and the usual boolean operations (\&\&, ||, etc.).

Finally, there is a special constant \texttt{undef : T} for any basic type \(T\) except \(BOOL\). When no ambiguity arises we write \(T\) instead of \(T^S\) for the carrier sets, and \(f\) instead of \(f_S\) 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 \(\Delta\) at \(S\), where the updates are of the form \((f, \overline{x}, y)\), where \(f : T_1 \ldots T_n \rightarrow T\) is a dynamic function name, \(\overline{x} \in T_1 \times \ldots \times T_n\), and \(y \in T\). The state \(S'\) resulting from firing \(\Delta\) at \(S\) is such that the carrier sets are unchanged (i.e., \(T^S' = T^S\) for each basic type \(T\)), and, for each function name \(f\):

\[ f_S' (\overline{x}) = \begin{cases} y & \text{if } (f, \overline{x}, y) \in \Delta \\ f_S (\overline{x}) & \text{otherwise.} \end{cases} \]

The update set \(\Delta\)—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 \(\Delta\) does not contain any two updates \((f, \overline{x}, y)\) and \((f, \overline{x}, y')\) with \(y \neq y'\) (i.e., if \(\Delta\) is consistent).

2.2 Language

Terms Terms (over the given signature \(\Sigma\)) 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 \(T : T_1 \times \ldots \times T_n \rightarrow T\) is a function name in \(\Sigma\), and \(t_i\) is a term of type \(T_i\) (for \(i = 1, \ldots, n\)), then \((t_1, \ldots, t_n)\) is a term of type \(T\).\(^3\)
- The meaning of a term \(t\) (of type \(T\)) in a state \(S\) is a value \(S(t) \in T\) defined by

\[ S(f(t_1, \ldots, t_n)) = f_S(S(t_1), \ldots, S(t_n)). \]

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 \(\Delta S (P)\) 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 \(f : T_1 \ldots T_n \rightarrow T\) is a dynamic function name in \(\Sigma\), \(t_i : T_i\) 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:

\[ \Delta S (R) = \bigcup_{i=1}^{n} \Delta S (R_i) \]

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 \(S_0\)—mode 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

\[ R := \text{if } G \text{ then } R_F \text{ else } R_T \]

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

\[ \Delta S(R) = \begin{cases} \Delta S(R_T) & \text{if } S(G) = \text{true} \\ \Delta S(R_F) & \text{otherwise.} \end{cases} \]

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 [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 abstact 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, REG, for register addresses, DISP, for the displacement field in load/store instructions.

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.:

```
transition DO_OR (RR, R1, R2) ==
  LogicalInstr (RR, or_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);
    end
```

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 + lend
```

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
```

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.²

## 4 Simulation and Experimental Results

The ASM model of the instruction set has been tested using The ASM Workbench¹⁰ on a simple test program which 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 x p, B of size p x n, written m x p x n for short), are presented in the following table. As
expected, the proportions of the executed instructions relative to different instruction groups are quite stable, except for branches, which become less influential as the sizes grow.

<table>
<thead>
<tr>
<th>No. of Inst.</th>
<th>$2 \times 2 \times 2$</th>
<th>$2 \times 3 \times 2$</th>
<th>$3 \times 2 \times 3$</th>
<th>$3 \times 3 \times 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>abs rel</td>
<td>abs rel</td>
<td>abs rel</td>
<td>abs rel</td>
</tr>
<tr>
<td>Logical</td>
<td>1 0.3</td>
<td>1 0.2</td>
<td>1 0.1</td>
<td>1 0.0</td>
</tr>
<tr>
<td>Additive</td>
<td>155 51.3</td>
<td>215 51.9</td>
<td>478 52.5</td>
<td>2066 53.1</td>
</tr>
<tr>
<td>Multiplic.</td>
<td>36 11.9</td>
<td>52 12.6</td>
<td>117 12.8</td>
<td>525 13.5</td>
</tr>
<tr>
<td>Division</td>
<td>0 0.0</td>
<td>0 0.0</td>
<td>0 0.0</td>
<td>0 0.0</td>
</tr>
<tr>
<td>MAC instr.</td>
<td>192 63.5</td>
<td>268 47</td>
<td>596 65.4</td>
<td>2592 66.6</td>
</tr>
<tr>
<td>Load</td>
<td>63 20.9</td>
<td>87 21.0</td>
<td>188 20.6</td>
<td>810 20.8</td>
</tr>
<tr>
<td>Store</td>
<td>12 4.0</td>
<td>16 3.9</td>
<td>36 4.0</td>
<td>159 3.8</td>
</tr>
<tr>
<td>Branch</td>
<td>35 11.6</td>
<td>43 10.4</td>
<td>91 10.0</td>
<td>341 8.8</td>
</tr>
<tr>
<td>IOC instr.</td>
<td>110 36.5</td>
<td>146 5.3</td>
<td>315 34.6</td>
<td>1301 33.4</td>
</tr>
<tr>
<td>Total</td>
<td>302 100%</td>
<td>414 100%</td>
<td>911 100%</td>
<td>3893 100%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>No. of Inst.</th>
<th>$2 \times 2 \times 2$</th>
<th>$2 \times 3 \times 2$</th>
<th>$3 \times 2 \times 3$</th>
<th>$3 \times 3 \times 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>abs rel</td>
<td>abs rel</td>
<td>abs rel</td>
<td>abs rel</td>
</tr>
<tr>
<td>Taken</td>
<td>21 60.0</td>
<td>28 58.1</td>
<td>43 58.9</td>
<td>52 57.1</td>
</tr>
<tr>
<td>Not taken</td>
<td>14 40.0</td>
<td>18 41.9</td>
<td>30 40.1</td>
<td>39 42.9</td>
</tr>
<tr>
<td>Total</td>
<td>35 100%</td>
<td>43 100%</td>
<td>73 100%</td>
<td>91 100%</td>
</tr>
<tr>
<td>Average distance</td>
<td>13.38</td>
<td>15.56</td>
<td>14.33</td>
<td>16.52</td>
</tr>
</tbody>
</table>

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 generator12, 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).

### References


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