Using the CASM Language for Simulator Synthesis and Model Verification [*]

Roland Lezuo
Vienna University of Technology
Institute of Computer Languages (E185)
Argentinierstraße 8
1040 Vienna, Austria
rlezuo@complang.tuwien.ac.at

Andreas Krall
Vienna University of Technology
Institute of Computer Languages (E185)
Argentinierstraße 8
1040 Vienna, Austria
andi@complang.tuwien.ac.at

ABSTRACT

We present the CASM language, an abstract state machine (ASM) based modeling language originally designed for verifying compiler backends. We demonstrate the expressiveness by describing an instruction set simulator (ISS) for MIPS in approximately 700 lines of code. Further we present a refinement of the models to cycle-accurately describe two implementations of the classic 5-stage MIPS pipeline. Utilizing symbolic execution allows us to prove semantic equivalence of the pipeline implementations and the instruction set description. Finally we compile the models to C++ and provide a small runtime to create a system simulator achieving a performance of approx. 1 MHz in MiBench and SPECInt2000 benchmarks.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Specification language*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model verification*

Keywords

ASM, Simulator Synthesis, Correctness Proofs

1. INTRODUCTION

Simulators are commonly used when developing embedded applications. Instruction Set Simulation (ISS) is among other things useful for debugging. Embedded applications are often difficult to debug in their execution environment. Running the application in a simulator may simplify producing error conditions and watching the program behavior. ISS are tuned for performance, simulating only what is needed for correct program semantics [7]. They often lack accuracy needed for detailed performance analysis [24]. Cycle-accurate simulation also simulates the behavior of pipelines and caches to achieve a much better estimation of the application's timing on real hardware. Due to the increased simulation effort, performance of cycle-accurate simulation is usually much lower. To execute complex embedded applications an environment has to be provided as well. Examples are file systems, interrupt controllers and other peripheral components. The resulting simulator is called a system simulator.

For many applications quick exploration of the design space is an important feature. How do changes in a processor's pipeline affect performance? Do they produce observable changes to execution semantics? The ability to quickly adapt the simulator to a changing instruction set may also be a desirable property, i.e. for an architecture under development.

In this paper we present the CASM language and demonstrate its usefulness for simulator generation. For this purpose we created high-level models of the MIPS instruction set. These models describe the observable semantics of MIPS instructions, their effects to observable machine state like registers and memory. We then refined these high-level models to describe execution in a classic 5-stage pipeline and modeled two different pipeline implementations. 1) A pipeline implementing operand forwarding and 2) a stalling pipeline inserting so called bubbles when a data hazard is detected.

We originally designed the CASM language to be used for compiler backend verification. The same CASM processor model can be used for compiler verification and processor simulation. CASM is capable of symbolic execution, which we use to perform model verification [20]. We are able to prove pipelined instruction models to be semantically equivalent to high-level models, which prevented any instruction model errors slipping into the simulators.

The remainder of the paper is structured as follows, section 2 introduces the CASM language, in section 3 we present the (almost) complete models for the MIPS architecture, in section 4 we present details about validation of the pipelined models. Section 5 gives details about the simulator runtime and in section 6 we present benchmark results for the generated simulators. Related work is discussed in section 7 and the paper concludes in section 8.

[*]This work is partially supported by the Austrian Research Promotion Agency (FFG) under contract 827485, *Correct Compilers for Correct Application Specific Processors* and Catena DSP GmbH.

2. CASM LANGUAGE

The CASM language is a refinement of the Abstract State Machine (ASM) [8] language described by Gurevich [12]. ASMs define a synchronous parallel execution model which is ideal to model cycled circuits like micro-processors.

The main concept of ASMs is the state, which is a set of *functions*, comparable to global variables in other programming languages. So called *rules* are executed and produce sets of *updates* which are applied atomically to the state when the rule *concludes* (returns). Rules may invoke other rules, in this case the update set produced by the invoked rule is not applied to the state but merged into the calling rule's update set.

A common extension to ASMs is *sequential composition* [5], which allows to write models more like traditional programs. CASM supports sequential ASM with the *seqblock* statement. A *seqblock* acts as if each update set is applied to the state before the next statement is evaluated (as opposed to the parallel execution normally performed). However, no changes to the global state are made. The update set produced by a *seqblock* describes the difference between the initial state and the state resulting from sequentially executing all statements of the block. The implementation of a swap (listing 1) points out the difference. The *:=* operator is used to express an *update* to a *function*, whereas the *let* statement purely binds a name to the given expression. *let* does not induce any updates.

```
rule swap = {        // parallel
  x := y
  y := x
}                    // values of x and y swapped

rule swap =
  let temp = x in    // temporarily store x
    seqblock
       x := y
       y := temp
    endseqblock
```

Listing 1: parallel and sequential swap

One design goal of the CASM language is the ability to be compiled into efficient code. The syntax of the language is inspired by CoreASM [10], but CASM added a static type system. ASM defines arguments to be passed by name which is difficult to implement efficiently [3]. CASM offers a call statement which evaluates all arguments before passing them. This way all arguments are constant values and pass-by-name is equivalent to pass-by-value. With this modification good performance of the generated code can be achieved [15].

ASM defines *external functions* as a way to communicate with the environment of a model. In CASM *external functions* are used to implement a domain specific vocabulary. Whilst originally intended to express symbolic operations on variable sized bit vectors (which, in conjunction with a set of axioms, are used to perform proofs) those functions can be concretely implemented and form building blocks for the instruction description. One example of such functions is *BVadd_result(32, a, b)* which performs a 32 bit addition using the 32 bit values *a* and *b*.

3. MODELS

In this section we present the models of the high-level and the classic 5-stage pipelined MIPS machine. An excellent description of the MIPS architecture is given by Patterson and Hennessy [17]. Wikipedias *Classic RISC pipeline* [1] article can also be recommended. Because of the expressiveness of the CASM language we are able to present the essential parts of the models and only skip a few initialization rules. The complete models consist of approx. 100 lines of code (LOC) (plus another 600 LOC to describe all instructions) for the high-level and approx. 400 LOC (plus 1500 LOC instructions) for the pipelined models. This is a quite concise specification compared to the 1054 LOC of a similar model in xADL, a specialized processor description language on a high abstraction level [6].

3.1 The State

Listing 2 gives the definition of the observable state of the MIPS architecture. The *MEMORY* function models the system memory and contains data and instructions. The functions *PARG* and *PMEM* are used as cache for decoded instructions. *PARG* contains decoded fields and *PMEM* contains the instruction itself. The function *GPR* (general purpose registers) is used to model the MIPS register set. Register 0 is always read as 0 and discards all writes. *PC* contains the program counter. *BRANCH* contains the calculated target address and is jumped to after the instruction in the delay slot has been executed. *CYCLES* is used to count the executed instructions, and *trapped* is set when a system trap condition is fulfilled. An implementation of the processor may add more function to the state, e.g. pipeline registers.

```
enum FieldValues = { FV_RS,FV_RT,FV_IMM,... }

function MEMORY : Int -> Int

function PARG: Int * FieldValues -> Int
function PMEM: Int -> RuleRef

function GPR : Int -> Int
function LO : -> Int
function HI : -> Int
function PC : -> Int

function BRANCH : -> Int

function CYCLES : -> Int
function trapped : -> Boolean
```

Listing 2: MIPS observable state

3.2 Instructions

The instructions operate on the state given in the previous section. As guiding example the *addiu* instruction is used. *addiu* adds an immediate value to a register, the immediate value needs to be sign-extended, the register containing the source operands is encoded in the field *rs* and the destination register is encoded in field *rt*.

Listing 3 shows the high-level description of *addiu* and the *write_reg* rule. *write_reg* models the fact that all writes to register 0 are discarded.

[1] http://en.wikipedia.org/wiki/Classic_RISC_pipeline

```
rule write_reg(reg : Int, val : Int) =
  if reg != 0 then
    GPR(reg) := val

rule addiu(addr : Int) =
let rs = PARG(addr, FV_RS) in
let rt = PARG(addr, FV_RT) in
let imm = PARG(addr, FV_IMM) in
  call(write_reg)(rt, BVadd_result(32, GPR(rs),
                  BVSignExtend(imm, 16, 32)))
```

Listing 3: ADDIU high-level model

Listing 4 shows the pipelined description of the *addiu* instruction. Each stage is further divided into 2 phases to model the fact that hardware signals can be produced and consumed in the same clock cycle. All signals are produced during the *begin* phase and are consumed during the *end* phase. In particular, registers are written in the *begin* phase of the write-back (*WB*) stage and are read during the *end* phase of the *ID* stage.

```
enum PipelineStages = {ID, EX, MEM, WB}
enum PipelinePhases = {begin, end}

rule addiu(addr:Int, stage:Int, phase:Int) =
{
  if stage = ID then
  if phase = end then
  let rs = PARG(addr, FV_RS) in
  let rt = PARG(addr, FV_RT) in
  let imm = PARG(addr, FV_IMM) in
  {
    call(ID_READ_OP1)(rs)
    IDOP2 := BVSignExtend(imm, 16, 32)
    IDRESREG := rt
  }

  if stage = EX then
  if phase = begin then
    EXRES := BVadd_result(32, EXOP1, EXOP2)

  if stage = WB then
  if phase = begin then
    call(write_reg)(WBRESREG, WBRES)
}
```

Listing 4: ADDIU pipelined model

3.3 High-level execution semantics

To execute the high-level instruction the rule given in listing 5 is sufficient. It executes the instruction at the current program counter and either takes a pending branch or simply increases the program counter. Execution stops when a trap occurred (*program(self) := undef* is the CASM idiom for terminating the program). The call instruction invokes the instruction model (*PMEM(PC)* yields a reference to a rule) at address *PC* and passes the current program counter as argument. *PC* is then used to fetch instruction fields (see listing 3).

```
rule run_program  =
let branch = BRANCH in
  seqblock
    call(PMEM(PC))(PC)
    CYCLES := CYCLES + 1

    if branch = undef then
           PC := PC + 4
    else {
           PC := BRANCH
           BRANCH := undef
    }
    if trapped then
           program(self) := undef
  endseqblock
```

Listing 5: Simulator Main Rule (high-level)

3.4 Pipeline with Forwarding

To implement operand forwarding, data words may need to be fetched from pipelined registers instead of accessing the register file, which still contains the old value. The rule *ID_READ_OP* is used to model this. It may need to forward the value from the *EX* or *MEM* stage. As registers are written in the *begin* phase of the *WB* stage, no special forwarding needs to be implemented for the *WB* stage. Listing 6 shows the implementation.

```
rule ID_READ_OP1(reg : Int) =
  if EXRESREG = reg then
    IDOP1 := EXRES
  else
    if MEMRESREG = reg then
      IDOP1 := MEMRES
    else
      IDOP1 := GPR(reg)
```

Listing 6: Operand fetch (forwarding)

The parallel execution semantics of the CASM language allow to easily write a rule to model the simultaneous execution of all pipeline stages (listing 7). Although the stages are split into a *begin* and a *end* phase their (combined) effects must be applied to the state atomically. This is what the *seqblock* ensures.

```
rule execute_pipeline =
seqblock
  forall s in PipelineStages do
    let op = pipeline(s) in
      if op != undef then
        call (PMEM(op))(op, s, begin)
  forall s in PipelineStages do
    let op = pipeline(s) in
      if op != undef then
        call (PMEM(op))(op, s, end)
endseqblock
```

Listing 7: Parallel execution of all pipeline stages

Listing 8 shows the implementation of the instruction fetch logic. The omitted rule *step_pipeline* simply moves each pipelined register and each instruction into the next stage. A new instruction is fetched from *PC* and transfered into the pipeline (stage *ID*). Any pending branches are handled subsequently, implementing the branch delay slot.

```
rule IF_stage =
let branch = BRANCH in
seqblock
  step_pipeline
  pipeline(ID) := PC

  if branch = undef then
    PC := PC + 4
  else {
    PC := BRANCH
    BRANCH := undef
  }
endseqblock
```

Listing 8: Instruction fetch (forwarding)

The main rule is shown in listing 9 and executed repeatedly until *program(self)* becomes *undef*. Unless a trap condition has been generated by any of the instructions this rule either executes the pipeline or fetches an instruction depending on the boolean value of *execute_pipeline*. This alternate execution of fetching and executing produces update sets which are very useful for tracing the program run. The update set produced (alternately) describe an instruction fetch and all operations performed by the instruction in the pipeline. The *dumps* annotation prints updates to the named functions to a debug stream (*trace*). Debug streams can selectively be enabled by providing command line switches to the simulator.

```
rule run_program dumps (GPR, LO, HI) -> trace =
if trapped then {
  dump_machine_state
  program(self) := undef
} else
  if pipeline_execute then {
      execute_pipeline
      pipeline_execute := false
  } else {
    IF_stage
    pipeline_execute := true
  }
```

Listing 9: Simulator Main Rule (pipeline)

3.5 Bubbling pipeline

To demonstrate the ease of modeling using the CASM language we also developed a model for a bubbling pipeline. In such a pipeline a no operation (*NOP*, bubble) is automatically inserted if a data hazard is detected by the hardware. To model this we need to change the definition of the *ID_READ_OP* rules. Listing 10 shows the new definition. Instead of forwarding the result a function is set to true, signaling that a data hazard has occurred. The definition of the instructions does not need to be changed at all.

```
rule ID_READ_OP1(reg : Int) =
  if EXRESREG = reg then
    BUBBLE1 := true
  else
    if MEMRESREG = reg then
      BUBBLE1 := true
    else
      IDOP1 := GPR(reg)
```

Listing 10: Operand fetch (bubbling)

In addition the *IF_stage* rule has to be modified, the new definition is shown in listing 11. If any operand fetch triggered a data hazard the pipeline is stalled, the calculated branch target address is invalidated and no new instruction is fetched.

```
rule IF_stage =
let branch = BRANCH in
let bubble = (BUBBLE1 or BUBBLE2 or BUBBLE3) in
seqblock
  step_pipeline
  if bubble = false then
    seqblock
    pipeline(ID) := PC
    if branch = undef then
      PC := PC + 4
    else {
      PC := BRANCH
      BRANCH := undef
    }
    endseqblock
  else {
    BUBBLE1 := false
    BUBBLE2 := false
    BUBBLE3 := false
    BRANCH := undef
  }
endseqblock
```

Listing 11: Instruction fetch (bubbling)

The pipeline needs to be stepped partially, listing 12 shows the details. Copying of the pipeline register has been removed for clarity. The instruction stays in the *ID* phase, the subsequent stages of the pipeline perform a step. In the *EX* phase a no operation (here the special value *undef*) is inserted.

```
rule step_pipeline =
{
  pipeline(WB) := pipeline(MEM)
  pipeline(MEM) := pipeline(EX)

  if (BUBBLE1 or BUBBLE2 or BUBBLE3) then {
    pipeline(EX) := undef
    EXRESREG := undef
  } else {
    pipeline(EX) := pipeline(ID)
    pipeline(ID) := undef

    // copy pipeline registers ID -> EX stage
    IDRESREG := undef
  }

  // copy pipeline registers EX -> MEM stage
  // copy pipeline registers MEM -> WB stage
}
```

Listing 12: Pipeline step (bubbling)

4. MODEL VERIFICATION

Whilst the high-level descriptions of the instructions are very easy to write, and could be extracted from a processor manual automatically like described in [4], the pipelined models are error prone to write. There are also some side conditions for writing correct models which may be violated with hardly noticeable effect. One could, for example, directly read the register file after the *ID* stage (correct behavior is to read in the *ID* stage and store it into a pipeline register). Such an error would only manifest if the register value changes between the *ID* stage and the stage which erroneously reads the register file. Chances are good that such errors slip all (small) test cases and only occur in large applications where they are difficult to find. Another source of error is the splitting of the operations into the various stages. One can easily swap operands and the like. We rule out such error by automatically proving the pipelined models to be equivalent to their high-level counterparts using the first-order theorem prover Vampire [19].

Instruction models can be understood as state transition systems, applying a set of well known bit vector operations to some state. The key observation is: two instructions models are semantically equivalent *iff* they induce the same state transition of the observable state [11]. For the MIPS instructions the observable state is exactly the state described in section 3.1. The pipeline models add additional state (e.g. pipeline registers) but this state is not observable and can therefore be ignored when reasoning about semantics.

We have developed a CASM interpreter which features symbolic execution and writes trace files describing the state transformations as logical facts in TPTP format [21], which is an input format supported by many theorem provers.

Symbolic Execution is a technique were symbols are provided to a program instead of normal values. A new symbol is returned by operations when their arguments are symbolic. This new symbol is associated with the fact that it resulted from the aforementioned operation. So each symbol is either a program input or it is known how it was calculated. Starting from an arbitrary symbolic value, one can reconstruct the sequence of operations performed and will eventually have an expression over input symbols. Expressions associated with symbolic values being part of a programs final state allow that value to be expressed in terms of input symbols. These expressions link the final and the initial states and allow reasoning over the model's induced state transition [14].

We symbolically execute the high-level and pipelined model of an instruction, generating two trace files. In each trace we identify the values changed. Because of the symbolic execution trace we can express each changed value in terms of initial symbols and a sequence of operations. For the pipelined model to be correct we require, that for each expressions associated with a changed value of the (observable) state there is an equal expressions (associated with the same changed value) in the high-level trace and vice versa. This equivalences can be proven by a theorem prover.

Listing 13 shows the result of symbolically executing the high-level model for the *addiu* instruction. Reading that trace backwards we see that *sym8* is written to the function *GPR* (register file) and the register written to is *sym4*. An update $func(a) := v$ is described by a predicate $ffunc(t, a, v)$. The first argument (t) is a sequential number acting as a (logical) time stamp. Further tracing identifies *sym8* to be the result of an 32 bit addition of *sym6* and *sym7*. *Sym6* is read from register *sym3*, logical time 0 identifies this to be the initial state, so tracing back ends here. The (omitted) trace produced by the pipelined model for the *addiu* instruction is 176 lines long and cluttered with facts about changing pipeline registers.

```
fof(id2,hypothesis,fPARG(0,0,2,sym3)).
fof(id3,hypothesis,fPARG(0,0,0,sym4)).
fof(id4,hypothesis,fPARG(0,0,5,sym5)).
fof(id5,hypothesis,fGPR(0,sym3,sym6)).
fof(id6,hypothesis,
    fSignExtend(sym5, 16, 32, sym7)).
fof(id7,hypothesis,
    fadd_result(32, sym6, sym7, sym8)).
fof(id8,hypothesis,fGPR(0,sym4,sym10)).
fof(id9,hypothesis,fGPR(1, sym4,sym8)).
```

Listing 13: Symbolic execution of high-level ADDIU

Technically we execute the high-level model of an instruction and prefix all symbols and state functions. We then execute the corresponding pipeline version of the instruction using the CASM code shown in listing 14 and differently prefix all symbols and state function. The *step_pipeline* and *executed_pipeline* rules call in 14 are exactly the same rules presented in section 3. For that reason the resulting trace also covers the pipeline implementation itself, not just the instruction model. Basic correctness properties of the pipeline models are therefore proven as well. *Undef* is assigned to all field values after the *ID* stage of the instruction has been executed. Should the instruction model access the register file in a stage other then *ID* it will result in an *undefined* read and the proof will fail.

```
PC := 0
PMEM(0) := @addiu
IF_stage
execute_pipeline

PARG(0, FV_RD) := undef
// undef all other fields as well

step_pipeline
execute_pipeline
// repeat
```

Listing 14: Proof generation (pipeline, addiu)

A proof script containing both traces is generated. The observable initial state of both traces is defined to be equivalent (by axioms). A conjecture is emitted stating that the observable functions are identical in the final state of the traces. The theorem prover is then used to perform the proof (proving all expressions associated with the changed values of the observable state to be equal). Despite the undecidability of first-order theorem proving in general, these very simple proofs all succeed very quickly.

5. SIMULATOR CONSTRUCTION

The CASM models are compiled to C++ using a straight forward compilation scheme. The update sets produced by the statements are implemented using hash maps.

Modeling the instruction set and execution semantics alone does not produce a usable simulator. One also needs a way to load programs and an environment to execute them. For that purpose we enriched the CASM runtime with an ELF loader and an instruction decoder.

To provide an environment we decided to port the newlib C library for the simulator. The newlib library is written with portability in mind and only a handful operations must be implemented to have a fully featured C library [2]. We imple-

[2]http://wiki.osdev.org/Porting_Newlib

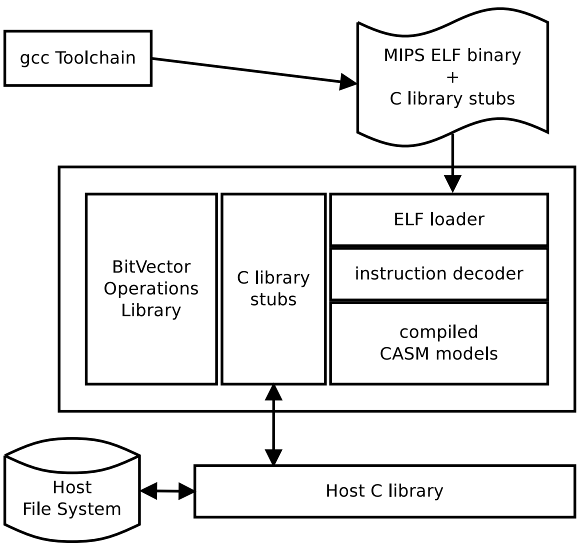

Figure 1: Simulator Architecture

ment these functions using the *syscall* instruction and link the resulting stubs library with all programs to be executed within the simulator.

In the simulator runtime the implementation of the *syscall* instruction is provided by native C++ code. Arguments are passed using the default MIPS ABI and the *syscall* implementation reads them accessing the *GPR* function. All arguments are translated into host system representation (endianness, word size), and the operation is then performed by the host C library. Operations involving memory buffers (e.g. read / write) need to be implemented using the *MEMORY* function. Figure 1 gives an overview of the simulator.

6. EVALUATION

Creation of the models for the instruction set (high-level and pipelined) was a task of 2 days. Modeling the forwarding pipeline took another day, but creating the bubbling pipeline took only 15 minutes. Most of the work was writing the pipelined models which also proved to be tedious and error-prone. Because of the model validation none of the copy-and-paste errors stayed unnoticed however. We found typical micro processor idioms to be elegantly expressed in the CASM language.

We used the small data sets of the MiBench [13] embedded benchmark suite and report the best of 3 runs. We also executed the 164.gzip benchmark of the SPECInt2000 suite, with test data size, though. Nonetheless we did not attempt 164.gzip with the bubbling pipeline simulator (estimated runtime: 3 days). The tests were performed on a Intel Core i7 Q820 @ 1.73 GHz with 8 GiB RAM running on a 64 bit Ubuntu 12.10 and are presented in figure 2.

The performance of the ISS (generated from the high-level) modes is approx. 950 kHz (instructions per second) on average with a peak performance of 1046 kHz (qsort). For CRC32 (775 kHz) and basicmath (380 kHz) the number of simulated instructions per second is much lower. Both benchmarks are dominated by I/O operations, reading the file (CRC32) and printing results (basicmath). The I/O implementation of the simulator performs poorly, so the results are to be expected. For the simulator generated with the forwarding models approx. 48 kHz simulation speed is achieved on average. The main reason for the huge performance gap compared to ISS is a suboptimal handling of large update sets in the generated code.

Surprisingly the bubbling simulator performs better (approx. 50 kHz) at the first glance. The number of simulated cycles is substantial larger than for the other simulators because of the automatically inserted bubbles. As bubbles are no operations they perform much better than real instructions. Reducing the cycle counter by the number of generated bubbles yields the instruction throughput which better indicates the simulation runtime. In figure 2 this is called Bubbling* and simulator performance is approx. 29 kHz.

The measured jitter was very low, which is expected as no dynamic optimizations are applied. Time measurement is implemented in the simulator using the *getrusage(2)* system call. It excludes the time it takes to load the simulator, the ELF binary and time for initializing data structures.

7. RELATED WORK

A good introduction into instruction set simulation gives the book chapter by Brandner et.al [7]. They describe the different implementation techniques, present existing processor description languages and discuss a large set of the related work.

Teich, Kutter and Weper [23] specified the ARM instruction set using ASM. They generate a cycle-accurate simulator based on the RTL description of the processor, but do not report on the achieved performance.

Nohl et.al [16] describe generating instruction set simulators using the LISA processor description language. LISA is a domain specific language for describing micro processors. They used JIT compiler technologies to improve performance.

Rajesh and Moona [18] describe a very similar approach to what we have presented in this paper. They use the Sim-nML language to describe the instruction set and generate C++ code which is linked to a runtime for generating a system simulator. Performance of their simulator is reported to be approx. 3000 instructions per second, although published in 2000. Assuming doubled CPU speed every 18 month a rough estimate would be around 768 kHz performance on modern hardware, approximately the same speed we report here. Casse el.al [9] report on on various optimization in a Sim-nML derived simulator. They achieve simulator performance above 10 MHz.

The Unisim project relies on the SystemC language and its tools [22] to provide an execution environment for the simulation of processors [1]. The project offers several components, such as memories, caches, buses, and even complete processors, with well defined interfaces that can quickly be integrated with other components.

Like the Unisim framework, the *ArchC* processor description language is based on SystemC [2]. Following the SystemC

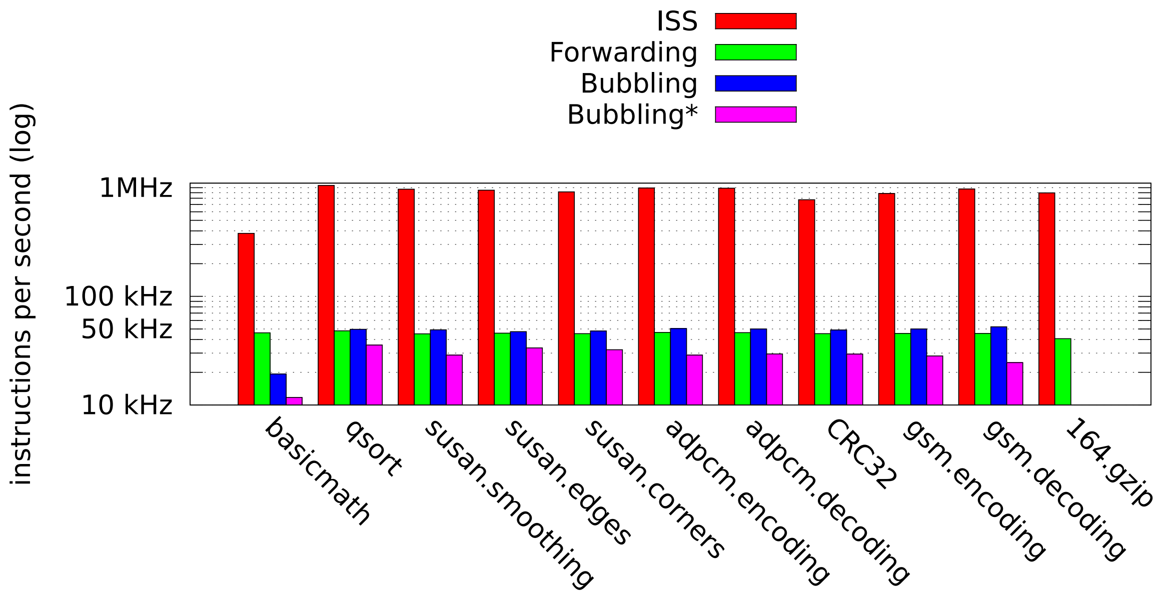

Figure 2: Simulator Performance (instructions per second)

TLM standard it is thus possible to integrate the derived simulators with external SystemC modules.

Brandner et al. [6] developed the xADL processor description language. Efficient instruction set simulators are using the just-in-time compiler of the LLVM compiler infrastructure. In addition to simple basic blocks, hot *regions* of code are recognized and recompiled using more aggressive optimizations. These regions may contain arbitrary control flow, including loops.

LISA, nML and xADL are processor description languages with a high abstraction level. This allows better optimization of the resulting instruction set simulators and higher simulation speeds. CASM like SystemC and ArchC is a more general purpose modeling language. This makes optimizations more difficult. If coding conventions are respected common specification patterns can be recognized and optimized.

8. CONCLUSION AND FUTURE WORK

We have shown how the CASM language can be used to elegantly model a pipelined micro processor. By using the feature of symbolic execution we were able to prove that the error-prone pipelined version of the instruction set models are semantically equivalent to the much simpler high-level models. No bugs manifested in any of the executed benchmark. We found this to be a very useful property of using the CASM language for modeling instruction sets. It would also be very interesting to extend the proof system and perform a complete verification of the pipeline implementation.

Simulator performance is satisfying considering the fact that CASM is not a language dedicated to simulator description. There is however ongoing research on how to further improve the performance of the generated simulators, and a few culprits have already been identified. We are also interested in completing the newlib port to be able to execute arbitrary C programs.

9. REFERENCES

[1] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani. UNISIM: An open simulation environment and library for complex architecture design and collaborative development. *IEEE Computer Architecture Letters*, 6(2):45–48, 2007.

[2] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros. The ArchC architecture description language and tools. *Int. J. Parallel Program.*, 33(5):453–484, 2005.

[3] J. Bergin and S. Greenfield. Teaching parameter passing by example using thunks in C and C++. *SIGCSE Bull.*, 25(1):10–14, Mar. 1993.

[4] F. Blanqui, C. Helmstetter, V. Joloboff, J.-F. Monin, and X. Shi. Designing a CPU model: from a pseudo-formal document to fast code. *CoRR*, abs/1109.4351, 2011.

[5] E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In *Computer Science Logic (Proceedings of CSL 2000), volume 1862 of LNCS*, pages 41–60. Springer-Verlag, 2000.

[6] F. Brandner, A. Fellnhofer, A. Krall, and D. Riegler. Fast and accurate simulation using the LLVM compiler framework. In *RAPIDO '09: 1st Workshop*

on Rapid Simulation and Performance Evaluation: Methods and Tools, 2009.

[7] F. Brandner, N. Horspool, and A. Krall. DSP instruction set simulation. In S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, editors, *Handbook of Signal Processing Systems*, pages 679–705. Springer, Aug. 2010.

[8] E. Börger. Abstract state machines: A method for high-level system design and analysis, 2003.

[9] H. Cassé, J. Barre, R. Vaillant-David, and P. Sainrat. Fast Instruction-Accurate Simulation with SimNML (regular paper). In *Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO), Heraklion, Crète, Grèce, 22/01/11*, pages 8–12, http://univ-lille1.fr, janvier 2011. Université de Lille.

[10] R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An extensible ASM execution engine. In *Proc. of the 12th International Workshop on Abstract State Machines*, pages 153–165, 2005.

[11] G. Goos and W. Zimmermann. Verifying compilers and ASMs. In *Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, ASM '00, pages 177–202, London, UK, 2000. Springer-Verlag.

[12] Y. Gurevich. *Evolving algebras 1993: Lipari guide*, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[14] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[15] R. Lezuo and A. Krall. A unified processor model for compiler verification and simulation using asm. In *ABZ*, pages 327–330, 2012.

[16] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation, 2002.

[17] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012.

[18] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *in Int. Conf. on VLSI Design*, pages 132–137, 2000.

[19] A. Riazanov and A. Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15:91–110, Aug. 2002.

[20] R. G. Sargent. Verification and validation of simulation models. In *Winter Simulation Conference*, WSC '09, pages 162–176. Winter Simulation Conference, 2009.

[21] G. Sutcliffe, S. Schulz, K. Claessen, and A. Van Gelder. Using the TPTP language for writing derivations and finite interpretations. In *Proceedings of the Third international joint conference on Automated Reasoning*, IJCAR'06, pages 67–81, Berlin, Heidelberg, 2006. Springer-Verlag.

[22] Open SystemC Initiative. `http://www.systemc.org/home`.

[23] J. Teich, P. W. Kutter, and R. Weper. Description and simulation of microprocessor instruction sets using ASMs. In *Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, ASM '00, pages 266–286, London, UK, 2000. Springer-Verlag.

[24] J. J. Yi and D. J. Lilja. Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations. *IEEE Trans. Computers*, 55(3):268–280, 2006.