Live Introspection of Target-Agnostic JIT in Simulation

Boris Shingarov
shingarov@labware.com

Abstract

A debugging system is described wherein an outer Smalltalk running on a real machine interacts with a GEM5 simulation of an inner Smalltalk. An experimental embodiment is applied towards understanding the behavior of a target-agnostic JIT prototype.

Categories and Subject Descriptors D.3.4 [*Programming languages*]: Processors

Keywords Smalltalk Virtual Machine, Liveness, Full-System Simulation, GEM5, Retargetable Compiler, Unification, GDB Remote Serial Protocol, Processor Description Language

1. Introduction

The traditional software observation approaches are increasingly proving inadequate as tools for understanding the ever more complex behavior of modern software. On the one hand, the conventional debugging techniques [34] are ineffective at explaining such classes of intricate effects as e.g. race conditions in a multicore system.

On the other hand, understanding correctness is only one side of the challenge; the other side is performance. Raw execution speed (as exemplified by "bytecodes/second" and "sends/second" metrics for a Smalltalk VM) is no longer the key performance metric, since energy consumption and heat constraints are becoming more important limiting factors. In addition, an optimization leading to an improvement in one application workload, can cause a deterioration in another, so there is no single measure of "code quality". Moreover, the desire for higher quality dynamically generated machine instructions must be balanced against the efficiency of the generator itself. This is so because in a dynamically translated runtime, the JIT translator and the code it generates compete for the same computational resources.

The complexity of contemporary state-of-the-art VMs is further increased by explicit michroarchitectural considerations (cf. the cache-awareness optimizations in a Java VM in [9]). Because of these challenges, the decisions related to the fine-tuning of the translator back-end can no longer be founded on the compiler expert's intuition about optimization strategies, but must be supported by measurement — possibly even at run-time by the JIT itself [4, 36]. Moreover, this measurement of these heavily microarchitecture-dependent effects will be specific to each realization of a given ISA by a particular system implementation (including the particular CPU core, cache implememtation, physical semiconductor embodiment by the chip manufacturer, etc.)

Simulated virtual platforms [23] are attracting increasing interest from both academia and industry because of their potential to help solve these problems. Full-system simulators such as Simics [1, 24, 25, 27] and GEM5 [10, 12, 18, 19, 23, 25, 35] have proven their effectiveness in applications to research and industrial problems such as debugging operating systems and hypervisors, understanding intricate defects in microprocessor design, and explaining unexpected energy consumption patterns in embedded systems, to name a few.

This paper describes an experiment in which virtual platform simulation using the open-source GEM5 system was applied to gain insight into the behavior of a new JIT translator for Smalltalk.

2. The Debugee JIT

The need for a new simulation-based observation tool grew from the practical requirement to understand the behavior of a prototype of an automatically-retargetable (a.k.a. "target-agnostic") JIT translator. Although that target-agnostic translator has been described elsewhere [38, 39], a quick recapitulation of its function will promote a better understanding of the driving forces behind the design of the simulation-based debugger which is the main subject of this paper.

2.1 Automatically-Retargetable Code Generation

The Target-Agnostic JIT is a Smalltalk v-code-to-n-code [15] translator backend with no knowledge of the target

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

IWST '15, July 15, 2015, Brescia, Italy.
Copyright is held by the owner/author(s).
ACM .
http://dx.doi.org/10.1145/

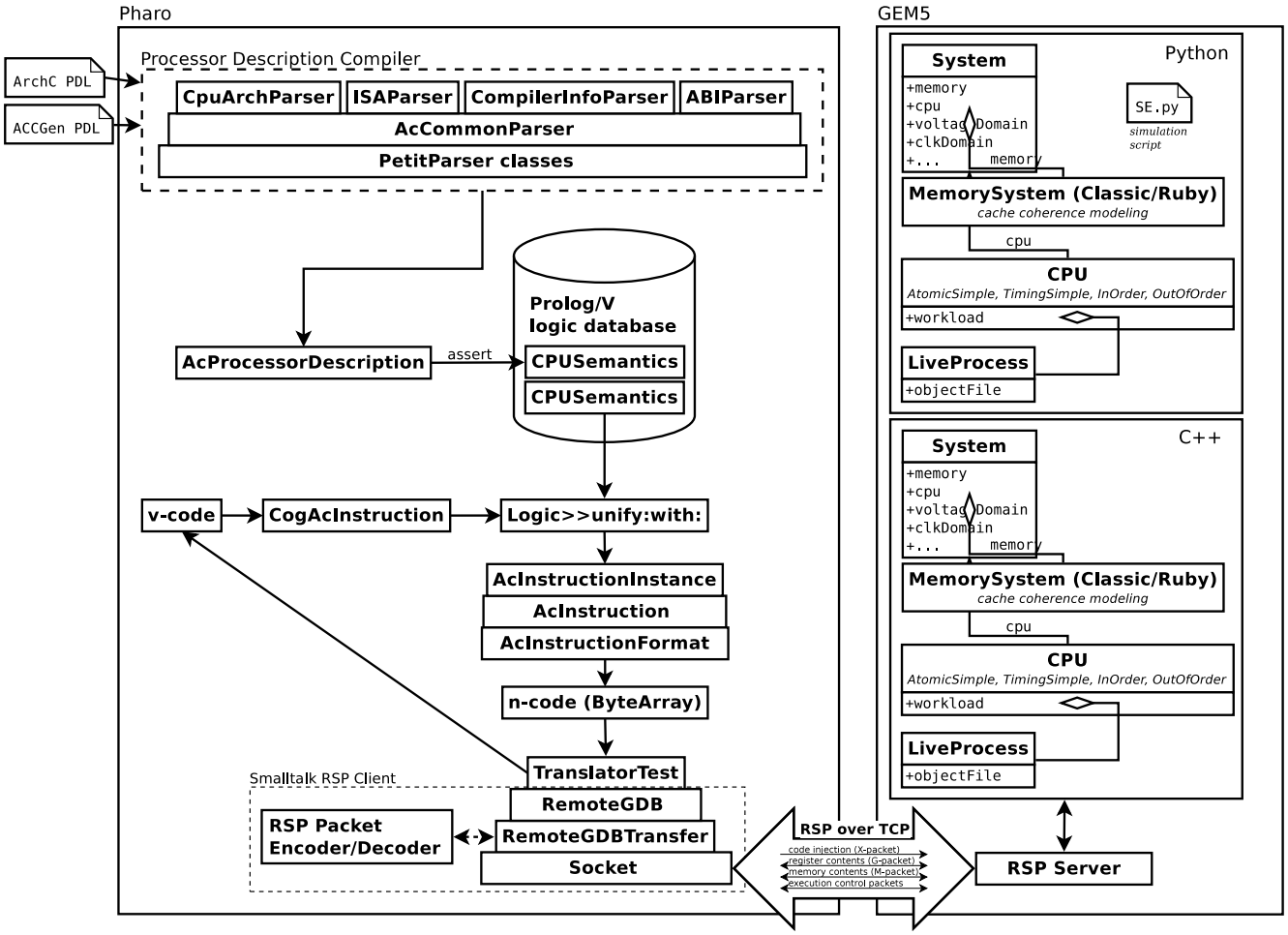

Figure 1. Overall Structure of the *"Target-Agnostic JIT"* Experiment

processor architecture coded in. It acquires target awareness at run-time by processing a formal specification of the target ISA written in a Processor Description Language (PDL) [17, 26, 31, 42] called ArchC [6]. Because the ArchC language is mostly oriented towards simulator generation, as opposed to machine reasoning about the ISA semantics which is essential for hardware synthesis and compiler derivation, the ISA semantics are extracted from a second specification written in the ACCGen language [5]. The Smalltalk JIT does not use any part of the ArchC nor ACCGen software, other than the language itself, and the specifications for four popular processor architectures.

If any PDL is to be useful for automatic code generator derivation, it must specify the following information about the target [13]: (1) the registers representing the processor state (General-Purpose Registers (GPR), Special-Purpose or Reserved Registers such as the Program Counter, condition flags, etc.); (2) addressing modes; (3) instruction encodings, (4) data types, and (5) machine operations, assigning each machine instruction a cost and a semantic definition. The following is an example of ACCGen semantic definition:

```
define instruction addu semantic as (
  (transfer Op1:GPR (+ Op2:GPR Op3:GPR));
) cost 1;
```

These semantic statements essentially define a function from instructions to primitive fragments of the semantic effect tree (called "I/O assertions" in [13]); however, they provide no computational recipe for going in the opposite direction — i.e., from effects to instructions. This can be represented by the following equation:

$$SJ = T \tag{1}$$

where J is an instruction, T is its corresponding I/O assertion, and S is the "semantic operator"; each instance of S (such as, $S_{SPARCv8}$ or S_{MIPS-I}) contains the whole plurality of instruction semantics for the given ISA. The task of code generation is to solve the inverse problem for (1), i.e. to determine J given S and T. This solution has to exhibit a number of obvious general properties, such as computability (the function has to be total over all IR inputs, i.e. terminate in finite time producing either a J or a proof that $SJ \neq T \forall J$)

and confluence (the function value has to be independent of the particular computation strategy chosen by the evaluator).

2.2 Unification

A target-specific translator backend benefits from human-programmed heuristic strategies for inverting the semantic operator S. In an automatically-retargetable backend, S is not known ahead of time. It is given by the processor description:

PDL $\xrightarrow{parse} S$.

However, merely parsing the PDL does not give us S^{-1}. In order to generate code, we need a general method for solving equation (1).

Because S, J and T don't seem to have any immediately recognizable algebraic structure, the only mathematical theory directly applying to (1) seems to be unification, which concerns itself with the study of equations as such. Indeed, a simple equational-theoretic argument yields a short and elegant program for computing S^{-1}. The idea is the same as that of the basic word-problem algorithm of Knuth–Bendix [21]. Let's start with a simple example. (Because Prolog is probably the most widely known language for logic programming, let's use that language to express the rewriting program).

2.3 Instruction Semantics Rewriting

Consider the PowerPC instruction $addiu\ RD,\ RA,\ D$. Here RD and RA are 5-bit-wide GPR numbers, and D is a 16-bit-wide immediate constant. This instruction adds the unsigned immediate constant D to the contents of register RA, and the result is stored in register RD. It follows the usual register-instruction "RA-convention" that if RA is registered to $r0$, then zero is used instead of the contents of $r0$.
Here is the ACCGen definition for addiu:

```
define instruction addiu semantic as (
  let Op2 = "0" in
  (transfer Op1:GPR imm:Op3:tgtimm);
) cost 1;

define instruction addiu semantic as (
  (transfer Op1:GPR (+ Op2:GPR imm:Op3:tgtimm));
) cost 1;
```

The parser for the ACCGen grammar can trivially transform these (and any number of other) definitions into a Prolog clause string:

```
transfer(Op1,Op3) => addi(Op1,Op2,Op3) :-
  Op2 = 0,
  integer(Op3), Op3 < 65536,
  true.

transfer(Op1,+(Op2,Op3)) => addi(Op1,Op2,Op3) :-
  integer(Op3), Op3 < 65536,
  true.
```

Each of these axioms can be seen as a conditional rewrite [8]. The operator "=>" here has no special significance; it is simply the functor "=>/2" in infix form:

```
op(500, xfx, =>).
```

A program to invert S combines any number of these instruction semantics definitions with the Cheng–van Emden–Parker's equation axioms [14, 32]:

```
e(X,Y) :- e2(X,Y).
e2(X,Z) :- e1(X,Y), e2(Y,Z).
e2(X,X).
e1(X,Y) :- X => Y.
```

The above four lines turn the logic interpreter into a rewrite machine.
We also need the following left- and right-substitutivity axioms:

```
e1(transfer(A1,B), transfer(A2,B)) :- e1(A1,A2).
e1(transfer(A,B1), transfer(A,B2)) :- e1(B1,B2).
```

Already this tiny program is sufficient to emit some code:

```
:- e(transfer(gpr(2), 5), J).
J = addi(gpr(2), 0, 5).
```

What happens if the constant does not fit in the immediate operand field?

```
:- e(transfer(gpr(2), 500000), J).
false.
```

This is because we still have to define the semantic equivalence properties of the abstract I/O effects. These specify the valid IR tree transformations which can be applied without changing the semantics of the program. For example, it is possible to store an intermediate computation result in a scratch register; this can be programmed as the following rewrite axiom:

```
transfer(A,C) => [Y,X] :-
  e(transfer(A,B), X),
  e(transfer(B,C), Y).
```

Many of these semantic equivalence axioms take us out of pure symbolic processing into meaning domains such as arithmetic. In the example above, the generator failed to emit a synthetic immediate constant because it had no notion of decomposition of integers into the result of an arithmetic operation over smaller-width integers. The Prolog clauses expressing such axioms will contain arithmetic predicates such as "is", "==/==" etc., triggering arithmetic computations as opposed to pure logical unification. For example, the meaning of "left shift" can be expressed thus:

```
transfer(Rd, V << S) => [X,Y] :-
  integer(V), integer(S),
  e(transfer(Rtmp,V), X),
  e(transfer(Rd,shl(Rtmp,S)),Y).
```

Such semantic equivalence axioms operate at the level of the abstract IR tree and not of the machine instructions. (They are the machine-independent part of S, and programmed only once.) Even if we decide to aid the generator by providing a heuristic hint, (favoring a particular shift amount for decomposition based on the immediate field width):

```
transfer(R,Imm) => transfer(R, Hi \/ Lo) :-
  integer(Imm),
  Hi is (Imm /\ 0xFFFF0000),
  Lo is (Imm /\ 0x0000FFFF).
```

where the `0x0000FFFF` and `0xFFFF0000` masks are precomputed from the

```
redefine operand tgtimm size to 16;
```

statement of the ACCGen description at parse time, the translator is still machine-invariant, all such information being parametrized by the PDL.

Upon adding similar transliterations of the ACCGen definitions for "or" and "shift", our program is now complete enough to emit combinations of register loads and or's of any size:

```
:- e(transfer(gpr(2), 500000), J).
J = [[addiu(_G2323, 0, 7),
instr_shl(_G2321, _G2323, 16)],
addiu(_G2322, 0, 41248),
ore(gpr(2),_G2321), _G2322))].
```

(Actually, the definition for shift is slightly more involved because on the PowerPC the left shift needs to be expressed in terms of rotate, but this does not change the operation of the rewriter).

2.4 Bootstrapping the VM

The simple code generator described above, exhibits some interesting properties. Notably, in the answer from `e/2` which we obtained in the previous section, J is not fully grounded. On the trivial level, this means that we have only performed instruction selection but not register allocation. On a much deeper level, the algorithm is capable of performing *narrowing* as opposed to mere *reduction*. This opens some interesting possibilities.

The dynamic nature of the JIT, namely the fact that it competes with the mutator for processor cycles, makes translation speed a critical component of the overall performance of the VM. Cattell [13] factorizes the translator into a *code generator* and a *code generator generator*, therefore distinguishing between a *machine description* and a *machine table*. In essence, a machine table is a pre-computed S^{-1}. The ability of the rewriter to perform narrowing, means that it is not necessary to do a full rewrite for every concrete T. Indeed, if J can be partially grounded (or, in Knuth–Bendix terminology [21], an "impure word"), then T can be too. Combined with the ability of certain Prolog implementations (such as SWI-Prolog) to compile logic programs into native binary, this can mean getting the ability to save S^{-1} into a native object format (such as `.so`) essentially for free effortwise. Choosing a bootstrapping strategy becomes a matter of programming. In general, this JIT does not aim to be Slang-transliterated into C for execution. It can be encapsulated in a native object format using a Prolog compiler, or, if reliance on an external compilation tool is undesirable, — such as in the context of trustworthy computing where formal set-theoretic or first-order-logic proofs are mandatory — it can start from a running full Smalltalk in the first phase of boorstrapping. How much is synthesized within that outermost Smalltalk, can vary considerably: at one extreme, it can be a short sequence of instructions (or even one instruction) representing the concretization of one abstract RTL instruction; at the other extreme, a complete base Smalltalk system including the translator itself (and certainly everything needed for the translator operation) is ahead-of-time translated and can then be lauched as a second-generation interactive environment. Application deployment is then reduced to a position somewhere in-between these extremes.

2.5 Smalltalk Embodiment

The JIT prototype built by the author for conducting these experiments, consists of several components. A parser based on the PetitParser framework parses the ArchC and ACCGen processor descriptions arriving at a number of Smalltalk objects:

AcProcessorDescription is the top-level processor description object containing structural information such as the definitions of the machine's primary memory, storage bases, word sizes, as well as a collection of instruction encoding format definitions and a collection of available instruction definitions.

AcInstructionFormat represents an instruction format scheme. From a data representation point of view, it is a sequence of bitfield name-to-width associations. From a code generation point of view, it is the least specific level of variable binding: at this stage, only the family of binary instruction encoding is known.

AcInstruction objects encapsulate name, assembly syntax, encoding, and a forest of semantic trees with their costs. From a variable binding point of view, only those fields are bound that specify which instruction was selected (e.g., $opcode$), but not any $operand$ fields; those are still unbound.

GroundedInstruction specifies the instruction fully, including the values of all operands.

During the parsing of the ISA semantics specification, rules about the ISA are `assert`'ed into the Prolog database. The final result of parsing the processor specification is an `AcProcessorDescription` object containing the Instruction Format and Instruction objects (representing "static"

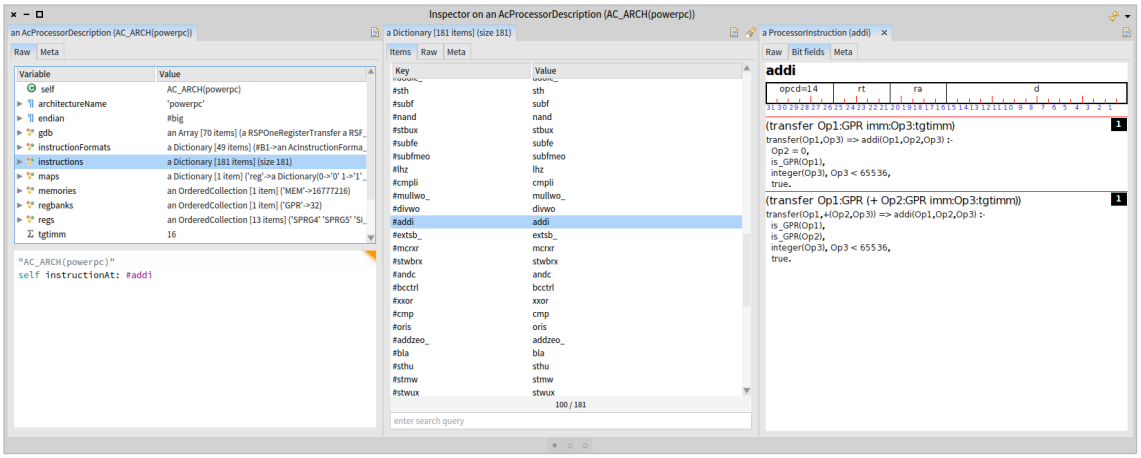

Figure 2. GTInspector showing an AcProcessorDescription
In the middle is the Dictionary of machine instructions. The instruction *addi* is showing on the right, including the encoding and the forest of two semantic trees; for each tree, we can see the ACCGen source, the Prolog rewriting rule, and the cost.

knowledge such as instruction encoding and mnemonics), and the logic database representing the knowledge of semantics. The actual instruction selection is achieved by matching the IR tree against the logic rules using the standard Prolog resolution engine as described above.

The result of the Prolog query is a list of fully grounded terms; these are converted back into an Array of GroundedInstructions which are in turn `#collect:`'ed to yield the final ByteArray of n-code.

These classes easily integrate with the Glamour tools to facilitate exploration of the processor model. Figure 2 shows one example where the GTInspector is open on an AcProcessorDescription instantiated by parsing the PowerPC ArchC definition.

In his experiments conducted by the author so far, he made no attempt to exploit the potential for higher code quality offered by matching the entire IR tree. The size of the instruction selection problem was strictly confined to concretization of individual "Abstract RTL Instructions" of the Cog translation framework. This is because the current instruction selector suffers from a number of problems. First, it performs depth-first search instead of dynamic-programming factorization of the input tree. Although it would be possible to implement the latter [11], the author chose the former as the most general formalism to aid understanding the properties of target-agnostic instruction selection. Therefore, the generator would quickly die of combinatorial explosion if confronted with a sizable IR input. Second, at this time there is no register allocator: the code simply takes as many scratch registers as there are intermediate results. For IR inputs representing Cog's abstract RTL, this just happens to fit in the modeled processors' register files. Third and perhaps worst of all, it does not take into account weights (instruction costs) and there is no complete proof of computability nor of confluence. While the generator appears to work on simple inputs, there are no fundamental guarantees that e.g. changing the order of instruction definitions will not change the emitted instructions, or cause the algorithm to not terminate altogether. Formulating such complete guarantees is a direction for future work.

3. The Simulation-Oriented Debugger

To investigate the behavior of the generated n-code on the various currently available targets, and with a view towards future (possibly unconventional) targets, the author has developed a debugging system based on fully-controlled simulation of the target machine. The central element of this system is the GEM5 simulator; however, the versatility of the Remote Serial Protocol used to communicate between the host Smalltalk and the container of the target, allows for many other interesting options. For example, one could use Simics [1] as the target simulation vehicle. Even more intriguing possibilities are unlocked by having an RSP-enabled scaffolding around a softcore running on reconfigurable silicon.

Unlike traditional tests in Smalltalk which run in a full live Smalltalk system and therefore bring its full complexity into every test, the new JTAG-boundary-scan-like introspection facility allows to partition the subsystem under test into a factorization which can be understood in ways fundamentally unattainable when observing a part interacting with a whole system. The reason is that in general,even a micro-scale subsystem can't be described in deterministic mechanical terms once it is a part of a non-deterministic macrosystem.

3.1 The GEM5 Simulator

GEM5 is an open-source framework for discrete-event simulation of computer platforms primarily aimed at architecture research [10, 12, 18, 19, 23, 25, 35]. It models many

processors and memory hierarchy organizations at various levels of fidelity depending on which aspect of the processor the researcher is interested in studying, such as system features below the architectural level, e.g. cycle-accurate microarchitecture-faithful timing and/or energy consumption.

3.2 Simulated Processor Workloads in GEM5

GEM5 is scripted in Python. Running the simulator comprises two phases. In the configuration phase, a system of Python objects is instantiated which represents how the different elements provided by the framework are put together into a complete computer system simulation. This is performed by a Python script called the "Simulation Script". This hierarchy of objects is mirrored by a parallel hierarchy of C++ objects. In the simulation phase, these C++ objects are manipulated by the discrete-event simulation engine.

There are two example simulation scripts provided in the GEM5 distribution which in many simple cases allow the user to run simulations without having to write their own simulation scripts. The pre-packaged "FS.py" script enables full-system simulation of a complete unmodified operating system. This script is sufficient for many scenarios of FSS-debugging an OS kernel.

The other pre-packaged script, "SE.py", allows the GEM5 user to model the execution of a userspace process in Linux. The Python class `LiveProcess`, together with its C++ counterpart, implement the concept of guest OS process; this includes putting the CPU model into "syscall emulation" mode which redirects the guest's syscall interrupts to the host, mapping guest memory to host memory (since the SE mode replaces the MMU operation model), etc. `LiveProcess`'s platform-specific subclasses encapsulate the knowledge of the ELF object code format and a few other platform specifics. Figure 3 shows the `LiveProcess` class hierarchy in standard GEM5.

3.3 The Modified SE Workload

The primary function of GEM5 in the presently described experiment was to allow the investigation of the behavior of code emitted by the target-agnostic JIT. To achieve this, a new workload class was added to GEM5. Instead of reading program text from ELF, establishing the environment and setting the initial register values, such as the Program Counter, it fills these values from data received from the outer Smalltalk via the GDB Remote Serial Protocol. The remaining aspects of its operation (e.g. file descriptor mapping) is similar to the operation of `LiveProcess`.

3.4 The GDB RSP Server in GEM5

GEM5 includes a GDB Remote Serial Protocol server to which a standard GDB (configured for cross-debugging the correct target architecture) can be attached to debug the software running in simulation. This is different both from debugging the simulator itself (which can be freely mixed with debugging the guest) and from attaching to a debug server that may be running inside simulation (the latter rarely makes much sense as it negates the benefits of simulation while serving the same purpose). The RSP server works the same way regardless of the simulation script (e.g., SE.py or FS.py) and of the SE/FS mode of the simulated CPU.

The RSP server provides access to the simulation within the capability meant by the GDB RSP protocol, but modifying the GEM5 source so that the RSP server supports functions not envisioned by GDB turns out to be relatively straightforward.

3.5 Pharo RSP Client

As part of this experiment, the test harness for the new JIT contains a Smalltalk RSP client comprising three layers.

Class `RemoteGDBTransfer` implements the RSP's *packet transfer* layer, approximately corresponding to Layer 6 of the OSI model. It opens a TCP connection to GEM5 via a Pharo `Socket` object. It then parses/synthesizes the RSP packet frame structure such as the "$"/"#" delimiters, checksums, special symbol escaping, distinguishing between 7- and 8-bit transfers, transmission of acknowledgements, and server interrupts.

Class `RemoteGDB` implements the Application layer of the RSP protocol (Layer 7 in OSI). The following is an example of a G packet exchange used to transfer the state of the processor's storage bases to the debugger:

Pharo ⟶ GEM5:

```
$g#67
```

Pharo ⟵ GEM5:

```
+
$00000000a6ffffbe0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000f0feffbe0000000050810000100000000000000000000000000000000000000000000000000000000000000000000000...0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000#f9
```

Pharo ⟶ GEM5:

```
+
```

This exchange is invoked by sending `#getRegisters` to a `RemoteGDB`. Other communications with the RSP server are initiated in a similar way.

This application layer of RSP necessarily contains parts specific not only to each ISA, but even to different processors within an ISA. For example, different processors in the PowerPC family may or not have registers `vr0`–`vr31` depending on whether the particular chip implements the AltiVec section of the Power ISA v2.03 spec. Accordingly, the G-packet will or will not reserve positions for the transmission of `vr0`–`vr31`.

The ARchC/ACCGen PDL descriptions do not contain such information. Fortunately, the implementation of this part of the wire protocol by GDB is not completely hand-coded, but parametrized by "processor feature" XML documents found under `gdb/features` within the GDB source. (There are, however, pieces which are not part of the XML but instead coded in C alongside the XML). The Pharo RSP

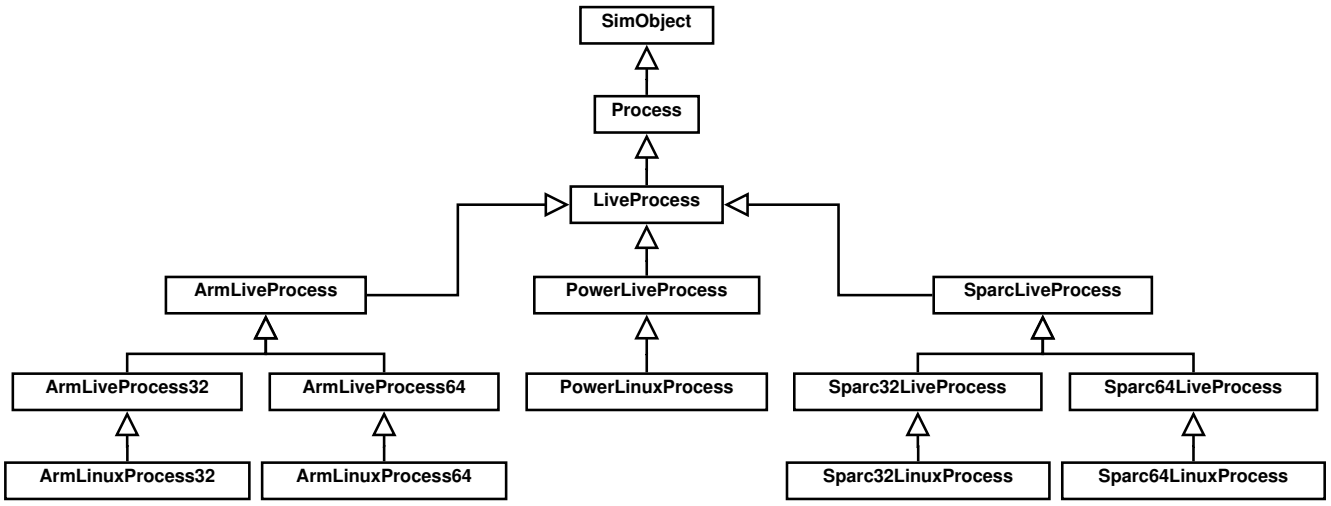

Figure 3. The LiveProcess Class Hierarchy

client uses PetitXml/PetitXPath to compute the processor's state bases wire transfer protocol from these feature descriptions.

3.6 Simulation Exits

Once the segment of n-code to be tested as part of the test suite is injected into the address space of the simulated process, the simulator needs to perform a delimited amount of simulation steps after which the resulting machine state can be verified. One approach to such delimiting consists of counting the instructions being executed. For certain processor models that take into account the details of the pipeline operation, such simple counting is impossible altogether. One approach to solving this problem is to use special markers through which the workload communicates with the simulator. Unfortunately, today's common ISAs are not designed with such software/processor communication as a goal. Therefore, a number of workarounds have been widely used by the simulation community. For example, Simics [1] resorts to "magic instructions" to mark specific points in the program text for the simulator to perform a "magic breakpoint". A magic instruction is an agreed-upon instruction considered extremely unusual. For example, on Intel i386, such an instruction is

```
xchg %bx, %bx
```

On SPARC,

```
sethi n, %g0
```

This device is only a workaround: on the one hand, it just happens that a whole operating system such as Linux does not contain the instruction, but there is no reason that a program *couldn't* contain this perfectly legal instruction; on the other hand, the flexibility of magic instructions as a software/simulator communication tool is limited.

The GEM5 SE mode uses the `exit(2)` syscall to terminate the simulation. This can be viewed as a case of normal execution: after all, this is the behavior of the unmodified program running on the physical microprocessor. However — in SE mode specifically — it can also be viewed as a case of software/simulator interaction, because instead of modeling the operation of the trap mechanism in the microprocessor (as it does in the FS mode) upon encountering the syscall interrupt instruction, the CPU model in SE mode branches into special "syscall emulation" code. This behavior is not part of the "live process" component but of the CPU model itself.

The Target-Agnostic JIT's test harness injects a syscall at the exit path(s) of the basic blocks emitted by the JIT in fine-granularity tests. (The JIT itself never emits syscalls; the latter are encoutered in the `libc` functions (let's ignore vDSO for simplicity) called from primitives, but not in the translated bytecode.)

In addition to being a simple marker, such syscall or interrupt instructions can serve as a two-way software/simulator communication mechanism. For example, they can transmit the elapsed simulated time ("ticks") from the simulator to either the outer or the inner Smalltalk (via RSP or placing the value into a simulated register, respectively).

4. Conclusions and Future Work

The experimental embodiment of the GEM5-based debugger built by the author proves it is possible to realize the repeatability, reversibility and other advantages offered by simulation in the context of gaining insight into the behavior of a Smalltalk VM.

The logical next research goal is to create a higher abstraction on top of the RSP client such that the simulated Smalltalk workload can be debugged at the level of the normal Smalltalk debugger. Indeed, the modular nature of the

simulator allows to introspect the workload's semantic constructs on the simulator side. In fact, such "context trackers" are what distinguishes simulation as an aid in understanding the software running under simulation as opposed to merely executing it (which is what a real machine does). A typical example are OS process trackers and context switchers which follow the state of the running OS's process (which is an abstract concept existing only within that guest OS).

A number of researchers, including this author, have built simulator modules for analysis of simulated execution of Smalltalk and Java runtimes at the VM constructs level [37, 41]. These modules can parse the object headers and bodies, walk over the Smalltalk stack, track the `ProcessorScheduler`'s manipulation of Smalltalk processes, etc., so that the programmer can debug the VM at a high level while enjoying the full benefits of repeatability and reversibility as well as detailed time and power measurements.

This allows the concept of liveness as defined within the framework of a "live Smalltalk system" to be generalized to a continuum of possibilities between the "image evolution" model of Smalltalk-80 and the "static program declaration" Algol-like model. In addition to the direct advantages to Smalltalk debugging such as repeatability and reversibility, it would then be intriguing to compare the properties of a high-level out-of-band [37] Smalltalk debugger based on the techniques proposed in this paper with those of remote debuggers based on e.g. distributed Smalltalk.

Future work on the debugee JIT itself will attempt to overcome the equation-theoretic deficiencies of the current rewrite engine. Further, the author intends to apply his system to serve as the implementation substrate for an actual Smalltalk. Modtalk [20] appears to be a singularly promising target for such an attempt, for a multitude of reasons. First, it allows its mechanical components (such as the proposed translator) to interact with a similarly deterministic mechanical surrounding environment. Second, it specifies the object layout, object memory access primitives, messaging, garbage collection and other mechanisms which are traditionally outside of the scope of the translator's concern, in a general formalism so that the same translator can also be used to derive the machine code implementing said mechanisms. Third, in traditional Smalltalk VMs some of these mechanisms rely on fundamentally untrustworthy implementation substrates. Removing this reliance may enable certain trustworthy-computing applications the reliability requirements of which are unsatisfiable with today's mainstream Smalltalk VM implememtations.

Acronyms

CPU Central Processing Unit.

ELF Executable and Linkable Format.

FS Full-System Simulation Mode.

FSS Full-System Simulation.

GDB GNU Debugger.

GPR General-Purpose Register.

I/O Input-Output.

IR Intermediate Representation.

ISA Instruction Set Architecture.

JIT Just-in-Time Translator.

MMU Memory Management Unit.

OS Operating System.

PDL Processor Description Language.

RSP Remote Serial Protocol.

RTL Register-Transfer Language.

SE Syscall-Emulation mode.

vDSO Virtual Dynamic Shared Object.

VM Virtual Machine.

References

[1] D. Aarno, J. Engblom. Software and System Development Using Virtual Platforms. Elsevier, 2015.

[2] A. V. Aho, S. C. Johnson. Optimal Code Generation for Expression Trees. J.ACM, Vol. 23, No. 3, 1976.

[3] A. V. Aho, M. Ganapathi, S. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. ACM Transactions on Programming Languages and Systems, 1989.

[4] M. Arnold, M. Hind, B Ryder. An Empirical Study of Selective Optimization. 13th International Workshop on Languages and Compilers for Parallel Computing.

[5] R. Auler, P. C. Centoducatte, E. Borin. ACCGen: An Automatic ArchC Compiler Generator. 24th International Symposium on Computer Architecture and High Performance Computing, New York, USA, 2012.

[6] R. Azevedo, et al. The ArchC Architecture Description Language. International Journal of Parallel Computing, 33(5): 453–484, October 2005.

[7] J. Bastian, S. Onder. Specification of Intel IA-32 Using an Architecture Description Language. IFIP TC-2 Workshop on Architecture Description Languages (WADL), Toulouse, France, 2004.

[8] J. A. Bergstra, J. W. Klop. Conditional Rewrite Rules: Confluence and Termination. Journal of Computer and System Sciences 32, 323–362 (1986)

[9] I. Bilicki, et al. Cache Line Reservation: Exploring a Scheme for Cache-Friendly Object Allocation. Proceedings of the 2009 conference of the Centre for Advanced Studies on Collaborative Research.

[10] N. L. Binkert, et al. The M5 Simulator: Modeling Networked Systems. IEEE Micro, Vol. 26, Issue 4, 2006.

[11] M. Bravenboer, E. Visser. Rewriting Strategies for Instruction Selection. Proceedings of the 13th International Conference on Rewriting Techniques and Applications. Springer, 2002.

[12] A. Butko, R. Garibotti, L. Ost, G. Sassatelli. Accuracy evaluation of GEM5 simulator system. IEEE, 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip, 2012.

[13] R. G. Cattell. Automatic Derivation of Code Generators from Machine Descriptions. ACM TOPLAS, Vol. 2, Issue 2, April 1980.

[14] M. H. M. Cheng, M. H. van Emden, D. Stott Parker. A Method for Implementing Equational Theories as Logic Programs.

[15] L. P. Deutsch, A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. ACM, 1983.

[16] M. Ganapathi, C. N. Fischer. Affix Grammar Driven Code Generation. ACM, 1985.

[17] M. Hohenauer, R. Leupers. C Compilers for ASIPs. Springer, 2010.

[18] F. A. Endo, D. Courouss, H.-P. Charles. Micro-architectural simulation of embedded core heterogeneity with gem5 and McPAT. Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation. ACM, New York, 2015.

[19] F. A. Endo, D. Courouss, H.-P. Charles. Micro-architectural simulation of in-order and out-of-order ARM microprocessors with gem5. IEEE, 2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV).

[20] J. Fridstrom, A. Jacques, K. Kilpela, J. Sarkela. Testing Modtalk. Lecture Notes in Business Information Processing, Agile Processes, in Software Engineering, and Extreme Programming — 16th International Conference, XP 2015 Helsinki, Finland, 2015.

[21] D. E. Knuth, P. B. Bendix. Simple word problems in universal algebras. In: Computational problems in abstract algebra. Pergamon, 1970.

[22] R. Leupers, P. Marwedel. Retargetable Code Generation Based on Structural Processor Descriptions. Kluwer, 1998.

[23] R. Leupers, O. Temam (eds.) Processor and System-on-Chip Simulation. Springer, 2010.

[24] P. Magnusson, B. Werner. Efficient Memory Simulation in Simics. Proceedings of the 28th Annual Simulation Symposium. IEEE, Phoenix, Arizona, USA, April 1995.

[25] M. M. K. Martin, et al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. ACM SIGARCH Computer Architecture News, Vol. 33, No. 4, September 2005.

[26] P. Marwedel, R. Leupers. Instruction Set Extraction from Programmable Structures. European Design Automation Conference, Grenoble, France, 1994.

[27] C. J. Mauer, M. D. Hill, D. A. Wood. Full-System Timing-First Simulation. Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems. 2002.

[28] E. Miranda. The Cog Blog. *http://www.mirandabanda.org/cogblog*

[29] E. Miranda. VisualWorks Threaded Interconnect. Smalltalk Solutions, New York, USA, 1999.

[30] E. Miranda. Context Management in VisualWorks 5i. OOPSLA, Denver, Colorado, USA, 1999.

[31] P. Mishra, N. Dutt (eds.) Processor Description Languages. Applications and Methodologies. Morgan Kaufmann Publishers, 2008.

[32] D. Stott Parker, M. H. M. Cheng, M. H. van Emden. A Prolog Technology Term Rewriter.

[33] D. Patterson. Future of Computer Architecture. Berkeley EECS Annual Research Symposium, 2006, Berkeley, California, USA

[34] J. B. Rosenberg. How Debuggers Work: Algorithms, Data Structures, and Architecture. Wiley Computer Publishing, 1996.

[35] S. H. Nikounia, S. Mohammadi. Gem5v: a modified gem5 for simulating virtualized systems. Springer, The Journal of Supercomputing, Vol. 71, Issue 4, 2015.

[36] R. N. Sanchez, et al. Using Machines to Learn Method-Specific Compilation Strategies. CGO'2011.

[37] B. Shingarov. Towards a Smalltalk VM for the 21st Century. 21st International Smalltalk Conference, Annecy, France, 2013.

[38] B. Shingarov. Modern Problems for the Smalltalk VM. International Workshop on Smalltalk Technologies, Cambridge, UK, 2014.

[39] B. Shingarov. The Synthesis of Target-Agnostic JIT. 8th Smalltalks — Argentina Conference, Córdoba, 2014.

[40] K. Vaswani, et al. Microarchitecture Sensitive Empirical Models for Compiler Optimizations. IEEE, International Symposium on Code Generation and Optimization, 2007.

[41] G. Wright et al. Introspection of a Java Virtual Machine under Simulation. SMLI TR-2006-159, Sun Microsystems, Calif., USA, 2006.

[42] G. Zimmermann. The MIMOLA design system: a computer aided digital processor design method. ACM, 1988.