Modern Problems for the Smalltalk VM

Boris Shingarov
boris@shingarov.com

Abstract

I propose an approach to managing new classes of Smalltalk VM complexity which emerged due to recent advances in technology, through the execution of the VM on formal models of the target processor. Three examples of this approach are discussed. First, I describe out-of-ISA-band observation of the VM based on full-system simulation in which the simulator is aware of the Smalltalk semantics. I also describe experiments in mechanical co-synthesis of the VM and the simulator from the same formal Processor Description Language, leading to an automatically-retargetable JIT. The major obstacle to the usefulness of this approach is the PDL's suitability for toolchain synthesis. Finally, an experimental attempt to bridge from hardware structure directly to JIT is discussed.

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

Keywords Smalltalk Virtual Machine, Full-System Simulation, Processor Description Language, Hardware-Software Codesign, Instruction Set Architecture

1. Introduction

Since the fundamental design of the major Smalltalk VMs has stabilized in the 1990s, the computing landscape has undergone radical evolution. The merge of technologies traditionally characteristic of high-performance computing with embedded computing, the System-on-Chip revolution, the end of the uniprocessor era — all these changes necessitate further evolution of the Smalltalk VM. One way to classify these changes is by position relative to the Smalltalk VM: the application workload we run on Smalltalk, vs. the computing platforms we run Smalltalk on. A shared characteristic of both is an increase, and a change in the nature, of complexity that the VM designer has to deal with.

Application workloads today often expose software defects related to multiprocessor/multicore parallelism, race conditions, or complex compiler optimizations; traditional source-level debugging techniques [Rosenberg'96] are ineffective in these situations and result in prohibitively high debugging effort. In-band observability aids (similar to DTrace) do help factoring the complexities involved in such defects to a limited extent.

The meaning of "performance" has been largely redefined with the end of the uniprocessor era. In traditional VM design, we mostly thought of "processor performance" in terms of sequential instructions-per-second, and of Smalltalk performance in terms of bytecodes-per-second and sends-per-second, n-code cache size-efficiency, and other metrics of similar nature. We now have very fast JITs making maximum use of the processor's ILP features, saturating the hardware in terms of bytecodes-per-second; very efficient PICs giving maximum sends-per-second; yet today's metrics of performance have changed, so we would consider trading sends-per-second for better power efficiency, as MIPJ (Million Instructions Per Joule) has become a more important performance characteristic than MIPS (Million Instructions Per Second). A typical Big Data customer today is concerned with the efficiency of the fundamental design of traditional Smalltalk object memory model, which imposes high FFI marshalling overhead and high d-cache miss rate due to high pointer dereference rate ("pointer-chasing") inherent in such object memory.

Modern architectures on which we run Smalltalk are undergoing a fundamental change which can be characterized as "the end of the uniprocessor era". Dave Patterson argues [Patterson'06] that the main challenges of today's computer architecture are the following "walls":

- Design cost wall
- Software legacy wall
- Power wall
- Memory wall
- ILP wall

These new classes of complexity mean the VM researcher has a new task before them: to create new observation strategies for getting insight into the Smalltalk VM. These mechanisms would allow us to regain "the clearness and ease of comprehension" (to use David Hilbert's expression) of the VM's behavior in today's complex post-uniprocessor computing environment; *"for what is clear and easily comprehended attracts, the complicated repels us"*.

2. Out-of-Band VM Observation

The Instruction Set Architecture (ISA) is the interface between software and the processor; when the program (for example, the Smalltalk VM) is running, this interface can be thought of as a communication channel between the program and the processor (one could imagine the instruction stream to be the program-to-processor direction of that channel, the other direction being the execution results: register values, flags, branching outcome, traps, etc.; we should also not forget about external signals flowing into the processor). Generally the mechanisms that we use to gain an understanding of what is happening in the software, can be classified by their positon relative to that communication channel as either In-Band or Out-of-Band.

Traditional debuggers [Rosenberg'96] are an example of In-Band mechanism. In-band mechanisms are inherently limited in what kind of information they can provide about the system being investigated. They also lack deterministic repeatability as well as reverse execution capabilities. This deficiency renders them incapable of aiding in understanding such important classes of situations as — to point out only one class — race conditions in parts of the system driven by asynchronous external events (e.g. network interrupts). In-band mechanisms are also destructive to the machine state (in other words, they are not truly transparent to the observed system). For example, hitting a hardware breakpoint will cause the execution of the breakpoint interrupt service routine and of the whole chain of context transitions, changing the state of the machine at the system level, and at a minimum, destroying the state of the memory cache hierarchy; in the end, analysis of e.g. timing-related phenomena becomes impossible under this class of debuggers.

In contrast, an Out-of-Band observation mechanism is any observation mechanism which does not work over the same ISA interface to the processor as the observed system. In the rest of this paper, I will talk about the following out-of-band mechanisms and my experiments to use them for better understanding of the Smalltalk VM:

- Functional simulation of the system at different levels of fidelity (instruction-accurate, cycle-accurate, etc.);
- Software simulation of a structural model of hardware;
- Custom-Instrumented FPGA models of hardware.

In section 6, I describe an experiment in which both the VM and (the simulator for) the machine on which the VM executes, are both automatically derived from the same processor description written in a formal language.

3. Full-System Simulation

Not to be confused with *emulation* (which focuses on mimicking the function of the emulated system; Bochs and QEMU are examples of processor emulators), system *simulation* is concerned with modeling the internal state of the simulated system. Simulation can rougly be generally classified into structural and behavioral. An extreme example of a structural model of a microprocessor is Michael Steil's transistor-level simulation of MOS 6502. Verilog simulation of the RTL-level hardware design of OpenSPARC T1 would be a more typical example of structural simulation.

Behavioral simulation is faced with trade-offs between completeness, accuracy, and efficiency. At the low side of the completeness scale are user-program-level simulators. At the opposite high side are full-system simulators. "Full-system" means that the simulation's functional fidelity is complete enough to run the whole operating system and applications unmodified and unaware of running in a simulation. Orthogonal to the measure of completeness is the measure of accuracy, i.e. regard of the model to certain aspects which do not affect the model's ability to execute the operating system quite fine in other aspects. The prime example of this concept is *timing accuracy*, roughly dividing simulators into *instruction-accurate* and *cycle-accurate*. Accuracy comes at the price of efficiency. Using more precise terms, in SystemC TLM language, the lowest level of timing accuracy is *software timing*: the time unit is one instruction, and the timing of memory hierarchy is not modelled at all (nor do these software-timed simulations model other microarchitectural time aspects such as out-of-order execution). Virtutech Simics in native mode and IBM CECsim use this level of time model accuracy.

Loose Timing and Approximate Timing are the next two levels of time modeling. These more accurate models come with progressively higher performance cost. In recent years, there has been a proliferation of full-system simulators offering extremely accurate microarchitechure-level models. Taken together, the Simics Microarchitectural Interface, Flexus, GEMS [Martin'05], Opal (formerly called TFSim [Mauer'02]), M5 [Leupers'10], GEM5 [Binkert'11], present a rich variety of cycle-accurate simulation approaches and provide models for a wide selection of ISAs.

4. Smalltalk-Aware System Simulation

Today's prevalent full-system simulators allow complete programmatic access to all aspects of the simulation. The fundamental event model, the processor, the memory hierarchy, and the peripheral devices are all modelled by loosely-coupled modules interacting via open APIs. These APIs allow automatic detailed analysis of the simulated execution. There are numerous modules available today providing full symbolic debugging of C programs, deep awareness of various operating systems' interal functionality, analysis of execution of network stacks, etc.

Wright et al. [Wright'06] use a similar approach for out-of-band introspection of the HotSpot JVM. Using full-system simulation of the SPARC processor, they were able to gain significant new insight into the interaction of the VM

with the memory hierarchy, and in particular into the effect of GC and n-method recompilation on cache efficiency.

In that light, full-system simulation appeared like a promising approach to gain new understanding of the Smalltalk VM. In one experiment, I created a module which makes the simulator aware of the semantics of execution of the Cog JIT VM. Because at the time of these experiments Cog JIT only ran on IA32, the FSS system selected for this experiment was Simics simulating an in-order Intel x86 processor running a stock "Tango" target (a Fedora Core Linux). The module introspects into the receiver object at any instruction when the processor is executing n-code. This is done using the simulator's Python API (for exploration, it felt more "immediate" than the also-available C API). First, the receiver oop, which in IA32 Cog JIT is kept in the `%EDX` register, is obtained from the state of the simulated processor:

```
oop = conf.cpu0.edx
```

The following simple function then does some reasoning about the object:

```
def print_class_of_oop(oop):
  if ((oop & 1)==1):
    print "SmallInteger"
  else:
    headerType = smalltalk_headerType(oop)
    if (headerType==3):
      print "...looks like compact class..."
    else:
      word2 = read_virt_value(oop-4, 4)
      classOop = word2&0xFFFFFFFC
      print "class oop: ", hex(classOop)
      clsNameOop = read_virt_value(classOop+32, 4)
      print "class name oop: ", hex(clsNameOop)
      str=""
      for offset in range(smalltalk_objByteSize(clsNameOop)):
        str += "%c" %
             read_virt_value(clsNameOop + 4 + offset, 1)
      print str
```

First we look at the tag bit to see if the oop is a pointer or a SmallInteger. A SmallInteger does not leave much to further introspection. With a pointer, we dereference it to access the object header:

```
def smalltalk_headerType(oop):
  return read_virt_value(oop, 4) & 3
```

where memory is accessed like this:

```
# Read a little-endian value from a physical address.
def read_phys_value(paddr, len):
  cpu = conf.cpu0
  val = 0L
  for i in range(len-1, -1, -1):
    try:
        val = (val<8) | SIM_read_phys_memory(cpu, paddr+i, 1)
    except:
        raise PTrackError("%s can not read byte at p:0x%x"
                          % (cpu.name, paddr+i))
  return val

# Read a little-endian value from a virtual address.
def read_virt_value(vaddr, len):
  cpu = conf.cpu0
  try:
    paddr = SIM_logical_to_physical(cpu, Sim_DI_Data, vaddr)
  except:
    raise PTrackError("%s can not translate v:0x%x to physical"
                      % (cpu.name, vaddr))
  return read_phys_value(paddr, len)
```

(The simulator interprets the state of the simulated MMU and performs address translation, modeling the details of the processor's TLB in `SIM_logical_to_physical()`). The crucial difference between this simulator-side read and printing a memory value in a traditional debugger is that the simulator-side read (and in fact, none of the operations in this module) does not disturb the state of the processor. If the simulation includes some sort of cycle-accurate model of time (e.g. a model of the memory hierarchy), oop dereferencing will not affect the state of that model. This is in contrast with the situation where the JIT dereferences the oop: such debugee-side access will cause simulated cache misses, pipeline stalls, or whatever other possible effects of the "`load`" instruction are modelled at the present accuracy level (cf. next section).

The simplest way to try the introspection module is to stop the simulator at a "magic breakpoint" in the middle of n-code. We insert a no-effect instruction into the emitted n-code. This "magic instruction" should not only have no effect but also be extremely unlikely to occur in real code. For example, on IA32 a usual candidate is

```
xchg %bx, %bx   # cf. 16r66 below
```

(To make the Cog JIT emit the magic instruction, first an abstract RTL instruction and its IA32 concretization are defined. The abstract instruction is added at the end of the list in `CogRTLOpcodes>>initialize`, and `#initialize` is resent. We also need to add a new method:

```
Cogit>>Magic
  <inline:true>
  <returnTypeC:#'AbstractInstruction*'>
  ^self gen: Magic
```

The IA32 concretization amounts to changing

```
CogIA32Compiler>>dispatchConcretize
  opcode caseOf: {
    ...
    [Magic] -> [^self concretizeMagic].
  }
```

and specifying the instruction encoding by adding this new method:

```
CogIA32Compiler>>concretizeMagic
   "Will get inlined into concretizeAt: switch."
   <inline: true>
   machineCode
      at: 0 put: 16r66;
      at: 1 put: 16r87;
      at: 2 put: 16rdb.
   ^machineCodeSize := 3
```

Once the new instruction is defined, we can modify the JIT to emit it. For illustration purposes, I modified

```
genGetClassFormatOfNonInt: instReg
               into: destReg
               scratchReg: scratchReg
  "Fetch the instance's class format into destReg,
   assuming the object is non-int."
  | jumpCompact jumpGotClass |
  <var: #jumpCompact type: #'AbstractInstruction *'>
  <var: #jumpGotClass type: #'AbstractInstruction *'>
  cogit Magic.  "THIS WILL STOP SIMULATION"
```

```
"Get header word in destReg"
cogit MoveMw: 0 r: instReg R: destReg.
... "rest of method"
```

The VM is regenerated and recompiled after these changes.)

After magic instruction support is installed in the JIT, the simulator is instructed to run the simulation until it encounters the magic instruction; with the simulator paused, our `print_smalltalk_receiver()` function can be invoked from the simulator's command-line interface.

In a more practical scenario, various routines in the Smalltalk-aware module can be driven by callbacks from the simulator. One useful application is collecting statistics. If the simulation includes modeling of microarchitectural details, this technique can provide information about relationships between high-level Smalltalk abstractions and the microarchitectural phenomena: for example, it would be possible to make queries such as "cache hit ratio when performing linked sends, but only when the receiver `isKindOf:` this specified class".

Even more interesting is stopping simulation and returning control to the simulator interface when certain programmatically specified conditions are met. The condition can be as simple as segmentation fault. Quite often in today's parallel computing environments, a bug could be caused by an extremely rare race condition in an asynchronous interrupt-triggered routine, and not reproducible in staged forward execution. This class of problems is extremely well-suited to debugging in simulation. We record some number of last simulation steps in a ring buffer. We run the simulation until the VM crashes. At that point, we can analyze the execution steps that led to the crash (performing the Smalltalk-aware introspection available to us at any step).

5. Deriving a JIT from Processor Description Language

A number of Smalltalk VM implementations have attempted various degrees of retargetability. While the original "PS" VM [Deutsch'83] strictly targeted Motorola 68000, its successor HPS [Miranda:Contexts], [Miranda:Thread] uses the C macro-processor to achieve a degree of portability by providing a "processor definition file" containing an agreed-upon set of "`#define`" C macro definitions. The Cog VM [Miranda:Cog], written in a subset of Smalltalk, contains abstract classes which represent a common concept of what a "processor" can be ("abstract RTL"); these are subclassed to describe concrete target ISAs (such as IA32 or a particular ARM variant).

Being embedded in the VM implementation language and relatively low-level, such ad-hoc processor description facilities require the human implementor of the port to comprehend the details of the processor/platform specifications and manually program the mapping to the "common machine", working in that implementation language. One limiting factor of this approach is the space of processors which can be parametrized given a particular "common machine": often, the set of considerations relevant to a new processor architecture can not be expressed in terms of the existing "common model". Thus, there is a trade-off between specificity (because we need to be able to derive a translator from the processor description) and generality (to cover a wide enough family of processors).

An even more important factor is the complexity of today's architectures and OS platforms. As an example, the ARM Architecture Reference Manual is 5158 pages (as of the ARMv8 "A.a" issue) of natural English language. Moreover, the information in the ARMARM alone is not enough: a VM port would need to take into account considerations of performance optimization, details of the ABI on the platform, etc. The effort on the part of the author of the port to comprehend all this complexity, makes the cost per processor port prohibitive even in the case of universal processor ISAs. The port author is trapped in a cycle of re-interpretation of the natural language of the specifications, as details of his interpretation differ from the processor designer's (ISA implementor's). This kind of porting effort becomes completely unrealistic in today's chip-level-integration systems-on-chip often involving application-specific instruction sets, which are quickly growing to become the norm in embedded applications.

Processor Description Languages [Mishra'08] have been used for automatic synthesis of compilers from formal specifications of processors in parallel with synthesis of hardware, thus eliminating or reducing the problem of software/hardware ISA interpretation discrepancy mentioned above. Pioneering work in PDLs in the 1970s was Gerhard Zimmermann and Peter Marwedel's MIMOLA (Machine Independent Microprogramming Language). Since then, a number of processor description languages and technologies have been developed, such as nML (the processor modeling language in Synopsys' "IP Designer" system), LISA from RWTH Aachen University, EXPRESSION, ArchC, and many others. I chose the open-source ArchC language and system [Azevedo'05], [Mishra'08] as the base for my experiments with deriving a Smalltalk VM from a PDL.

The ArchC system takes a processor description (written in the ArchC language) and generates a simulator for the processor. In addition, the ACCGen compiler generator [Auler'12] takes an ArchC processor description and generates an LLVM "llc" backend.

My experiments with deriving a JIT from ArchC PDL did not involve the LLVM infrastructure. Also, in the proof-of-concept I did not go beyond a trivial mockup of the Cattell algorithm [Cattell'80]; in other words, no optimization was attempted as my focus was on the automatic/unsupervised retargetability and the guarantees it gives against the hardware/software ISA interpretation discrepancies mentioned above.

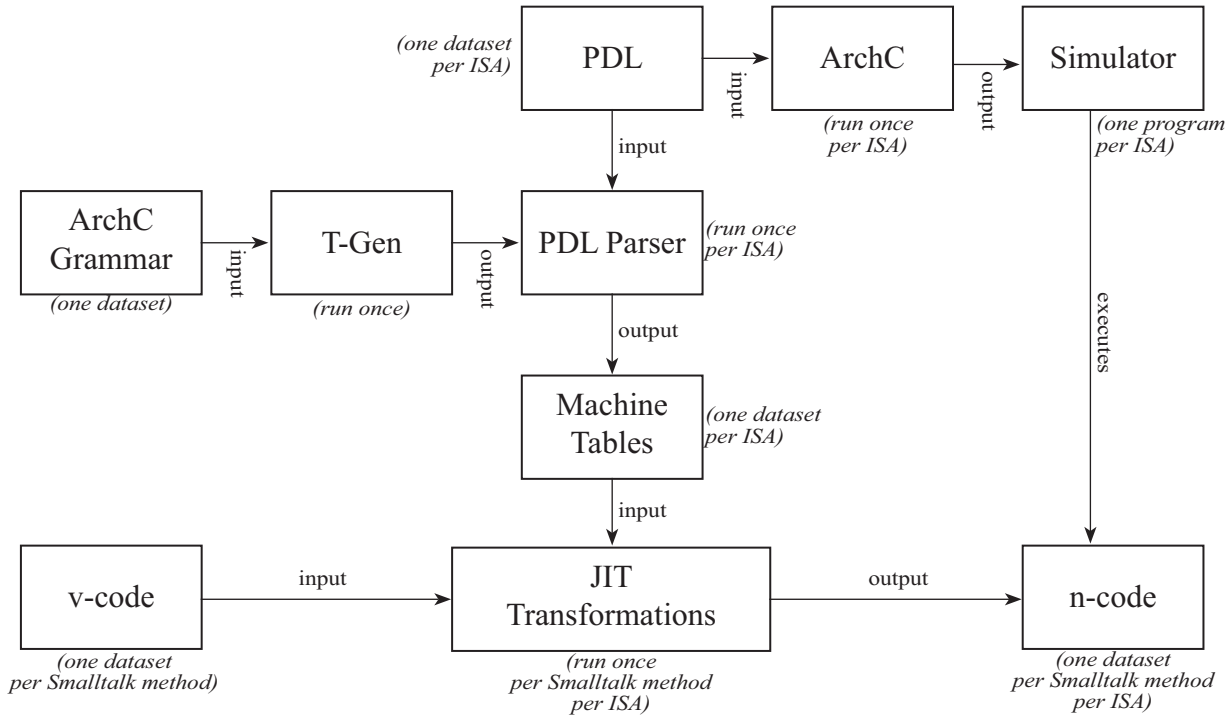

Figure 1. Deriving a JIT from PDL

Figure 1 shows the basic structure of the JIT generator. The same PDL specification of the processor is input to both the ArchC system and the JIT generator. ArchC generates a system simulator allowing out-of-band debugging of the JIT, in line with ideas described in sections 3–5 of this paper. The JIT generator is implemented as a Smalltalk program. With the aid of the T-Gen parser generator, it re-uses ArchC's and ACCGen's PDL grammars. The parser produced by T-Gen, transforms the processor description into Machine Tables which parametrize the Cattell algorithm [Cattell'80]. The Machine Table is a set of productions, and Cattell's algorithm searches for a suitable chain of rewrites leading from the IR goal on the left ("TCOL" in Cattell's paper) to instruction terminals on the right. The experimental Smalltalk JIT is much simpler. While the general Cattell algorithm allows arbitrary and possibly recursive rewrites, my JIT's derivations always consist of two fixed productions: (1) *bytecode ::= RtlInstruction**, and (2) *RtlInstruction ::= MachineInstruction**. The search degenerates into a lookup in the table. In this sense, the JIT is equivalent to any well-known JIT such as Cog or HPS: the difference is not in what code generation it does, but in how it is parametrized automatically from a formal specification rather than manually by the VM programmer. However, with the concept of a Smalltalk JIT derived from PDL now proven possible, there is no reason, in principle, why this approach would not work with more advanced forms of code optimization.

Achieving the derived JIT met with two primary difficulties: (1) the presence of algorithmic definitions in the ArchC PDL, and (2) formal specification of platform ABI.

Extraction of instruction semantics from the ArchC processor description. ArchC's primary focus is on synthesis of efficient simulators. To facilitate this goal, ArchC allows algorithmic expression of instruction semantics. Auler et al. [Auler'12] give the following example of how an instruction behavior may be specified in custom C++ within the PDL model:

```
void ac_behavior( add )
{
  RB[rd] = RB[rs] + RB[rt];
};
```

This approach of being able to specify processor behavior as an algorithm expressed in a universal programming language, favors the point of view of simulation over that of both hardware synthesis and compiler generation. Several authors, including Marwedel and Leupers [Marwedel'94], proposed various approaches to the problem of instruction semantics extraction for the purpose of instruction selection. These approaches generally do not work for ArchC. After all, due to the halting problem it is not even possible to decide, given such algorithmic description of an instruction, whether the instruction's execution will terminate at all, let alone to compute the information necessary for instruction selection. To solve this problem for their C complier, Auler

et al. [Auler'12] propose an RTL-based extension to ArchC. Existing ArchC processor models need to be amended to include this instruction-semantics information. Their ACCGen compiler demonstrates that such amendment does in fact allow to synthesize a working C compiler. My Smalltalk JIT reuses Auler's extensions to the PDL language and the extended models of ARM, SPARCv8, PowerPC and MIPS which are provided with the ArchC distribution.

The major drawback of this approach is that there are essentially two processor definitions (declarative and procedural) and we are again facing the same problem of diverging ISA interpretations.

FFI Glue is the custom machine code that connects the abstract computation machinery of the JIT to the rest of the system which is written in C. Its crucial role is that it's the way for computation to produce effect. The main issue the glue has to deal with, is the platform ABI convention about the C stack which the glue has to synthesize and manage. Most of the time, PDL models don't include an ABI specification. After all, in a full-system simulation such ABI model is altogether not needed because ABI is a software-level concern: it exists entirely within the software system running *on top* of the simulator.

ACCGen partially addresses this problem by extending ArchC to describe calling conventions; however, only their ARM model contains such ABI descriptions.

In light of all this, a production PDL-derived JIT for Smalltalk may need a different substrate for translator synthesis. Ideally, the same processor description would be used to synthesize the hardware, the compiler toolchain (Smalltalk VM included) and the simulator. Machine-readable ABI specs do not need to be integral part of the processor description, but will have to be the accepted starting point of the synthesis of the C toolchain, lest calling convention interpretation discrepancies lead to ABI violations analogous to the hardware-software discrepancy bugs discussed above.

6. Introducing Smalltalk Awareness into Structural Models

In the ideal scenario, the final destiny of a VM synthesized from a high-level processor description, would be running on hardware structures synthesized from the same description (after being debugged in simulation derived from the same description). At the time of this writing, I am not aware of a PDL framework openly available and suitable for such end-to-end experiments with Smalltalk. There is no doubt it will become available in the future. How can we attempt to bridge processor structure and the VM before we have such an end-to-end framework in our hands?

A number of processor instrumentation approaches has been described in the literature [Stollon'11]. EJTAG is a processor debugging facility very specific to MIPS. ARM has a comparable instrumentation system called ETM. I am not aware of open implementations of either EJTAG or ETM,

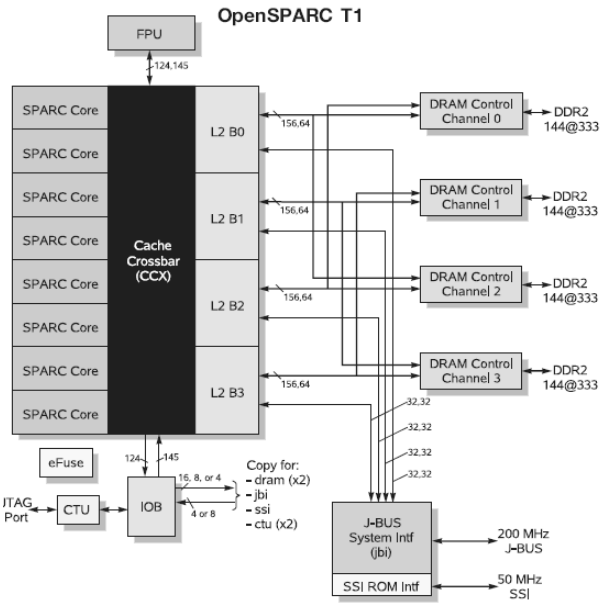

Figure 2. Block Diagram of OpenSPARC T1 [Weaver'08]

and it is not clear how either can be used as the foundation for a research effort to implement Smalltalk-aware processor instrumentation.

Therefore, in one experiment, I attempted to use a hardware description of a complete processor as the starting point. In recent years, open-source processor and system-on-chip IP has matured to a point of significant practical importance in critical production systems. The experiment described here starts from the Verilog source code for the OpenSPARC T1 microprocessor available under the GNU General Public License. It might be possible, using approaches discussed by Marwedel and Leupers [Marwedel'94], to extract enough ISA semantics from this structural description to be able to synthesize a Smalltalk VM, thus closing the hardware-software circle. I have not attempted this yet. Instead, in this experiment, the hardware structure is extended with probes which afford access to the internal state of the hardware at points of interest to the VM researcher and automatic analysis of that state in accord with the programmatic structure of the Smalltalk VM.

The OpenSPARC microprocessor consists of a designer-variable number of processor cores, connected to the memory cache, FPU and other components by the Cache Crossbar (CCX). Figure 2, reproduced from [Weaver'08], shows a block diagram of OpenSPARC T1. In a reference implementation of OpenSPARC on FPGA [Thatcher'08], only the processor core is synthesized from Verilog to the FPGA's programmable logic. The CCX, as well as everything else on the other side of it across from the core (memory interface, FPU, etc., and also some system components which would be off-chip in an ASIC implementation — e.g. Ethernet MAC; I will call the total of these components "the off-core"), are

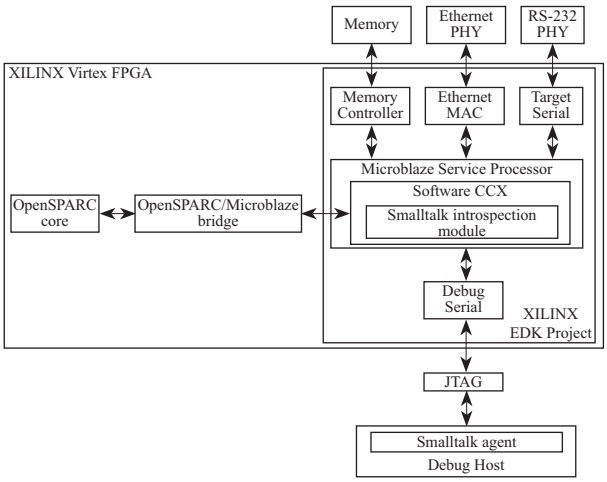

Figure 3. OpenSPARC Implementation on XILINX Virtex FPGA with Smalltalk-Aware Crossbar. Cf. the corresponding diagram in [Thatcher'08], NB the position of the Smalltalk module.

implemented in a XILINX EDK design. Although ultimately both the OpenSPARC core and the EDK project are physically sharing the same programmable logic fabric, the Microblaze service processor, the Ethernet MAC, the memory controller etc. are just standard XILINX IP, and the "off-core" functions are programmed in software running on the Microblaze. This is what facilitates the Smalltalk-aware probe. The off-core code (written in C) is amended with a Smalltalk observation module. The module is controlled by an agent program running on the debug host via the standard Microblaze debug serial port. The module (and the agent through it) have full access to the state of the OpenSPARC, and can reason about Smalltalk objects and VM program-structures similarly to the Smalltalk-aware FSS discussed in Section 4.

Experimenting with this configuration suggests even more importance of deriving hardware and software from a common higher-level processor description (as in section 5 above). Indeed, even the reference implementation of this mainstream microprocessor contains two embodiments (ASIC- and FPGA-oriented) which are not derivative of each other, and hence potentially diverging interpretations of the SPARCv9 ISA.

7. Conclusion

Out-of-band observation techniques can provide deeper insight into crucial aspects of the Smalltalk VM than possible with traditional observation approaches. In particular, this strategy can shed new light on the VM's use of time and power. A JIT can be written in a processor-agnostic manner. Targeting such JIT at a new ISA is a matter of automatically processing the new architecture's PDL description. The same PDL description can also be used for the synthesis of the actual processor, as well as of the simulators for the out-of-band observation, thus eliminating the hardware/software mismatch as a major source of software defects.

References

[Auler'12] 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.

[Azevedo'05] R. Azevedo, et al. The ArchC Architecture Description Language. International Journal of Parallel Computing, 33(5): 453–484, October 2005.

[Binkert'11] N. Binkert, et al. The GEM5 simulator. ACM SIGARCH Computer Architecture News, Vol.39(2), May 2011.

[Cattell'80] R. G. Cattell. Automatic Derivation of Code Generators from Machine Descriptions. ACM TOPLAS, Vol. 2, Issue 2, April 1980.

[Deutsch'83] L. P. Deutsch, A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. ACM, 1983.

[Hohenauer'10] M. Hohenauer, R. Leupers. C Compilers for ASIPs. Springer, 2010.

[Leupers'10] R. Leupers, O. Temam (eds.) Processor and System-on-Chip Simulation. Springer, 2010.

[Magnusson'95] P. Magnusson, B. Werner. Efficient Memory Simulation in Simics. Proceedings of the 28th Annual Simulation Symposium. IEEE, Phoenix, AZ, USA, April 1995.

[Marwedel'94] P. Marwedel, R. Leupers. Instruction Set Extraction from Programmable Structures. European Design Automation Conference, Grenoble, France, 1994.

[Martin'05] M. M. K. Martin, et al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. Computer Architecture News, September 2005.

[Mauer'02] 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.

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

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

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

[Mishra'08] P. Mishra, N.Dutt (eds.) Processor Description Languages. Applications and Methodologies. Morgan Kaufmann Publishers, 2008.

[Patterson'06] D. Patterson. Future of Computer Architecture. Berkeley EECS Annual Research Symposium, 2006, Berkeley, Calif., USA

[Rosenberg'96] J. B. Rosenberg. How Debuggers Work: Algorithms, Data Structures, and Architecture. Wiley Computer Publishing, 1996.

[Stollon'11] N. Stollon. On-Chip Instrumentation. Springer, 2011.

[Thatcher'08] T. Thatcher, P. Hartke. OpenSPARC T1 on Xilinx FPGAs. RAMP Retreat, Stanford, August 2008.

[Weaver'08] G. L. Weaver (ed.) OpenSPARC Internals — OpenSPARC T1/T2 CMT Throughput Computing. Sun Microsystems, Calif., USA, 2008.

[Wright'06] G. Wright et al. Introspection of a Java Virtual Machine under Simulation. SMLI TR-2006-159, Sun Microsystems, Calif., USA, 2006.