The SOM Family

Virtual Machines for Teaching and Research

Michael Haupt[1] Robert Hirschfeld[1] Tobias Pape[1] Gregor Gabrysiak[1]
Stefan Marr[2] Arne Bergmann[1] Arvid Heise[1] Matthias Kleine[1] Robert Krahn[1]

[1] Software Architecture Group, Hasso-Plattner-Institut, University of Potsdam, Germany
[2] Software Languages Lab, Vrije Universiteit Brussel, Belgium
michael.haupt@hpi.uni-potsdam.de, hirschfeld@hpi.uni-potsdam.de, stefan.marr@vub.ac.be
{firstname.lastname}@student.hpi.uni-potsdam.de

ABSTRACT

This paper introduces the SOM (Simple Object Machine) family of virtual machine (VM) implementations, a collection of VMs for the same Smalltalk dialect addressing students at different levels of expertise. Starting from a Java-based implementation, several ports of the VM to different programming languages have been developed and put to successful use in teaching at both undergraduate and graduate levels since 2006. Moreover, the VMs have been used in various research projects. The paper documents the rationale behind each of the SOM VMs and results that have been achieved in teaching and research.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education, Information systems education; D.3.4 [**Processors**]: Interpreters, Memory management, Optimization, Run-time environments; D.2.11 [**Software Architectures**]: Domain-specific architectures

General Terms

Design, documentation, languages

Keywords

Virtual machine, teaching, research, architecture.

1. INTRODUCTION

The increasing importance of virtual execution environments (*virtual machines*, VMs), for programming-language execution indicates that it is not enough to teach the languages alone. Maybe counter-intuitively, the relevance of knowledge about the inner workings of the underlying VMs grows—in spite of the promise that employing a hosted language allows for ignoring details of its implementation. The idea that the VM will take care of it all is naïve: for instance, literature on Java idioms devotes numerous pages to describing performance impacts of using synchronization [5].

In the light of this, it is apparent that VMs are an important topic of teaching in computer science. A solid knowledge base of VM implementations is helpful, especially given that the topics one will inevitably touch when dealing with the matter include aspects of systems programming that are of broader interest, such as memory management, synchronisation and locking, compilers, and (adaptive) optimisation.

Admittedly, an operating systems curriculum will likewise treat most of these topics, but addressing them from the point of view of programming language implementation constitutes an alternative angle, putting *programmer interest* in perspective. Few computer science students will eventually develop operating systems, but many, if not most of them will use hosted programming languages, so learning about their implementations seems to be a good choice.

Teaching VMs faces two problems. On the one hand, the time available to students in any given particular course is limited—typically, there are other courses to work for. On the other hand, VM implementations tend to be rather complex, so that one typically cannot expect students to deeply understand and appreciate, not to say *extend* (e. g., in fulfilment of coursework assignments) a large body of source code given the limited timeframe mentioned above.

That said, a certain skill level in low-level programming should be expectable from students who enrol for a course on VM implementations. Therefore, a tool for teaching VMs—i. e., a VM used as a model and for extensions—does not have to be *simplistic*, but it should be *accessible*. This core requirement has three parts, namely *simplicity*, *openness*, and *adequacy*, which will be detailed below.

This paper describes a family of VM implementations[1] hosting the same language (a Smalltalk [7] dialect) that fulfils the accessibility requirement. The *SOM family* (Simple Object Machine) today has four members implemented in different programming languages. All of these VMs can be applied in teaching; some at undergraduate, some at graduate levels. Moreover, they have proven to be viable tools for graduate, doctoral, and post-doctoral research.

[1]The entire family is available from the project home page at `www.hpi.uni-potsdam.de/swa/projects/som/`.

ITiCSE'10, June 26–30, 2010, Bilkent, Ankara, Turkey.

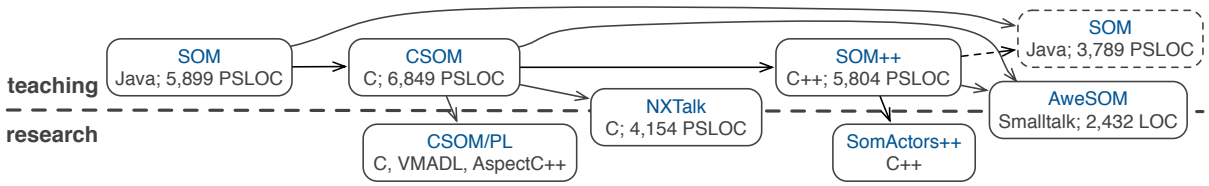

Figure 1: The SOM Family and genealogy.

An overview of the entire SOM family is given in Fig. 1; the figure separates the teaching VMs from research projects (some are hybrids) and also mentions their sizes (in *physical source lines of code*, PSLOC[2] [15]) for comparison. The family members, and results achieved using them, will be described in the next section after detailing the requirements imposed on VM implementations used in teaching. In Sec. 3, the design of the VM course in which SOM family members were used and developed is described in detail; also, results of students' evaluation of the course are given. Sec. 4 wraps up the paper and outlines future plans with the SOM family.

2. INTRODUCING THE SOM FAMILY

The accessibility requirement breaks down into three requirements. *Simplicity* refers to both the hosted language or virtual instruction set architecture (ISA) and the VM architecture. The former should not be overly complicated and should allow a quick start in understanding the VM's inner workings based on tracing how the hosted language's semantics is implemented. The latter should be comprehensible and use clean abstractions. Next, *openness* means that the VM should be ready for use as well as open for extension, and that documentation should be available to provide answers to at least the most common questions. The final requirement, *adequacy*, addresses the way a VM appropriately provides low-level structures and typical VM mechanisms. In brief, a teaching VM should be an *enabler* for the curious, not getting in their way too much.

This section first wraps up results of the evaluation of some VMs that were available at the time SOM was ultimately chosen. The ensuing sections 2.2 through 2.5 each present one particular member of the SOM family, and the ways in which they were put to use in teaching; coursework results are listed. Sec. 2.6 briefly summarises research projects building on particular SOM VMs.

2.1 Exploring Virtual Machines

At the time the choice of VM had to be made, two VMs were closely investigated. The Jikes Research Virtual Machine[3] [1, 2] is a Java VM implemented in Java, exceptionally well documented, and an excellent research platform. Due to the complexity of the Java ISA, it is however not *simple*. Since it is intended to be a research platform for high-performance server applications, its source code is highly optimised and thus not easily accessible.

The Squeak VM[4] [17] supports the simple Smalltalk ISA [7]. As it is implemented in a stripped-down Smalltalk dialect, the VM is entirely visible and observable from within a Smalltalk environment. Documentation is scarce, however, and the Smalltalk dialect mentioned above is actually a variant of C with Smalltalk-like syntax. In summary, the Squeak VM is far from being accessible.

Other approaches tailored to teaching VMs are Javy [8] and Jack [14]. *Javy* provides a 3D environment in which a software agent, represented by an avatar, emulates a Java VM. Javy concentrates on conveying details about the Java bytecode instruction set but does not touch upon actual VM implementation aspects. The *Jack* VM is one part in a complete software stack intended to teach all aspects of computer architecture, ranging from the processor to user applications. The Jack ISA is very simple and does not provide object-oriented abstractions.

2.2 SOM

Foraging for VM implementations suitable for teaching led to the eventual discovery of SOM (Simple Object Machine). SOM had been implemented in the Java programming language at the University of Århus, Denmark, and had been applied there in a course on object-oriented VMs in 2002.

Its features made SOM an excellent choice. First of all, it was implemented in a mainstream high-level programming language (Java), good knowledge of which could safely be assumed in students at both undergraduate and graduate levels. Next, its implementation applied object-oriented abstraction sensibly—employing interfaces and classes—leading to an overall very comprehensible architecture. Finally, it came with a set of tests and benchmarks that could be used to assess modifications to its implementation.

SOM's Smalltalk ISA consists of only 16 bytecode instructions; they are executed by a simple switch/case interpreter. The compiler translating Smalltalk code to bytecodes is implemented as an abstract syntax tree (AST) traversal; the AST is generated by an ANTLR[5] parser.

The SOM authors (see Acknowledgements) kindly contributed the source code and permitted its further use under the terms and conditions of the MIT license[6].

In this form, SOM was applied in an undergratuate course on compilers and VMs at Lancaster University, United Kingdom, and in a graduate course on VMs at Technische Universität Darmstadt, Germany. Students were given various small-to-medium-scale coursework assignments, such as implementing integer canonicalisation (for boxed integers: exactly one box per number), verbose stack traces for "does not understand" messages [7], and a bytecode manipulation framework. Another programming assignment was to implement a mark/sweep memory manager—a strange thing to do when the implementation language is Java and features a garbage collector anyway.

[2] Lines of code, excluding comments and blank lines.
[3] `jikesrvm.org`
[4] `squeakvm.org`
[5] `antlr.org`
[6] `www.opensource.org/licenses/mit-license.php`

In 2009, SOM was subject to a major revision, during which the Smalltalk compiler was replaced with the one contained in SOM++ (see below in Sec. 2.4), which was backported from C++ to Java. The SOM source code was also updated to Java 6, making use of the available modern language features such as generics and enumerable types. This new version of SOM is 3,789 PSLOC large and thus considerably smaller than the original.

2.3 CSOM

Accepting the inadequacy of letting students implement a low-level tool (a garbage collector) for a VM whose implementation actually relied on the memory management facilities of a high-level language (Java) led to the decision to port SOM to the C programming language. This choice was further fortified because C provides natural access to low-level structures and thus allows for conducting many more interesting VM-related implementation experiments.

The port, resulting in CSOM, was subject to some requirements. The object-oriented internal structure of SOM was to be retained, as well as compatibility with pre-existing SOM Smalltalk applications, and accessibility should be preserved as much as possible.

Retaining the object-oriented architecture required OOP emulation in C. Virtual method tables were now managed by hand, and all message sends were handled by a `SEND` macro. The Smalltalk compiler could be easily ported, as a C backend for the ANTLR parser generator became available at the time of porting. CSOM did not have a memory manager—this was left for an exercise. The port was achieved by one skilled undergraduate student in six months time.

This first version of CSOM was applied in a graduate course on VMs in 2007. Students worked in groups of 2–3 members and were given one large-scale programming assignment per group. They implemented mark/sweep and reference counting garbage collectors, native and green[7] multithreading support, and just-in-time compilers.

After the course, CSOM was assessed. At the time, it consisted of no less than 15,232 PSLOC, 11,324 of which belonged to the Smalltalk compiler: too much code was spent on too little actual functionality. The compiler, first translating Smalltalk source code into an AST before generating bytecodes, was also too complicated given the simplicity of the SOM bytecode. Memory management was nonexistent. The entire VM source code was also not well structured (there was not even a proper directory structure). In a nutshell, CSOM was not as accessible as intended.

To improve on this, CSOM underwent a massive refactoring and partial reimplementation. The source code was organised into logical modules, which were mapped to corresponding directories. The robust mark/sweep garbage collector from the past coursework assignment was included as the default memory manager and enriched by statistics facilities. The Smalltalk compiler was completely rewritten and ANTLR abandoned; instead, a recursive-descent parser was written by hand. This resulted in a reduction in lines of code by 55.5 % to 6,782. The compiler alone shrunk by 86.4 %, amounting to a mere 1,535 PSLOC.

This new version of CSOM was employed in the same course in 2008, where students implemented Smalltalk virtual images (as opposed to only reading Smalltalk source files upon startup) [7], threaded interpretation (to speed up execution) [4], and tagged integers (to save memory for representing small integral numbers) [7].

[7] Green (user-level) multithreading does not rely on OS thread support, but implements scheduling at VM level.

2.4 SOM++

Employing CSOM in teaching in two subsequent years, it was observed that especially the OOP emulation unnecessarily bloated the VM's source code. This led to the decision to port CSOM to a "low-level" OOP language, namely C++. This resulted in SOM++, a member of the growing SOM family exploiting the facilities offered by C++ where possible and sensible. That is, the standard template library was used to manage data structures, operator overloading was used to design more convenient interfaces, and so forth.

One graduate student achieved the port in his Master's research, addressing in his thesis the question of which programming language would be the optimal choice for implementing teaching VMs, and to investigate the benefit of using object-oriented abstractions in low-level system code. Overall, the SOM++ source code is 14.4 % smaller than that of CSOM (5,804 PSLOC). Moreover, the implementation is more type-safe and slightly faster than CSOM.

SOM++ was used in 2009, in the same graduate course as CSOM before. Students implemented a wide range of extensions: a debugger, a profiler, memory managers based on an object table [7] and generational garbage collection [12], multithreading with M:N scheduling (M green threads mapped on N native ones), and a just-in-time compiler.

2.5 AweSOM

The youngest member of the SOM family is AweSOM, which started out as a coursework project in the 2009 VM course in which SOM++ was applied. AweSOM is a complete implementation of a SOM VM in Squeak Smalltalk.

The rationale behind AweSOM is twofold. On the one hand, it was interesting to have a SOM VM implemented in a language that is even more "extreme" than Java, which is after all statically typed. Moreover, Squeak, being a Smalltalk environment, features sophisticated development tools and supports a development style that is highly explorative and avoids edit-compile-run-debug cycles. Instead, one permanently deals with "living objects" and can observe and manipulate the running AweSOM VM. This is a highly interesting perspective for future teaching activities.

On the other hand, AweSOM is a first step in a larger research context, inspired by the PyPy[8] project [16], a metacircular Python implementation. Like PyPy, AweSOM shall either run in the high-level environment it is implemented in (Squeak), or lower-level (C++) code shall be generated and compiled, yielding a quick "production" VM. C++ code generation is a current project.

AweSOM is the smallest of all SOM family members, with only 2,432 LOC (including comments, ignoring blank lines). It is around 11 times slower than SOM++. AweSOM can however be completely explored as a live system in Squeak, which greatly improves accessibility.

2.6 Spin-off Projects

This section briefly presents results from applying SOM VMs in research projects. For details on the projects, the reader is referred to the respective resources.

[8] `codespeak.net/pypy`

CSOM/PL is a post-doctoral research project, providing a VM product line [6], i. e., it is possible to choose among various features to build a customised variant of CSOM. The feature set consists of most of the coursework results on CSOM from the years 2007 and 2008; in CSOM/PL, they are *combinable* instead of stand-alone extensions. To achieve this, the entire implementation of the base VM and all features was subject to a drastic remodularisation that was conducted in one student's Master's research. Based on the observation that VM implementations tend to be complex, leading to a high degree of entanglement between logical modules, an aspect-oriented approach was chosen to enhance the modularity characteristics of CSOM [9].

NXTalk[9] [3] is a Smalltalk programming environment for the Lego Mindstorms NXT[10] platform. NXTalk applications are implemented in Squeak and transferred to the dedicated Smalltalk run-time environment installed on the NXT. The NXTalk VM resulted from a Master's thesis and is a direct derivative of CSOM, which was tailored to run in the NXT environment. It is not only a research project investigating the feasibility of implementing dynamic object-oriented programming languages on embedded platforms, but also a viable teaching device for embedded systems.

SomActors++ is an experiment conducted as part of a doctoral research project concerned with providing VM abstractions for concurrency [13]. Inspired by work on an actors-based execution model [18], SOM++ was adapted to implement concurrency using actors [10]. The goal of this experiment was the evaluation of different implementation strategies for VMs on upcoming many-core architectures. SomActors++ was enabled by the results of using SOM++ for teaching, being based on a coursework group's implementation of a classic object table [7] for SOM++.

3. TEACHING VIRTUAL MACHINES

This section first gives an overview of the organisation and arrangement of the graduate VM course that was taught at HPI for three subsequent years (2007–2009). At HPI, all courses are evaluated by students; results for the VM course from all three years are also described below.

3.1 Course Design

The HPI VM course is a 4-SWS[11] course with 50 % lecturing and 50 % coursework sessions. *Lecturing* covers, in the given order, the following topics.

Introduction: Abstraction vs. virtualisation, history of virtualisation, machine architectures and VM types. This part of the lecture builds on chapter 1 of Smith and Nair's book on VMs [11]. The introduction is broad; the remainder of the course focuses on *high-level language* (HLL) VMs.

High-Level Language VMs: Process and HLL VMs, HLL VM building blocks by example: a tour of the source code of CSOM (2007/8) or SOM++ (2009).

Representing Application and Application Entities: This large block deals with the different forms in which applications can be fed to a VM for execution (*static representation*), and how applications and their elements look at run-time (*dynamic representation*). The "static" part covers classes (blueprints for objects, virtual method dispatch tables, formats: source code, class files, virtual images) and methods (bytecode instructions, instruction set architectures); the "dynamic", objects (slots, layout) and methods (in the form of activations / stack frames).

Execution: Interpreters and their optimisation (threading, selective inlining), just-in-time compilation (including a brief discussion of call graph-based optimisations), adaptive optimisation (profiling, sampling, on-stack replacement).

Memory Management: History of memory management, garbage collection in object-oriented HLL VMs (reference counting, mark/sweep, compaction, copying and generational collectors), factors influencing memory behaviour.

The three large main blocks of the course all use example VM implementations to illustrate realisations of the different concepts. After having introduced the concepts by means of a CSOM/SOM++ walkthrough, students are able to map them to more complex implementations easily. The used example VMs are the implementations of Smalltalk-80, Squeak, and Java in the form of Jikes (see Sec. 2.1).

Optionally, an introduction to the Smalltalk programming language is given if students have no experience with the language. This lecture is inserted right after the course introduction, and introduces the language in general, and the SOM Smalltalk dialect in particular.

In 2008/9, lecturing was mostly done in block mode early in the lecturing term, to convey the technical background early on, thus setting the stage for programming coursework.

3.2 Grading

Students are graded *solely* based on coursework assignments. They work in groups of two to three members, which are fixed. Coursework is split in two parts: a *programming* and a *reading* part. Both are equally weighted in grading.

Each group is assigned a single large-scale programming task (cf., e. g., Sec. 2.3) that has to be completed by the end of the course. On a regular basis, each group presents their intermediate results to all course participants and discusses them with the plenum. On the final (block-mode) day of the course, all groups give final presentations and submit a paper describing their implementation, along with their code. These three items are graded.

Two papers on VM research, ranging from historical papers to papers presenting very recent results, are assigned to each group. Each paper is also dealt with by two groups: one *proponent* group presents the paper in plenum and defends it in the ensuing discussion, and one *opponent* group attacks it. This requires both groups to thoroughly read the paper, and assigning two papers to each group ensures that groups act as both proponents and opponents. Presentation and discussion performance are both graded.

3.3 Course Evaluation

In Tab. 1 (see next page), relevant results from students' evaluation of the VM courses in which members of the SOM family were applied are displayed. All grades shown in the table may range from 1.0 (best) to 5.0 (worst), according to the German grading system. The upper part of the table gives information about course participants including the average grade they obtained in the course.

The lower part of the table presents results from course evaluation, focusing on the overall result and on those parts

[9] `www.hpi.uni-potsdam.de/swa/projects/nxtalk`

[10] `www.mindstorms.com`

[11] SWS is the German abbreviation for *contact hour per week per semester*; 1 SWS denotes a 45-minute block. Blocks occur in pairs, so 4 SWS imply two 90-minute slots per week during a 14-week lecturing period.

year →	2007	2008	2009
participants	15	8	17
participation in evaluation	69 %	50 %	63 %
participants' average grade	1.26	1.25	1.23
course evaluation	1.67	1.23	1.38
material	1.74	1.42	1.47
enough available	*1.78*	*1.5*	*1.3*
improves understanding	*1.67*	*1.5*	*1.4*
clearly arranged	*1.78*	*1.25*	*1.7*
coursework	1.96	1.0	1.33
improves understanding	*1.78*	*1.0*	*1.33*
encouraging atmosphere	*1.89*	*1.0*	*1.44*
aligned with lectures	*2.22*	*1.0*	*1.22*

Table 1: Evaluation of VM courses at HPI.

of the course where SOM VMs played an important role, namely the teaching materials and coursework categories. As a more detailed evaluation is not performed at HPI, the teaching materials category covers lecture slides as well as tools (to which group the VMs belong). Likewise, coursework covers both the programming and reading assignments.

Students generally obtained very good grades in all of the three courses, as the table shows. In fact, the worst grade in all three years was 1.7. Thus, it can be deduced that the SOM VMs are well appreciated, and that they allow students to participate in a course with great success. The VM courses are generally among the top-rated courses in graduate education at HPI. In 2008, the VM course was ranked the best non-seminar course. Notably, this was the second year where CSOM was applied. In 2009, using SOM++, the course is evaluated a little less favourably, which is especially apparent in the coursework category.

4. SUMMARY AND FUTURE WORK

The SOM family is a set of *accessible* VM implementations for use in teaching VM concepts to students at all levels and even constitutes a viable research platform. The facts that the hosted language, a Smalltalk dialect, is the same for all SOM family members, and that the different members' architectures are essentially the same, allows for seamless transitions between members, e. g., as skill levels improve.

Future work on the SOM family members consists in providing more standard components, e. g., a just-in-time compiler, and on continuously assessing and improving the overall architecture and shape of the code. Comprehensive written documentation is also an important part of our ongoing work, to eventually enable the employment of SOM VMs in other than our own institution.

Acknowledgements

SOM was created at the University of Århus by Jakob Roland Andersen, Kasper Verdich Lund, Lars Bak, Mads Torgersen, and Ulrik Pagh Schultz. The authors are grateful for their contribution. Sebastian Kanthak and Jan-Arne Sobania have made particular contributions to the implementations of SOM and CSOM, respectively, for which we thank them. We also thank all students that have participated in VM courses at HPI in 2007–2009. Their feedback has helped improve both the VMs' source code and the understanding of what properties are important in a VM implementation used in teaching.

5. REFERENCES

[1] B. Alpern et al. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[2] B. Alpern et al. The Jikes Virtual Machine Research Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44(2):399–418, 2005.

[3] M. Beck, M. Haupt, and R. Hirschfeld. NXTalk. Dynamic Object-Oriented Programming in a Constrained Environment. In *Proc. IWST*. ACM, 2010. To appear.

[4] J. R. Bell. Threaded code. *CACM*, 16(6), 1973.

[5] J. Bloch. *Effective Java: A Programming Language Guide*. Addison-Wesley Longman, 2nd revised edition, 2008.

[6] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[7] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[8] P. P. Gómez-Martín, M. A. Gómez-Martín, and P. A. González-Calero. Javy: Virtual Environment for Case-Based Teaching of Java Virtual Machine. In *Proc. KES*, 2003.

[9] M. Haupt, B. Adams, S. Timbermont, C. Gibbs, Y. Coady, and R. Hirschfeld. Disentangling Virtual Machine Architecture. *IET Software*, 3(3):201–218, 2009.

[10] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proc. IJCAI*, San Francisco, CA, USA, 1973.

[11] J. E. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan-Kaufmann, 2005.

[12] R. Jones and R. Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[13] S. Marr, M. Haupt, S. Timbermont, B. Adams, T. D'Hondt, P. Costanza, and W. De Meuter. Virtual machine support for many-core architectures: Decoupling abstract from concrete concurrency models. In *Proc. PLACES*, volume 17 of *EPTCS*, 2010.

[14] N. Nisan and S. Schocken. *The Elements of Computing Systems. Building a Modern Computer from First Principles*. MIT Press, 2005.

[15] R. E. Park. Software size measurement: A framework for counting source statements. Technical Report CMU/SEI-92-TR- 20, ESC-TR-92-20, Software Engineering Institute, Carnegie Mellon University, September 1992.

[16] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *Proc. OOPSLA'06*. ACM, 2006.

[17] T. Rowledge. A Tour of the Squeak Object Engine. In M. Guzdial and K. Rose, editors, *Squeak: Open Personal Computing and Multimedia*. Prentice Hall, 2001.

[18] H. Schippers, T. Van Cutsem, S. Marr, M. Haupt, and R. Hirschfeld. Towards an actor-based concurrent machine model. In *Proc. ICOOOLPS*. ACM, 2009.