Back to the Future

The Story of Squeak, A Practical Smalltalk Written in Itself

by

Dan Ingalls Ted Kaehler John Maloney Scott Wallace Alan Kay

at Apple Computer while doing this work,
now at Walt Disney Imagineering
1401 Flower Street, P.O. Box 25020, Glendale CA 91221
dani@wdi.disney.com

Abstract

Squeak is an open, highly-portable Smalltalk imple-mentation whose virtual machine is written entirely in Smalltalk, making it easy to debug, analyze, and change. To achieve practical performance, a translator produces an equivalent C program whose performance is comparable to commercial Smalltalks.

Other noteworthy aspects of Squeak include: a compact object format that typically requires only a single word of overhead per object; a simple yet efficient incremental garbage collector for 32-bit direct pointers; efficient bulk-mutation of objects; extensions of BitBlt to handle color of any depth and anti-aliased image rotation and scaling; and real-time sound and music synthesis written entirely in Smalltalk.

Overview

Squeak is a modern implementation of Smalltalk-80 that is available for free via the Internet, at

<http://www.research.apple.com/research/proj/learning_concepts/squeak/>

and other sites.

It includes platform-independent support for color, sound, and image processing. Originally developed on the Macintosh, members of its user community have since ported it to numerous platforms including Windows 95 and NT, Windows CE, all common flavors of UNIX, and the Acorn.

Squeak stands alone as a practical Smalltalk in which a researcher, professor, or motivated student can examine source code for every part of the system, including graphics primitives and the virtual machine itself, and make changes immediately and without needing to see or deal with any language other than Smalltalk. It also runs bit-identical images across its wide portability base.

Three strands weave through this paper: (1) the design of the Squeak virtual machine, which differs in several interesting ways from the implementation presented in the Smalltalk "Blue Book" [Gold83] and explored in the "Green Book" [Kras83]; (2) an implementation strategy based on writing the Squeak virtual machine in Smalltalk and translating it into C, using an existing Smalltalk for bootstrapping until Squeak was able to debug and generate its own virtual machine; and (3) the incremental develop-ment process through which Squeak was created and evolved over the course of a year.

Background

In December of 1995, the authors found themselves wanting a development environment in which to build educational software that could be used—and even programmed—by non-technical people, and by children. We wanted our software to be effective in mass-access media such as PDAs and the Internet, where download times and power considerations make compactness essential, and where hardware is diverse, and operating systems may change or be completely absent. Therefore our ideal system would be a small, portable kernel of simple and uniform design that could be adapted rapidly to new delivery vehicles. We considered using Java but, despite its promise, Java was not yet mature: its libraries were in a state of flux, few commercial implementations were available, and those that were available lacked the hooks required to create the kind of dynamic change that we envisioned.

While Smalltalk met the technical desiderata, none of the available implementations gave us the kind of control we wanted over graphics, sound, and the Smalltalk engine itself, nor the freedom to port and distribute the resulting work, including its host environment, freely over the Internet. Moreover, we felt that we were not alone, that many others in the research community shared our desire for an open, portable, malleable, and yet practical object-oriented programming environment. It became clear that the best way to get what we all wanted was to build a new Smalltalk with these goals and to share it with this wider community.

Project Plan

We did not have to start from scratch, as we had access to the existing Apple Smalltalk-80 implementation, which was a gold mine of useful software. This system consisted of an *image*, or object memory, containing the Smalltalk-80 class library, and a separate *interpreter*, or VM (virtual machine), for running on the Macintosh. However, the Apple image format was limited by its use of indirect pointers and an object table. Worse yet, the original interpreter consisted of 120 pages of sparsely commented 68020 assembly code that had passed through the hands of seven authors. Portable it was not.

We determined that implementation in C would be key to portability but none of us wanted to write in C. However, two of us had once adapted the Smalltalk formatter (pretty-printer) to convert a body of code to BCPL. Based on that experience, we determined to write and debug the virtual machine in Smalltalk. Then, in parallel, we would write (also in Smalltalk) a translator from Smalltalk to C, and thus let Smalltalk build its own production interpreter. Out of this decision grew the following plan for building a new Smalltalk system in the shortest possible time:

Produce a new image:
- Design a new Object Memory and image file format.
- Alter the ST-80 System Tracer to write an image in the new format.
- Eliminate uses of Mac Toolbox calls to restore Smalltalk-80 portability.
- Write a new file system with a simple, portable interface.

Produce a new interpreter written in Smalltalk:
- Type in the Blue Book descriptions for the Interpreter and BitBlt.
- Write a completely new Object Memory class.
- Debug the new Object Memory, Interpreter and BitBlt.

Compile the interpreter to make it practical:
- Design a translator from a subset of Smalltalk-80 to C.
- Implement this translator.
- Translate the virtual machine to C and compile it.
- Write a small C interface to the Mac OS.
- Run the compiled interpreter with the new image.

By following this plan, facilities became available just as they were needed. For example, the interpreter and object memory were debugged using a temporary memory allocator that had no way to reclaim garbage. However, since the system only executed a few byte codes, it never got far enough to run out of memory. Likewise, while the translator was being prepared, most of the bugs in the interpreter and object memory were found and fixed by running them in Smalltalk.

It was easy to stay motivated, because the virtual machine, running inside Apple Smalltalk, was actually simulating the byte codes of the transformed image just five weeks into the project. A week later, we could type “3 + 4” on the screen, compile it, and print the result, and the week after that the entire user interface was working, albeit in slow motion. We were writing the C translator in parallel on a commercial Smalltalk, and by the eighth week, the first translated interpreter displayed a window on the screen. Ten weeks into the project, we “crossed the bridge” and were able to use Squeak to evolve itself, no longer needing to port images forward from Apple Smalltalk. About six weeks later, Squeak’s performance had improved to the point that it could simulate its own interpreter and run the C translator, and Squeak became entirely self-supporting.

We attribute the speed with which this initial work was accomplished to the Squeak philosophy: do everything in Smalltalk so that each improvement makes everything smaller, faster, and better. It has been a pleasant revelation to work on such low-level system facilities as real-time garbage collection and FM music synthesis from within the comfort and convenience of the Smalltalk-80 language and environment.

Once we had a stable, usable interpreter, the focus shifted from creation to evolution. Performance improved steadily and support for color, image transforms, sound synthesis, and networking were added. These improvements were made incrementally, as the need arose, and in parallel with other projects that relied on the stability of the virtual machine. Yet despite the apparent risk of frequent changes to the VM, Squeak has proven as dependable as most commercial Smalltalks we have used. We attribute this success partly to our passion for design simplicity but mostly to the strategy of implementing the virtual machine in Smalltalk.

The remainder of the paper discusses various aspects of the Squeak implementation, its memory footprint and performance, the evolution of its user community, and plans for its future.

The Interpreter

We knew that the published Blue Book interpreter description would suffice to get us started. Moreover, we were spared from the tedium of transcription by Mario Wolczko, who had already keyed in the code for use as an on-line reference source for a Smalltalk implementation project at the University of Manchester.

The interpreter is structured as a single class that gets translated to C along with the Object Memory and BitBlt classes. In addition, a subclass (Interpreter Simulator) runs all the same code from within a Smalltalk environment by supporting basic mouse, file, and display operations. This subclass was the basis for debugging the Squeak system into existence. All of this code is included in the Squeak release and it can run its own image, albeit at a snail’s pace (every memory access, even in BitBlt, runs a Smalltalk method). Having an interpreter that runs within Smalltalk is invaluable for studying the virtual machine. Any operation can be stopped and inspected, or it can be instrumented to gather timing profiles, exact method counts, and other statistics.

Although we have constantly amended the interpreter to achieve increasing performance, we have stayed pretty close to the Blue Book message interface between the Interpreter and the Object Memory. It is a testament to the original design of that interface that completely changing the Object Memory implementation had almost no impact on the Interpreter.

The Object Memory

The design of an object memory that is general and yet compact is not simple. We all agreed immediately on a number of parameters, though. For efficiency and scalability to large projects, we wanted a 32-bit address space with direct pointers (i.e., a system in which an object reference is just the address of that object in memory). The design had to support all object formats of our existing Smalltalk. It must be amenable to incremental garbage collection and compaction. Finally, it must be able to support the “become” operation (exchange identity of two objects) to the degree required in normal Smalltalk system operation.

While anyone would agree that objects should be stored compactly, every object in Smalltalk requires the following “overhead” information:

- Size of the object in bytes: 24 bits or more,
- Class of the object: a full 32-bit object pointer,
- Hash code for indexing objects: at least 12 bits,
- Format of the object, specifying pointer or bits, indexable or not, etc.: 4 bits at least,
- ...and, of course, a few extra bits for storage management needs.

A simple approach would be to allocate three full 32-bit words as the header to every object. However, in a system of 40k objects, this cavalier expenditure of 500k bytes of memory could make the difference between an undeployable prototype and a practical application. Therefore, we designed a variable-length header format which seldom requires more than a single 32-bit word of header information per object. The format is given in Tables 1 and 2.

offset	contents	occurrence
-8	size in words (30 bits), header type (2 bits)	1%
-4	full class pointer (30 bits), header type (2 bits)	18%
0	base header, as follows...	100%
	storage management (3 bits)	
	object hash (12 bits)	
	compact class index (5 bits)	
	object format field (4 bits, see below)	
	size in words (6 bits)	
	header type (2 bits)	

Table 1: Format of a Squeak object header

0	no fields
1	fixed pointer fields
2	indexable pointer fields
3	both fixed and indexable pointer fields
4	unused
5	unused
6	indexable word fields (no pointers)
7	unused
8-11	indexable byte fields (no pointers): low 2 bits are low 2 bits of size in bytes
12-15	compiled methods: low 2 bits are low 2 bits of size in bytes. The number of literals is specified in method header, followed by the indexable bytes that store byte codes.

Table 2: Encoding of the object format field in a Squeak object header

Our design is based on the fact that most objects in a typical Smalltalk image are small instances of a relatively small number of classes. The 5-bit compact class index field, if non-zero, is an index into a table of up to 31 classes that are designated as having compact instances; the programmer can change which classes these are. The 6-bit size field, if non-zero, specifies the size of the object in words, accommodating sizes up to 256 bytes (i.e., 64 words, with the additional 2 bits needed to resolve the length of byte-indexable objects encoded in the format field). With only 12 classes designated as compact in the 1.18 Squeak release, around 81% of the objects have only this single word of overhead. Most of the rest need one additional word to store a full class pointer. Only a few remaining objects (1%) are large enough to require a third header word to encode their size, and this extra word of overhead is a tiny fraction of their size.

Storage Management

Apple Smalltalk had achieved good garbage collection behavior with a simple two-generation approach similar to [Unga84]. At startup, and after any full garbage collection (a mark and sweep of the entire image), all surviving objects were considered to be old, and all objects created subsequently (until the next full collection) to be new. All pointer stores were checked and a table maintained of "root" objects—old objects that might contain pointers to new objects. In this way, an incremental mark phase could be achieved by marking all new objects reachable from these roots and sweeping the new object area; unmarked new objects were garbage. Compaction was simple in that system, owing to its use of an object table. Full garbage collection was triggered either by an overflow of the roots table, or by failure of an incremental collection to reclaim a significant amount of space. That system was known to run acceptably with less than 500k of free space and to perform incremental reclamations in under 250 milliseconds on hardware of the 80's (16MHz 68020).

For Squeak, we determined to apply the same approach to our new system of 32-bit direct pointers. We were faced immediately with a number of challenges. First, we had to write an in-place mark phase capable of dealing with our variable-length headers, including those that did not have an actual class pointer in them. Then there was the need to produce a structure for remapping object pointers during compaction, since we did not have the convenient indirection of an object table. Finally there was the challenge of rectifying all the object pointers in memory within an acceptable time.

The remapping of object pointers was accomplished by building a number of relocation blocks down from the unused end of memory. A thousand such blocks are reserved outside the object heap, ensuring that at least one thousand objects can be moved even when there is very little free space. However, if the object heap ends with a free block, that space is also used for relocation blocks. If there is not enough room for the number of relocation blocks needed to do compaction in a single pass (almost never), then the compaction may be done in multiple passes. Each pass generates free space at the end of the object heap which can then be used to create additional relocation blocks for the next pass.

One more issue remained to be dealt with, and that was support of the **become** operation without an object table. (The Smalltalk **become** primitive atomically exchanges the identity of two objects; to Smalltalk code, each object appears to turn into, or "become," the other.) With an object table, the **become** primitive simply exchanges the contents of two object table entries. Without an object table, it requires a full scan of memory to replace every pointer to one object with a pointer to the other. Since full memory scans are relatively costly, we made two changes. First, we eliminated most uses of **become** in the Squeak image by changing certain collection classes to store their elements in separate Array objects instead of indexed fields. However, **become** operations are essential when adding an instance variable to a class with extant instances, as each instance must mutate into a larger object to accommodate the new variable. So, our second change was to restructure the primitive to one that exchanges the identity of many objects at once. This allows all the instances of a class to be mutated in a single pass through memory. The code for this operation uses the same technique and, in fact, the very same code, as that used to rectify pointers after compaction.

We originally sought to minimize compaction frequency, owing to the overhead associated with rectifying direct addresses. Our strategy was to do a fast mark and sweep, returning objects to a number of free lists, depending on size. Only when memory became overly fragmented would we do a consolidating compaction.

As we studied and optimized the Squeak garbage collector, however, we were able radically to simplify this approach. Since an incremental reclamation only compacts the new object space, it is only necessary to rectify the surviving new objects and any old objects that point to them. The latter are exactly those objects marked as root objects. Since there are typically just a few root objects and not many survivors (most objects die young), we discovered that compaction after an incremental reclamation could be done quickly. In fact, due to the overhead of managing free lists, it turned out to be more efficient to compact after every incremental reclamation and eliminate free lists altogether. This was especially gratifying since issues of fragmentation and coalescing had been a burden in design, analysis, and coding strategy.

Two policy refinements reduced the incremental garbage collection pauses to the point where Squeak became usable for real-time applications such as music and animation. First, a counter is incremented each time an object is allocated. When this counter reaches some threshold, an incremental collection is done even if there is plenty of free space left. This reduces the number of new objects that must be scanned in the sweep phase, and also limits the number of surviving objects. By doing a little work often, each incremental collection completes quickly, typically in 5-8 milliseconds. This is within the timing tolerance of even a fairly demanding musician or animator.

The second refinement is to tenure all surviving objects when the number of survivors exceeds a certain threshold, a simplified version of Ungar and Jackson's feedback-mediated tenuring policy [UnJa88]. Tenuring is done as follows. After the incremental garbage collection and compaction, the boundary between the old and new object spaces is moved up to encompass all surviving new objects, as if a full garbage collection had just been done. This "clears the decks" so that future incremental compactions have fewer objects to process. Although in theory this approach could hasten the onset of the next full garbage collection, such full collections are rare in practice. In any case, Squeak's relatively lean image makes full garbage collections less daunting than they might be in a larger system; a full collection typically takes only 250 milliseconds in Squeak.

We have been using this storage manager in support of real-time graphics and music for over a year now with extremely satisfactory results. In our experience, 10 milliseconds is an important threshold for latency in interactive systems, because most of the other critical functions such as mouse polling, sound buffer output and display refresh take place at a commensurate rate.

BitBlt

For BitBlt as well, we began with the Blue Book source code. However, the Blue Book code was written as a simulation in Smalltalk, not as virtual machine code to run on top of the Object Memory. We transformed the code into the latter form, made a few optimizations, and this sufficed to get the first Squeak running. The special cases we optimized are:

- the case when there is no source (store constant),
- the case when there is no halftone (store unmasked),
- the horizontal inner loop (no partial word stores).

Once Squeak became operational, we immediately wanted to give it command over color. We chose to support a wide range of color depths, namely: 1-, 2-, 4-, and 8-bit table-based color, as well as 16- and 32-bit direct RGB color (with 5 and 8 bits per color component respectively).

It was relatively simple to extend the internal logic of BitBlt to handle multiple pixel sizes as long as source and destination bit maps are of the same depth. To handle operations between images of different depth, we provided a default conversion, and added an optional color map parameter to BitBlt to provide more control when appropriate. The default behavior is simply to extend smaller source pixels to a larger destination size by padding with zeros, and to truncate larger source pixels to a smaller destination pixel size. This approach works very well among the table-based colors because the color set for each depth includes the next smaller depth's color set as a subset. In the case of RGB colors, BitBlt performs the zero-fill or truncation independently on each color component.

The real challenge, however, involves operations between RGB and table-based color depths. In such cases, or when wanting more control over the color conversion, the client can supply BitBlt with a color map. This map is sized so that there is one entry for each of the source colors, and each entry contains a pixel in the format expected by the destination. It is obvious how to work with this for source pixel sizes of 8 bits or less (map sizes of 256 or less). But it would seem to require a map of 65536 entries for 16 bits or 4294967296 entries for 32-bit color! However, for these cases, Squeak's BitBlt accepts color maps of 512, 4096, or 32768 entries, corresponding to 3, 4, and 5 bits per color component, and BitBlt truncates the source pixel's color components to the appropriate number of bits before looking up the pixel in the color map.

Smalltalk to C Translation

We have alluded to the Squeak philosophy of writing everything in Smalltalk. While the Blue Book contains a Smalltalk description of the virtual machine that was actually executed at least once to verify its accuracy, this description was meant to be used only as an explanatory model, not as the source code of a working implementation. In contrast, we needed source code that could be translated into C to produce a reliable and efficient virtual machine.

Our bootstrapping strategy also depended on being able to debug the Smalltalk code for the Squeak virtual machine by running it under an existing Smalltalk implementation, and this approach was highly successful. Being able to use the powerful tools of the Smalltalk environment saved us weeks of tedious debugging with a C debugger. However, useful as it is for debugging, the Squeak virtual machine running on top of Smalltalk is orders of magnitude too slow for useful work: running in Squeak itself, the Smalltalk version of the Squeak virtual machine is roughly 450 times slower than the C version. Even running in the fastest available commercial Smalltalk, the Squeak virtual machine running in Smalltalk would still be sluggish.

The key to both practical performance and portability is to translate the Smalltalk description of the virtual machine into C. To be able to do this translation without having to emulate all of Smalltalk in the C runtime system, the virtual machine was written in a subset of Smalltalk that maps directly onto C constructs. This subset excludes blocks (except to describe a few control structures), message sending, and even objects! Methods of the interpreter classes are mapped to C functions and instance variables are mapped to global variables. For byte code and primitive dispatches, the special message **dispatchOn:in:** is

mapped to a C switch statement. (When running in Smalltalk, this construct works by **perform:**-ing the message selector at the specified index in a case array; since a method invocation is much less efficient than a branch operation, this dispatch is one of the main reasons that the interpreter runs so much faster when translated to C).

The translator first translates Smalltalk into parse trees, then uses a simple table-lookup scheme to generate C code from these parse trees. There are only 42 transformation rules, as shown in Table 3. Four of these are for integer operations that more closely match those of the underlying hardware, such as unsigned shifts, and the last three are macros for operations so heavily used that they should always be inlined. All translated code accesses memory through six C macros that read and write individual bytes, 4-byte words, and 8-byte floats. In the early stages of development, every such reference was checked against the bounds of object memory.

```
& | and: or: not
+ - * // \\ min: max:
bitAnd: bitOr: bitXor: bitShift:
< <= = > >= ~= ==
isNil notNil
whileTrue: whileFalse: to:do: to:by:do:
ifTrue: ifFalse: ifTrue:ifFalse: ifFalse:ifTrue:
at: at:put:
<< >> bitInvert32 preIncrement
integerValueOf: integerObjectOf: isIntegerObject:
```

Table 3: Operations of primitive Smalltalk

Our first translator yielded a two orders of magnitude speedup relative to the Smalltalk simulation, producing a system that was immediately usable. However, one further refinement to the translator yielded a significant additional speedup: inlining. Inlining allows the source code of the virtual machine to be factored into many small, precisely defined methods—thus increasing code-sharing and simpli-fying debugging—without paying the penalty in extra procedure calls. Inlining is also used to move the byte code service routines into the interpreter byte code dispatch loop, which both reduces byte code dispatch overhead and allows the most critical VM state to be kept in fast, register-based local variables. All told, inlining increases VM performance by a factor of 3.4 while increasing the overall code size of the virtual machine by only 13%.

Sound

Several of us were involved in early experiments with computer music editing and synthesis [Saun77], and it was a disappointment to us that the original Smalltalk-80 release failed to incorporate this vital aspect of any lively computing environment. We determined to right this wrong in the Squeak release.

Early on, we implemented access to the Macintosh sound driver. As the performance of the Squeak system improved, we were delighted to find that we could actually synthesize and mix several voices of music in real time using simple wave table and FM algorithms written entirely in Smalltalk.

Nonetheless, these algorithms are compute-intensive, and we used this application as an opportunity to experiment with using C translation to improve the performance of isolated, time-critical methods. Sound synthesis is an ideal application for this, since nearly all the work is done by small loops with simple arithmetic and array manipulation. The sound generation methods were written so that they could be run directly in Smalltalk or, without changing a line of code, translated into C and linked into the virtual machine as an optional primitive. Since the sound generation code had already been running for weeks in Smalltalk, the translated primitives worked perfectly the first time they ran. Furthermore, we observed nearly a 40-fold increase in performance: from 3 voices sampled at 8 KHz, we jumped to over 20 voices sampled at 44 KHz.

WarpBlt

As we began doing more with general rotation and scaling of images, we found ourselves dissatisfied with the slow speed of non-integer scaling and image rotations by angles other than multiples of 90 degrees. To address this problem in a simple manner, we added a “warp drive” to BitBlt. WarpBlt takes as input a quadrilateral specifying the traversal of the source image corresponding to BitBlt’s normal rectangular destination. If the quadrilateral is larger than the destination rectangle, sampling occurs and the image is reduced. If the quadrilateral is smaller than the destination, then interpolation occurs and the image is expanded. If the quadrilateral is a rotated rectangle, then the image is correspondingly rotated. If the source quadrilateral is not rectangular, then the transformation will be correspondingly distorted.

Once we started playing with arbitrarily rotated and scaled images, we began to wish that the results of this crude warp were not so jagged. This led to support for over sampling and smoothing in the warp drive, which does a reasonable job of anti-aliasing in many cases. The approach is to average a number of pixels around a given source coordinate. Averaging colors is not a simple matter with the table-based colors of 8 bits or less. The approach we used is to map from the source color space to RGB colors, average the samples in RGB space, and map the final result back to the nearest indexed color via the normal depth-reducing color map.

As with the sound synthesis work, WarpBlt is completely described in Smalltalk, then translated into C to deliver performance appropriate to interactive graphics.

Code Size and Memory Footprint

Table 4 gives the approximate size of the main components of Squeak in lines of code, based on version 1.18 of December, 1996. Our measurement includes all comments, but excludes all blank lines. We present these statistics not as rigorous measurement, but more as an order-of-magnitude gauge. For instance, the entire virtual machine is approximately 100 pages. Of that, 6547 lines are in Smalltalk (translator not included) versus 1681 lines of OS interface in C that may need to be altered for porting.

Smalltalk	Lines	C	Lines
Interpreter	3951	OS interface	1681
Object Memory	1283		
BitBlt with Warp	1313		

Table 4: Lines of code in Squeak VM

The size of the 1.18 Squeak release image, with all development support, including browsers, inspectors, performance analyzers, color graphics, and music support is 968K bytes on the Macintosh. The code for the virtual machine, simulator, and Smalltalk-to-C translator, which are only needed by those

engaged in virtual machine development, adds 290K to this figure. The interpreter, when running, requires 300K on a Power PC Macintosh, and the entire Smalltalk environment runs satisfactorily with as little as 200K of free space available. In monochrome, the system runs comfortably in 1.8 MB. We distribute a 650K image with the complete development environment that runs in less than 1MB on the Cassiopeia hand held computer.

Performance and Optimization

Thanks to today's fast processors, Squeak's performance was satisfactory from the moment the translator produced its first C translation of the virtual machine. Since this debut, Squeak's performance has improved steadily, and the current version, 1.18, executes about four million byte codes or 173 thousand message sends per second on a 110 MHz Power PC Mac 8100.

Table 5 shows the improvement in Squeak's performance over its first year. Two simple benchmarks from the release were used to track the approximate byte code execution rate ("10 benchmark") and the cost of full method activation and return ("26 benchFib"). Note that the latter benchmark measures the worst case; not all message sends require a full activation.

Date	byte codes/sec	sends/sec
Apr. 14	458K	22,928
May 20	1,111K	60,287
May 23	1,522K	69,319
July 9	2,802K	134,717
Aug. 1	2,726K	130,945
Sept. 23	3,528K	141,155
Nov. 12	3,156K	133,164
Dec. 12	3,410K	169,617
Jan. 21	4,108K	173,360

Table 5: Squeak performance over time

The rapid early leaps in performance were due partly to removal of scaffolding—such as assertion checks and range checks on memory references—and partly to improving the runtime model of the translator. For example, object references were originally represented as offsets relative to the base of the object memory rather than as true direct pointers. After May, however, the easy changes had all been made and improvements came in smaller increments, sometimes only a few percent at a time. The most significant of these optimizations include:

- recycling method contexts (this cut the allocation rate by a factor of 10)
- managing the frequency of checks for user and timer interrupts
- keeping the instruction and stack pointers (IP and SP) in registers
- making the IP and SP be direct pointers, rather than offsets into their base object
- patching the dispatch loop to eliminate an unneeded compiler-generated range check
- eliminating store-checks when storing into the active and home contexts
- comparing small integers as oops rather than converting them into integers first
- peeking for and doing a jump-if-false byte code that follows a compare

Table 6 compares Squeak's current performance over a small suite of benchmarks with that of several commercial Smalltalk implementations that cover a cross-section of implementation technologies, including a bytecode interpreter similar to the original Smalltalk-80 virtual machine (Apple Smalltalk), an aggressively optimized interpreter (ST/V Mac 1.1), and two implementations using dynamic translation to native code (ParcPlace Smalltalk 2.3 and 2.5). In order to draw meaningful comparisons between Squeak and these 68K-based virtual machines, all timings except those in the last column were taken on a Duo 230 (33Mhz 68030). Since Squeak runs significantly better on modern processors with instruction caches and a generous supply of registers, the final column of the table, SqueakPPC, shows Squeak's performance relative to C on a Power PC-based Macintosh.

	AppleST	ST/V	PP2.3	PP2.5	Squeak	SqueakPPC
IntegerSum	185.00	32.00	7.58	6.92	62.34	72.56
VectorSum	99.00	30.00	10.30	11.50	61.70	41.01
PrimeSieve	53.00	40.00	16.07	12.10	70.53	51.57
BubbleSort	88.23	35.29	21.35	13.98	80.29	63.12
TreeSort	43.90	5.00	20.29	1.98	16.33	7.31
MatrixMult	40.79	6.00	22.80	2.94	18.00	36.74
Recurse	28.26	9.47	3.73	2.08	50.26	35.19

Table 6: Virtual machine performance relative to optimized, platform-native C for various benchmarks. Smaller numbers are better. A result of 1.0 would indicate that a benchmark ran exactly as fast as optimized C.

So far in the design of Squeak, we have emphasized simplicity, portability, and small memory footprint over high performance. Much better performance is possible. The PP2.3 and PP2.5 columns of Table 6 are examples of Deutsch-Schiffman-style dynamic translation (or "JIT") virtual machines [Deut84]. Dynamic translation avoids the overhead of byte code dispatch by translating methods into native instructions kept in a size-bounded cache. The Self project [ChUn91] [Hölz94] broke new ground in high performance by investing more compilation time in heavily used methods, using inlining to eliminate expensive calls and enable further optimizations. This work, which was later extended to Smalltalk and Java [Anim96], shows that one can obtain performance approaching half the speed of optimized C without compromising the semantics of a clean language. Unfortunately both of these approaches have resulted in virtual machine implementations that are, by Squeak standards, unapproachable and difficult to port.

We believe that Squeak can enjoy the same performance as commercial Smalltalk implementations without compromising malleability and portability. In our experience the byte code basis of the Smalltalk-80 standard [Inga78] is hard to beat for compactness and simplicity, and for the programming tools that have grown around it. Therefore dynamic translation is a natural avenue to high performance. The Squeak philosophy implies that both the dynamic translator and its target code sequences should be written and debugged in Smalltalk, then automatically translated into C to build the production virtual machine. By representing translated methods as ordinary Smalltalk objects, experiments with Self-style inlining and other optimizations could be done at the Smalltalk level. This approach is currently being explored as a way to improve Squeak's performance without adversely affecting its portability.

The Squeak Community

As exciting as the day the interpreter first ran, was the day we released Squeak to the Internet community. In the back of our

minds, we all felt that we were finally doing, in September of 1996, what we had failed to do in 1980. However, the code we released ran only on the Macintosh and, although we had worked hard to make it portable, we did not know if we had succeeded.

Three weeks later, we received a message announcing Ian Piumarta's first UNIX port of Squeak. He had ported it to seven additional UNIX platforms two weeks later. At the same time, Andreas Raab announced ports of Squeak for Windows 95 and Windows NT. Neither of these people had even contacted us before starting their porting efforts! A mere five weeks after it was released, Squeak was available on all the major computing platforms except Windows 3.1, and had an active and rapidly growing mailing list. Since that time, Squeak ports have been done for Linux, the Acorn RISC, and Windows CE, and several other ports are underway.

The Squeak release, including the source code for the virtual machine, C translator and everything else described in this paper, as well as all the ports mentioned above, is available through the following sites:

```
<http://www.research.apple.com/research
   /

          proj/learning_concepts/squeak/>
<ftp://ftp.create.ucsb.edu>
<ftp://alix.inria.fr>
<ftp://ftp.cs.uni-magdeburg.de/pub/
                Smalltalk/free/squeak>
```

The Squeak license agreement explicitly grants the right to use Squeak in commercial applications royalty-free. The only requirement in return is that any ports of Squeak or changes to the base class library must be made available for free on the Internet. New applications and facilities built on Squeak do not need to be shared. We believe that this licensing agreement encourages the continued development and sharing of Squeak by its user community.

Related Work

For the Smalltalk devotee, nothing is more natural than the desire to attack all programming problems with Smalltalk. Thus, there has long been a tradition of using Smalltalk to describe and debug a low-level system before its final implementation. As mentioned earlier, the Blue Book used Smalltalk as a high-level description of a Smalltalk virtual machine, and this description was actually checked for accuracy by running it. In LOOM [Kaeh86], the kernel of a virtual object memory was written and executed in a separate, simplified Smalltalk virtual machine whenever an "object fault" occurred. For better performance, this kernel was later translated into BCPL semi-automatically, then fixed up by hand. This experience planted the seed for the approach taken in Squeak two decades later.

A number of recent systems translate complete Smalltalk programs into lower-level languages to gain speed, portability, or application packaging advantages. Smalltalk/X [Gitt95] and SPiCE [YaDo95] generate C code from programs using the full range of Smalltalk semantics, including blocks. Babel [MWH94] translates Smalltalk applications into CLOS, and includes a facility for automatically winnowing out unused classes and methods.

Producer [Cox87] translated Smalltalk programs into Objective C, but required the programmer to supply type declarations and rules for mapping dynamically allocated objects such as Points into Objective C record structures. Producer only supported a subset of Smalltalk semantics because it depended on the Objective C runtime and thus did not support blocks as first-class objects. Squeak's Smalltalk-to-C translator restricts the programmer to an even more limited subset of Smalltalk, but that subset closely mirrors the underlying processor architecture, allowing the translated code to run nearly as efficiently as if it were written in C directly. The difference arises from a difference in goals: The goal of Squeak's translation is merely to support the construction of its own virtual machine, a much simpler task than translating full-blown Smalltalk programs into C.

Squeak's translator is more in the spirit of QUICKTALK [Ball86], a system that used Smalltalk to define new primitive methods for the virtual machine. Another Smalltalk-to-primitive compiler, Hurricane [Atki86], used a combination of user-supplied declarations and simple type inference to eliminate class checks and to inline integer arithmetic. Unlike Squeak's translator, Hurricane allowed the programmer to also use polymorphic arithmetic in the Smalltalk code to be translated. Neither QUICKTALK nor Hurricane attempted to produce an entire virtual machine via translation.

Type information can help a translator produce more efficient code by eliminating run-time type tests and enabling inlining. Typed Smalltalk [JGZ88] added optional type declarations to Smalltalk and used that type information to generate faster code. The quality of its code was comparable to that of QUICKTALK but, to the best of the authors' knowledge, the project's ultimate goal of producing a complete, stand-alone Smalltalk virtual machine was never realized. A different approach is to use type information gathered during program execution to guide on-the-fly optimization, as done in the Self [ChUn91] [Hölz94] and Animorphic [Anim96] virtual machines. Note that using types for optimization is independent of whether the programming language has type declarations. The Self and Animorphic virtual machines use type information to optimize declaration-free languages whereas Strongtalk [BrGr93], which augments Smalltalk with an optional type system to support the specification and verification of interfaces, ran on a virtual machine that knew nothing about those types. The subset of Smalltalk used for the Squeak virtual machine maps so directly to the fundamental data types of the hardware that the translator would not benefit from additional type information. However, we have contemplated building a separate primitive compiler that supports polymorphic arithmetic, in which case the declaration-driven optimization techniques of Hurricane and Typed Smalltalk would be beneficial.

Future Work

Work on Squeak continues. We are overhauling Squeak's graphics model to supplant the MVC model with a new one along the lines of Morphic [Malo95] and Fabrik [Inga88]. We also plan to complete Squeak's sound and music facilities by adding sound input and MIDI input and output.

We are collaborating with Ian Piumarta to produce a dynamic translation engine for Squeak, inspired by Eliot Miranda's BrouHaHa Smalltalk [Mira87] and his later work with portable threaded code. A top priority is to build the entire engine in Smalltalk to keep it entirely portable.

Just as we wanted Squeak to be endowed with music and sound capability, we also wanted it to be easily interconnected with the

rest of the computing world. To this end, we are adding network stream and datagram support to the system. While not yet complete, the current facilities already support TCP/IP clients and servers on Macintosh and Windows 95/NT, with UNIX support to follow soon.

Conclusions

As far as we know, Squeak is the first practical Smalltalk system written in itself that is complete and self-supporting. Squeak runs the Smalltalk code describing its own virtual machine fast enough for debugging purposes: although it requires some patience, one can actually interact with menus and windows in this mode. This is no mean feat, considering that every memory reference in the inner loop of BitBlt is running in Smalltalk.

To achieve useful levels of performance, the Smalltalk code of the virtual machine is translated into C, yielding a speedup of approximately 450. Part of this performance gain, a factor of 3.4, can be attributed to the inlining of function calls in the translation process. The translator can also be used to generate primitive methods for computationally intensive inner loops that manipulate fundamental data types of the machine such as bytes, integers, floats, and arrays of these types.

The Squeak virtual machine, since its source code is publicly available, serves as an updated reference implementation for Smalltalk-80. This is especially valuable now that the classic Blue and Green Books [Gold83] [Kras83] are out of print. A number of design choices made in the Blue Book that were appropriate for the slower speed and limited address space of the computer systems of the early 1980's have been revisited, especially those relating to object memory and storage reclamation. Squeak also updates the multimedia components of this reference system by adding color support and image transformation capabilities to BitBlt and by including sound output. While Squeak is not the first Smalltalk to use modern storage management or to support multimedia, it makes a valuable contribution by delivering these capabilities in a small one-language package that is freely available, and that runs identically on all platforms.

Final Reflections

While we considered using Java for our project, we still feel that Smalltalk offers a better environment for research and development. At a time when the world is moving toward native host widgets, we still feel that there is power and inspiration in having all of the code for every aspect of computation and display be immediately accessible, changeable, and identical across platforms. Finally, when most development environments fill 100 megabytes of disk space or more, Squeak is a portable, malleable, full-service computing environment, including browsing, split-second recompilation, and source debugging tools, all in a 1-megabyte footprint. Though many of its strengths are rooted in the past, Squeak is suited to the intimate computing potential of PDAs and the Internet, and our work is, now more than ever, inspired by the future.

Acknowledgments

The authors wish to acknowledge the support of Apple Computer throughout this project, especially Jim Spohrer, Don Norman, and Elizabeth Greer. We especially appreciate their wisdom in seeing that Squeak would be worth more if it were made freely available. We also wish to thank the entire Squeak community for their encouragement and support, especially those who have submitted code or donated their time and energy to maintaining Squeak ports and the Squeak mailing list and web sites.

References

[Anim96] Animorphic Systems, Exhibit at OOPSLA '96. Animorphic Systems was a small company that included several members of the Self team and produced extremely high performance virtual machines for Smalltalk and Java. The company has since been purchased by Sun Microsystems.

[Atki86] Atkinson, R., "Hurricane: An Optimizing Compiler for Smalltalk," *Proc. of the ACM OOPSLA '86 conf.*, September 1986, pp. 151-158.

[BrGr93] Bracha, G. and Griswold, D., "Strongtalk: Typechecking Smalltalk in a Production Environment," *Proc. of the ACM OOPSLA '93 conf.*, September 1993.

[Ball86] Ballard, M., Maier, D., and Wirffs-Brock, A., "QUICKTALK: A Smalltalk-80 Dialect for Defining Primitive Methods," *Proc. of the ACM OOPSLA '86 conf.*, September 1986, pp. 140-150.

[ChUn91] Chambers, C. and Ungar, D., "Making Pure Object-Oriented Languages Practical," *Proc. of the ACM OOPSLA '91 conf.*, November 1991, pp. 1-15.

[Cox87] Cox, B. and Schmucker, K., "Producer: A Tool for Translating Smalltalk-80 to Objective-C," *Proc. of the ACM OOPSLA '87 conf.*, October 1987, pp. 423-429.

[Deut84] Deutsch, L., and Schiffman, A., "Efficient Implementation of the Smalltalk-80 System," *Proc. 11th ACM Symposium on Principles of Programming Languages*, January 1984, pp. 297-302.

[Gitt95] Gittinger, Claus, Smalltalk/X, http://www.informatik.uni-stuttgart.de/stx/stx.html, 1995.

[Gold83] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA,1983.

[Hölz94] Hölzle, U., *Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming*, Ph.D. Thesis, Computer Science Department, Stanford University, 1994.

[Inga78] Ingalls, D., "The Smalltalk-76 Programming System, Design and Implementation" *Proc. 5th ACM Symposium on Principles of Programming Languages*, Tucson, January 1978.

[Inga88] Ingalls, D., Wallace, S., Chow, Y., Ludolph, F., and Doyle, K., "Fabrik: A Visual Programming Environment," *Proc. of the ACM OOPSLA '88 conf.*, September 1988, pp. 176-190.

[JGZ88] Johnson, R., Graver, J., and Zurawski, L.,"TS: An Optimizing Compiler for Smalltalk," *Proc. of the ACM OOPSLA '88 conf.*, September 1988, pp. 18-26.

[Kaeh86] Kaehler, Ted, "Virtual Memory on a Narrow Machine for an Object-Oriented Language," *Proc. of the ACM OOPSLA '86 conf.*, September 1986, pp. 87-106.

[Kras83] Krasner, G., ed., *Smalltalk-80, Bits of History, Words of Advice*, Addison-Wesley, Reading, MA,1983.

[Malo95] Maloney, J. and Smith, R., "Directness and Liveness in the Morphic User Interface Construction Environment," *UIST '95*, November 1995.

[Mira87] Miranda, E., "BrouHaHa—A Portable Smalltalk Interpreter," *Proc. of the ACM OOPSLA '87 conf.*, October 1987, pp. 354-365.

[MWH94] Moore, I., Wolczko, M., and Hopkins, T., "Babel—A Translator from Smalltalk into CLOS," *TOOLS USA 1994*, Prentice Hall, 1994.

[Saun77] Saunders, S., "Improved FM Audio Synthesis Methods for Real-time Digital Music Generation," in *Computer Music Journal 1:1*, February 1977. Reprinted in *Computer Music*, Roads, C. and Strawn, J., eds., MIT Press, Cambridge, MA, 1985.

[Unga84] Ungar, D.,"Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm," *Proc. ACM Symposium on Practical Software Development Environments*, April 1984, pp. 157-167. Also published as *ACM SIGPLAN Notices 19(5)*, May 1984 and *ACM Software Engineering Notes 9(3)*, May 1984.

[UnJa88] Ungar, D. and Jackson, F.,"Tenuring Policies for Generation-Based Storage Reclamation," *Proc. of the ACM OOPSLA '88 conf.*, September 1988, pp. 18-26.

[YaDo95] Yasumatsu, K. and Doi, N., "SPiCE: A System for Translating Smalltalk Programs Into a C Environment," *IEEE Transactions on Software Engineering 21(11)*, 1995, pp. 902-912.