Implementing a high-performance Smalltalk interpreter with genbc and genvm

Paolo Bonzini

Abstract

This paper will detail the implementation of the high-performance Smalltalk interpreter that is used in GNU Smalltalk. This virtual machine reaches good performance levels thanks to a very orthogonal bytecodes encoding, and a rich set of superoperators.

This paper explains the operation of the interpreter and of the tools that were developed to build it. genvm builds optimized code to execute elementary bytecodes and superoperators; genbc, instead, helps decoding the bytecodes in multiple places with a very limited amount of duplicated code.

A third tool, called superops, is more specialized that genbc and genvm. It is a static analyzer that finds a suitable set of superoperators, and produces the very fast peephole optimizers that actually combine bytecodes into superoperators.

Using these tools, we obtained a bytecode interpreter that is up to 40% faster than the previous optimized interpreter used by GNU Smalltalk; also, they allowed us to design a bytecode set that is on average 20% smaller than the previous one.

1 Introduction

This paper will detail the implementation of the GNU Smalltalk virtual machine, a high-performance, open-source implementation of the Smalltalk programming language.

GNU Smalltalk can use one of two execution engines. One is an indirect threaded code interpreter, which exploits the orthogonal encoding and the wealth of superoperators in the bytecode set in order to reach acceptable performance levels; the other is a just-in-time compiler, which trades portability for an approximately two-fold improvement in performance. Section 2 explains the operation of the interpreter.

Recently, we decided to write a translator from Java bytecodes to Smalltalk bytecodes. We were then confronted with the lack of free opcodes in the instruction set that we were using: this prevented us from fully optimizing operations such as modulo 2^{32} arithmetic, which would have been greatly sped up if they were assigned to special bytecodes.

We then proceeded to a complete redesign of the bytecode set, with the additional goals of making easy to compile for it, and to provide improved performance of interpreted Smalltalk code on modern microprocessors. The new bytecode set, analyzed in 3, demanded the development of an *ad hoc* tool

to really improve the performance of its predecessor. Section 4 concentrates on genvm, the tool that was developed to build the interpreter and improve its performance.

Despite its origin, genvm is not restricted to our bytecode set, nor is it restricted to stack-based virtual machine, or to virtual machines with lots of superoperators; it would also be possible to use it in a translator from bytecode to threaded code.

Prior to the adoption of genvm, the interpreter included a lot of duplicated code and C preprocessor tricks for the sake of efficiency. Another place where code duplication was notable was the decoding of bytecodes: there were separate bytecode decoders in the peephole optimizer, in the method verifier, in the disassembler, and so on. Upgrading all these separate decoders to a new bytecode set would have meant a lot of boring and error-prone work.

Since genvm worked very well, we decided to create another special tool to handle decoding, which is called genbc and is described in section 5. More versatile, and designed for flexibility rather than performance, genbc proved to be another useful tool for virtual machine developers.

The third tool, called superops, is more specialized and is meant to aid the design of the bytecode set, rather than its implementation. Its purpose is to determine a good set of superoperators from a static analysis of bytecodes; its usefulness lies in the different output formats that are available, ranging from superoperator recognizer to genbc source code. The functionality and implementation of superops is in section 6.

2 Architecture of the interpreter

The structure of the interpreter is based on the implementation presented by the Blue Book [5]. However, there are some differences between Smalltalk-80 and GNU Smalltalk.

First of all, Smalltalk-80 uses polling, with processes that run 100% of the time since the system is started, while GNU Smalltalk has a read-eval-print loop and is idle except during evaluations: this is a unique feature of GNU Smalltalk among the various implementations of the language.

Secondly, GNU Smalltalk is a modern implementation of Smalltalk and treats blocks as *closures*: in other words, a block to be a invoked multiple times recursively without any side effects. In older implementations, arguments to the blocks were kept in the enclosing method's activation record[1].

There are other important differences. For example, GNU Smalltalk offers interoperability between C and Smalltalk code, like most other scripting languages. It also has a richer object model: for example, it includes fixed objects, which are guaranteed not to be moved by the garbage collector, and weak references, which do not prevent the referenced object from being garbage collected. All these features however are beyond the scope of this paper.

[1] At the time the Blue Book was written, Smalltalk-80 had a hard limit of 32768 objects live in the system at the same time: this included classes, metaclasses, methods and execution contexts, for both the application and development environment: these peculiar binding rules were adopted to save on these objects.

2.1 Re-entrancy and thread-safety

The interpreter registers are stored in global variables such as `ip` and `sp`. For this reason, the interpreter is *not* thread-safe: processes are implemented entirely at the user level. User level threads have been measured to yield very good performance, also because only few operating systems (Windows, Linux, Solaris) support efficient thread-local storage. If such support were more widespread, it would be relatively easy[2] to employ kernel-mode threads, using synchronization primitives to implement recursive invocations of the interpreter.

However, the interpreter is reentrant, and every callback from C to Smalltalk starts a new instance of the interpreter. Every callback also starts a new Smalltalk process: for this reason, some care is necessary to simulate the LIFO behavior of the C call-stack even though the various callback processes effectively run in parallel. Each recursive invocation of the interpreter, however, will track a particular process and return as soon as the execution of that process terminates.

The interpreter expects every process to follow a particular setup: when it is started, a process should have exactly two activation records, one that will do the requested computation, and one pointing to a special *termination method*. This method holds a special `EXIT_INTERPRETER` bytecode: this makes the destruction of activation records faster, because the execution stack cannot ever become empty, nor the frame pointer become invalid.

2.2 Operation of the interpreter

The interpreter uses an indirect threaded code scheme. This is very similar to a big `switch` statement, but GNU Smalltalk uses GCC's computed-`goto` extension, together with a vector of addresses to the bytecode execution routines. This makes it possible to check for interrupts with a very small overhead *and* a low latency: when an asynchronous interrupt happens (for example when a timer is fired, or file I/O becomes possible), the signal handler for the interrupt swiftly changes the vector so that the interrupt will be handled at the next bytecode[3]. Bytecodes that perform jumps are performed by merely adjusting the value of `ip`.

When a method is invoked, the interpreter determines the class of the receiver, and checks to see if it already has cached the method definition for the given selector and receiver class. If so, that method is used, and if not, the receiver's method dictionary is searched for a method with the proper selector. If it cannot be found in that method dictionary, the method dictionary of the classes parent is examined, and on up the hierarchy, until a matching selector is found. If no selector is found, the receiver is sent a `doesNotUnderstand:` message.

If a method is found, it is examined for some special cases (including primitives, no-op methods, methods that return a constant, and accessor methods). If the method does not fit in the special cases, or if it is attached to a primitive that fails to complete, a "normal" method invocation is performed. The

[2] At least as far as the bytecode interpreter is concerned. Accessing thread-local variables from within just-in-time compiled code would be harder.

[3] There are two vectors: one holds the addresses of the 256 bytecode execution routines, while the other one holds 256 copies of the address of the interrupt handler.

interpreter is not called recursively: rather, the interpreter saves away its state in the currently executing activation record; then a new activation record is created and the global variables such as `ip`, `sp`, and `_gst_self` are initialized accordingly.

While the interpreter's registers are physical addresses (such as a pointer to the stack top, or to the next bytecode to be executed), the activation records hold relative locations, because one or more garbage collections could occur before the method is restarted, and the absolute pointers would be invalid.

Many fields in the activation records are also found in virtual machine registers such as `self`: these are left uninitialized when an activation record is created; indeed, they will never be filled for leaf activation records, but only when a method is invoked.

Once the interpreter's registers are set, the bytecode dispatching loop cheerfully begins fetching bytecodes from the new method, totally unaware that the method that it was executing has changed.

When a method returns, the context that called it is examined to restore the interpreter's global variables to the state that they were in before the callee was invoked. The values of `ip` and `sp` are restored to their absolute address values, and the other global state variables are restored accordingly. After the state has been restored, the interpreter continues execution, again totally oblivious to the fact that it's not running the same method it was on its previous byte code.

It is simpler to invoke blocks than methods, due to the absence of special cases, method lookup logic, and `doesNotUnderstand:`. The setup of the activation record, however, is the same in both cases. That's because, unlike the Blue Book and following other modern implementation of the language, GNU Smalltalk stores bytecodes for blocks into separate `CompiledBlock` objects, rather than embedding into the same `CompiledMethod` object that is used for the method enclosing the block.

3 Designing an efficient bytecode set

The bytecode encoding that GNU Smalltalk used prior to this work was substantially based on the Blue Book's bytecode encoding. This is a variable length encoding, where bytecode lengths range from 1 to 3 bytes. Almost all bytecodes are used, and the few that the Blue Book leaves free, were used either to implement further stack operations (such as *replace stack top* operations), or to extend the range of the operands.

We decided to redesign the bytecode set primarily because there were no more free opcodes, and we felt that the Blue Book specification was not clever enough in its use of the limited bytecode space. Also, previous research by Ertl [2] [3] suggested that a completely different design of the bytecode set would have improved the performance of interpreted Smalltalk code on modern superscalar machines.

We then set two precise design goals. We wanted to have complex bytecodes, in order to give more freedom to the compiler scheduler; and we also wanted to limit the overhead of interpretation, which in our case was dominated by the cost of fetching the bytecodes and decoding their arguments.

3.1 A new encoding for bytecodes

The first goal was achieved through the use of *superoperators*. These bytecodes represent a sequence of operations and, most of the times, the combination is less expensive than the sum of individual costs: this is because dispatching only happens one, and because the cost of adjusting the stack pointer can be done only once per bytecode.

The second goal was achieved with a very orthogonal bytecode encoding. All of the bytecodes are two bytes long and are formed by an opcode byte and an argument byte; if the bytecodes have no operand, the second byte is unused. Most of the times, bytecodes have one operand, and eight bits suffice to store it: then the argument byte hold the operand.

The argument is loaded at decode time into `arg`, a virtual machine register. When it is modified, `arg` behaves like a shift register. Most bytecodes reset it to zero after they are executed: so the net effect is usually that of loading the argument byte into `arg`[4].

There is an exception though: a special *extension bytecode* is provided, which is a no-op except that it does *not* reset `arg` to zero. So, the interpretation loop is as follows:

```
1 arg = (arg << 8) | ip[1];
2 ... execute the routine for the bytecode to *ip ...
3 if (*ip != EXT_BYTE)
4   arg = 0;
5 ip += 2
```

Then, it can be easily seen that the *extension bytecode* enables the shift register functionality of `arg`, widening the range of the arguments to the bytecodes. For example, the instruction `PUSH_INTEGER 1000` is encoded by these four bytes:

55 is the opcode for the extension bytecode

3 is the most significant byte of the argument 1000

44 is the opcode for the `PUSH_INTEGER` bytecode

232 is the least significant byte of the argument 1000

The *extension bytecode* is also used to encode two operand instructions: in this case, bits 0-7 of `arg` hold an argument[5], while the higher bits hold the other argument.

In the case of superoperators, all the suboperations except one must have fixed arguments: the one variable argument is found as usual in the `arg` register. The extension bytecode can be one of the operations that are composed, but you can apply extension bytes to a superoperator *only if the first bytecode of the superator is the one whose argument is variable.* This constraint avoids that the most significant bytes of `arg` refer to a different bytecode than the least significant byte.

[4]Since bytecodes are decoded at the same time the previous bytecode is executed, the compiler will propagate the assignment of zero, and eliminate the useless shift. As shown in the following sections, relying on this kind of optimization is very common in the implementation of the virtual machine.

[5]In the GNU Smalltalk bytecode set, there are only four two-operand instructions, and all of them have an argument with a limited range.

3.2 Rationale

This encoding is substantially variable length, just like the Blue Book and JVM bytecode sets. It does make two-operand bytecodes more expensive, but they are only four and relatively rare in this bytecode set. In addition, it has two interesting properties, so that it suits interpretation especially well.

First of all, it makes the decoding of the arguments pipelinable. Since the encoding of the next executed bytecode is known, it can be fetched and decoded while another bytecode is executing, and this allows the C compiler to do a better work.

Secondly, while small arguments *are* cheaper, this happens without wasting bytecodes and without slowing down the execution of bytecodes with small arguments. In the Blue Book, 32 bytecodes are reserved to push an object from the methods constant pool, and 32 more bytecode push a global variable: these two shortcut encodings alone take up a quarter of the bytecode space.

This happens to some extent in the Java Virtual Machine as well. The JVM is particularly complicated in this regard: some of the times, large arguments are encoded using a different bytecode; in other cases, instead, the same bytecode is used but is prefixed with a special `wide` bytecode: this can be implemented efficiently, but is very hard to pipeline the execution of the interpreter.

It is also worth noting that our encoding of two arguments bytecode, coupled with the usage of superoperators, makes these bytecodes a lot less expensive. This is because in our bytecode set, all two-argument bytecodes have one very predictable argument: so, most of the time, there is a one-operand superoperator that encodes a two-operand operation in a cheap and compact way.

4 Generating a high-performance interpreter with genvm

In a virtual machine interpreter, the code for each instruction is very similar to the code for the other bytecodes—this is especially true in our case, where we have literally hundreds of duplicated snippets in the implementation of the superoperators. While this can be taken care of with preprocessor macros, these make debugging harder and do not guarantee that the expanded code is suitable for being optimized by the C compiler.

For this reason, a specialized preprocessor was written with two purposes in mind: the first is to automatically generate boilerplate code (such as the dispatching tables, which can hold up to 256 copies of the same address); and the second is to optimize the execution of superoperators, avoiding unnecessary stack pointer movements.

4.1 genvm's input

genvm's input is divided into two parts: a part that declares the elementary operations that make up the various bytecodes and superoperators, and a part that combines these operations generating the actual code for the interpreter.

The operations are introduced by an `operation` directive and have several properties: a name, a variable number of arguments, and a *stack effect*. For example, the first operation described in figure 1 is called `PUSH_OUTER_TEMP`,

has two arguments named `n` and `scopes`, and pushes a single item on the stack that is referenced as `tos` in the source code. Stack effect is described as in Forth: input items are written before a `--`, and output items follow it.

Some operations, like `POP_STACK_TOP`, are completely described by the stack effect; most operations, however, include some C code that computes the value of the output items. As an additional hint to `genvm`, if some identically named items are placed in both the input and output parts of the stack effect, `genvm` knows that they are read but not changed. This is the case for the `tos` item in the second declaration of figure 1.

The other section (see figure 2) contains declarations for the vectors that will be used by the bytecode interpreter. Actual parameters are given: when they are constant, the compiler may remove or unroll loops. More than one `operation` can be referenced: not only this is the key to describing superoperators, but it also gives more opportunities to reduce code duplication while still optimizing the interpreter.

Every bytecode in GNU Smalltalk's virtual machine, for example, includes a `PREFETCH` operation which fetches the next bytecode and argument byte: for bytecodes that do not transfer control, it can be put at the beginning of the definition, so that the processor can prefetch the first few instructions of the next bytecode. Jumps, instead, invoke this operation after having adjusted `ip`.

Ertl et al. [3] describe how to use a similar technique to simplify the implementation of register-based machines: in this case, one can define simple fake stack-based operations and then implement register-based "superoperators" on top of them.

4.2 genvm's operation

As anticipated, `genvm` can compute the overall stack effect of a superoperator like bytecode 225, leaving the stack pointer unchanged until the end of the sequence.

`genvm` internally represents the stack effect of each operation as a tuple $(read, pop, push)$. For example the effect `(aa bb cc -- aa zz)` is represented by the tuple $(3, 2, 1)$: three elements are read and two of these (`bb` and `cc`) are popped off the stack; then one element, `zz`, is pushed on the stack. The overall difference in the stack pointer is $push - pop$, which is -1 in this case.

It is possible to define an operator on these tuples which yields the combined stack effect of two operations: `genvm` uses this operator to derive the state of the stack after each elementary component of a superoperator. This operator is defined as:

$$\begin{pmatrix} read_1 \\ pop_1 \\ push_1 \end{pmatrix} * \begin{pmatrix} read_2 \\ pop_2 \\ push_2 \end{pmatrix} = \begin{pmatrix} \max\{read_1, read_2 - (push_1 - pop_1)\} \\ \max\{pop_1, pop_2 - (push_1 - pop_1)\} \\ \max\{push_1, push_2 + (push_1 - pop_1)\} \end{pmatrix}$$

Note that the *read* and *pop* items are *negative* stack offsets (that is, $read_1 = 4$ means that you access items up `sp[-4]`), and the definition *subtracts* the stack effect of the first operator. Instead *push* is a *positive* stack offset, and the definition *sums* the stack effect of the first operator.

The code that `genvm` generates heavily relies on the presence of an optimizing compiler that can do good global constant propagation and dead code

```
operation PUSH_ESCAPED_TEMP n scopes ( -- tos ) {
  OOP contextOOP;
  gst_block_context context;

  for (; scopes-- >= 0; contextOOP = context->outerContext)
    context =
      (gst_block_context) OOP_TO_OBJ (_gst_this_context_oop);

  tos = context->contextStack[n];
}

operation DUP_STACK_TOP ( tos -- tos tos2 ) {
  tos2 = tos;
}

operation POP_STACK_TOP ( tos -- ) {
}
```

Figure 1: Some examples of `operation` declarations

```
table normal_byte_codes {
  ...

  33 = bytecode bc33 {
    PREFETCH ();
    PUSH_ESCAPED_TEMP (arg >> 8, arg & 255);
  }

  ...

  40 = bytecode bc40 {
    JUMP_BACK (arg);
  }

  ...

  225 = bytecode bc225 {
    PREFETCH ();
    POP_STACK_TOP ();
    DUP_STACK_TOP ();
    PUSH_SELF ();
  }

  ...
}
```

Figure 2: Part of a `table` declaration

elimination. For example, in the generated C source code for bytecode 225 (see figure 3), all the assignment to variables like `_stack0` can be propagated to the final instructions that actually modify the stack, and removed as dead code.

GCC handles this nicely: it produces assembly code equivalent to the right column of figure 3, and uses a register to hold `_extra1`. The compiler's output is shown in figures 4 and 5: executing three stack operations and fetching the next bytecode takes 11 instructions on the Intel x86, and 12 on the SPARC.

4.3 genvm and vmgen

On the surface, genvm is very similar to Anton Ertl's vmgen, described by Ertl et al. [3]. However there are several differences between the two:

- genvm does not attempt to cache the top of the stack in a register. This can be obtained with suitable definitions of the macros that access the stack, such as `PUSH_OOP` and `STACK_AT`.
- genvm does not attempt to manage the instruction pointer and to prefetch bytecodes. This is again left to macros like `NEXT_BC` and operations like `PREFETCH`, and gives more flexibility to the virtual machine implementor. For example, GNU Smalltalk uses a different implementation of this macro on architectures with a lot of registers, in order to implement pipelining as described by Hoogerbrugge and Augusteijn [6][6].
- genvm does not generate code to disassemble bytecodes and recognize superoperators. In GNU Smalltalk, other tools take care of this, and they are described in the following sections.
- genvm's syntax is easier to read, while vmgen uses almost no keywords and is a lot more terse.
- genvm can generate different vectors for bytecode dispatching. This is used in GNU Smalltalk to dispatch interrupts fast, as explained earlier in this section, but also to do limited inter-bytecode optimization: conditional jumps after bytecodes that push a boolean are optimized (depending on the boolean that is pushed) either to an unconditional jump or to a *no-op* bytecode.

5 Decoding bytecodes with genbc

genbc is a companion tool to genvm that takes care of other bytecode-related tasks. While genvm is limited exclusively to the implementation of a high-performance interpreter and to the optimization of stack references, genbc provides a "little language" that specifies the format of bytecodes: this description is completed with pluggable behavior that is included in the C source code for the virtual machine. The decoder that genbc generates is invoked with a special macro, `MATCH_BYTECODES` (see figure 6).

[6]Note that GNU Smalltalk's bytecode encoding allows a more efficient implementation of pipelining than the one described in the paper. In particular, since every opcode byte is followed by exactly one operand byte, it is possible to pre-decode only every other byte in the pre-fetch queue, because odd bytes are known to be arguments.

```
1    OOP _stack0, _stack1;
2    OOP _extra1;
3    _stack1 = STACK_AT (1);
4    _stack0 = STACK_AT (0);
5    PREFETCH;
6    _stack0 = _stack1;
7    _extra1 = _gst_self;
8    STACK_AT (0) = _stack0;
9    PUSH_OOP (_extra1);
10   NEXT_BC;
```

```
OOP _extra1;


PREFETCH;

_extra1 = _gst_self;
STACK_AT (0) = STACK_AT (1);
PUSH_OOP (_extra1);
NEXT_BC;
```

Figure 3: Code generated for bytecode 225, before and after copy propagation

```
mov    -4(%edi),%edx            # %edi holds sp
mov    _gst_self,%eax
mov    %edx,(%edi)              # STACK_AT (0) = STACK_AT (1)
add    $0x4,%edi                # PUSH_OOP (_gst_self)
mov    %eax,(%edi)              # ...
add    $0x2,%esi                # %esi holds ip
mov    dispatch_vec,%eax
movzbl (%esi),%edx
lea    (%eax,%edx,4),%eax
movzbl 0x1(%esi),%ebx           # %ebx holds arg
jmp    *eax
```

Figure 4: x86 assembly language code for the bytecode in figure 3

```
ld     -4(%l0),%o0              # %l0 holds sp
sethi  %hi(_dispatch_vec),%g1
st     %o0,(%l0)                # STACK_AT (0) = STACK_AT (1)
add    %l1,2,%l1                # %l1 holds ip
ld     [%g1+%lo(_dispatch_vec)],%o1
add    %l0,4,%l0                # PUSH_OOP (_gst_self)
ldub   [%l1],%o0                # Fetch the next bytecode
sll    %o0,2,%o0
ldub   [%l1+1],%l2              # %l2 holds arg
add    %o0,%o1,%g1
st     %l5,[%l0]                # %l5 caches _gst_self
jmp    %g1
```

Figure 5: SPARC assembly language code for the bytecode in figure 3
On the SPARC, every virtual machine register can be cached in a processor register.

The speed of genbc-generated code, while good, is not as important as in genvm; instead, flexibility is the primary goal of this tool. Most bytecode decoders in GNU Smalltalk take a help from genbc, including the decoder that is embedded in the just-in-time compiler.

5.1 genbc's input

genbc is built around two parsers written with Flex and Bison. The first builds a decision tree that decodes the bytecodes, while the second extracts the pluggable code from the C source files and combines them with the decision tree in order to produce a decoder.

genbc's input, shown in figure 7, is a sequence of pattern-action pairs similar to Awk scripts. Each pattern is simply a bytecode number in the range from 0 to 255, or the special patterns `BEGIN` and `END`; the action is mostly C code, with four specially translated statements. Each action also has access to two special variables, `IP` and `IP0`, which point inside the stream of bytecodes that is being decoded.

The four special instructions are:

- `extract`, which receives a set of *(variable, no. of bits)* pairs, and extracts the bytecode's operands in the given variables. The ability to specify the number of bits avoids to fill genbc's input with bit-masking operations: our new bytecode encoding does not employ bitfields, but this proved to be very useful in the old (Blue Book) bytecode set. `IP` is adjusted when a full byte is parsed.

- `dispatch`, which invokes an operation whose source code is in the body of the `MATCH_BYTECODES` macro. `dispatch` is very flexible: for example, it is possible to put it inside loops or conditional statements, and it does not terminate the execution of the action. This flexibility is central to the implementation of superoperators, which simply consist of multiple `dispatch` statements. The targets of `dispatch` statements should be declared one by one in the input file, together with the arguments that they accept.

- `continue`, which restarts the decoder on another bytecode without adjusting `IP0`. It is used, for example, when parsing the *extension bytecode*, used when the operand of a bytecode does not fit into a single byte.

- `break` gets out of the decoder. It is never used by GNU Smalltalk, but it may be useful if a superoperator included a jump bytecode, for example.

5.2 genbc's output

The C code that genbc outputs is, to say the least, of embarrassing quality. There are two reasons for this: limiting the code duplication in the output, and supporting the semantics of `dispatch` (which are, as said earlier, very important in the implementation of superoperators).

The decoder consists of a huge `switch` statement which has, in addition to a `case` label for each bytecode, another `case` label for each appearance of `dispatch`: before jumping to the body of the `MATCH_BYTECODES` macro, the

```
1   /* XLAT_BUILD_CODE_TREE is a unique identifier for this
2      occurrence of MATCH_BYTECODES; bp is the pointer to
3      the bytecodes to be decoded.  */
4   MATCH_BYTECODES (XLAT_BUILD_CODE_TREE, bp, (
5     PUSH_RECEIVER_VARIABLE {
6       push_tree_node (bp, NULL, TREE_PUSH | TREE_REC_VAR,
7                       (PTR) (uintptr_t) n);
8     }
9
10    ...
11
12    EXIT_INTERPRETER, INVALID {
13      abort ();
14    }
15  ));
```

Figure 6: Invoking the genbc-generated decoder

```
1  PUSH_LOCAL_VAR (n);
2  PUSH_ESCAPED_LOCAL (n, scopes);
3
4  ...
5
6  32 {
7    extract opcode (8), arg_lsb (8);
8    dispatch PUSH_LOCAL_VAR ((arg_msb << 8) | arg_lsb);
9  }
10
11 33 {
12   extract opcode (8), arg_lsb (8);
13   dispatch PUSH_ESCAPED_LOCAL (arg_msb, arg_lsb);
14 }
15
16 ...
17
18 /* EXT_BYTE */
19 55 {
20   extract opcode (8), arg_lsb (8);
21   arg_msb = (arg_msb << 8) | arg_lsb;
22   continue;
23 }
```

Figure 7: Part of the input to genbc

decoder sets a "return address" variable to the value of one such `case` label; at the end of the user code, the program does a `goto` to the `switch` statement, decoding another part of the superoperator. Anyway, the structure of the code is repetitive and explained in the genbc source code; it is not really worse than the outcome of most other program generators.

Each invocation of `MATCH_BYTECODES` generates a preprocessor macro: this macro includes the same C code that is specified in the macro, and some glue code to do the aforementioned `goto`s. All that `MATCH_BYTECODES` does is to concatenate the decoder with one of these macros.

6 Peephole optimization and superoperators

After generating bytecodes, the code runs through a peephole optimizer that also takes care of the generation of superoperators. This pass achieves a very good speedup, because it enables the interpreter to use genvm-optimized code for superoperators.

The peephole optimizer's operation is, once more, driven by machine-generated data: the generator program for these tables is called superops. But superops is not limited to generating the peephole optimizer: in fact, the input to genbc and genvm is itself generated in part by superops.

6.1 Peephole optimization

After generating bytecodes, the code runs through a peephole optimizer that also takes care of the generation of superoperators. The peephole optimizer simplifies a few sequences that are not worth being devoted a superoperator (for example, `STORE/POP/PUSH` is optimized to a single `STORE` if possible), and rewrites some other sequences to be more amenable to the construction of superoperators (for example, the first two bytecodes are swapped in `POP/LINE_NUMBER/PUSH` because `POP/PUSH` is a good candidate for a superoperator); then, it recognizes pair of adjacent bytecodes that can be condensed in a single superoperator.

Peephole optimization is done a basic block at a time, because jump destinations should be fixed up after the basic blocks are optimized. And since jumps are being considered, the optimizer also does limited *jump threading*, simplifying jump-to-jump constructions that appear in the bytecodes.

When rewriting the bytecodes in a basic block to use superoperators, the peephole optimizer does not use a grammar as in Evans and Fraser [4]; instead, we used a perfect hash function [1]. The perfect hash function generator embedded in the superops tool is based on GPERF [8]; the source code to GPERF is freely available, since the program is part of the GNU project.

The recognizer examines the last two bytecodes in the instruction stream and checks whether they can be replaced by a single superoperator; if there have already been successful substitutions, one or both of the considered bytecodes can possibly be a superoperator. The complexity of this algorithm is linear, because every iteration will either write a bytecode in the output basic block, or will combine two input bytecodes into one superoperator, thus reducing the length of the input basic block by one bytecode.

6.2 Determining the set of superoperators

There are basically three places that are influenced by the choice of a particular set of superoperators: the input to genbc, the `table` directives in genvm's input, and the peephole optimizers. To reduce the amount of repeated work, we wrote superops. This tool examines a Smalltalk image without superoperators, finds the best candidates, and then produces these three bodies of code.

The set of superoperators is established following Proebsting [7]: the program loops on all the bytecodes in every basic block and looking for the most frequent pairs of bytecodes (Proebsting actually starts from trees rather than bytecodes); superoperators that represent more than two operations can be created from those that have already been constructed.

A few superoperators simply encode a two-operand bytecode with an omitted argument. This may look like a waste of space, but *superops* only decided to build nine sequences of this kind, out of the 192 superoperators that it found.

The analysis was made on a large set of class libraries, totalling around 300,000 SLOC, including:

- a class library to map Smalltalk objects to a MySQL database (GLORP);
- two class libraries for graphical user interfaces, providing bindings to the Tk and Gtk+ toolkits;
- a lexical analyzer and parser for the Smalltalk language;
- the Smalltalk class browser;
- bindings to BSD sockets, and implementations of popular Internet protocols (FTP, HTTP, SMTP, etc.);
- a class library for internationalization and localization;
- the numerical methods code from Didier Besset's book *Numerical Methods in Smalltalk*.

Doing this analysis statically, instead of doing profiling at run-time, is much simpler (because it does not require any modification to the execution engine), reduces bytecode size significantly (by up to 45%) and does achieve significant improvements in execution speed. Also, building superoperators from the static frequencies of instruction sequences sidesteps the difficulty of establishing benchmarks that resemble the distribution of bytecodes in real applications.

superops is meant to be run just once; instead, genbc and genvm are run whenever the source code for the virtual machine is modified. Running superops, in practice, is part of the process of designing the bytecode set.

This is very different from what is done by vmgen. vmgen is designed for direct threaded code interpreters, which translate bytecodes to threaded code and synthesize superoperators on the fly. So, with vmgen you can change the set of superoperators without breaking binary compatibility. This is a weakness of our approach, but we believe it is balanced by the significant footprint reduction that we achieved.

7 Results

The new bytecode encoding that we designed and adopted is efficient and friendly to the underlying microprocessor, yet has a smaller footprint than the Blue Book's bytecode set. This represents a big improvement with respect to many of the virtual machines that are in use today, including the JVM.

The introduction of genbc eliminated a good deal of duplicated code from GNU Smalltalk source code, and it paid off even more when superoperators were added to the virtual machine: in same places, the code made assumptions that were not correct anymore, but all the complex decoding logic was embedded in genbc and in the code it generated. In fact, it took far longer to write superops than to fix these few bugs in the virtual machine!

The improvement in execution speed achieved together by genvm and superops ranges from 10% to 40%, mostly depending on the size of the L2 cache: this is because the interpreter may not fit in a small cache. In addition, a 45% reduction in the size of the bytecodes was measured: the basic encoding of the bytecodes was quite expensive (all bytecodes carry an operand byte even if it is unused), but the net result is that programs, on average, are *both* 15% smaller and 20% faster.

The best results were obtained on a Pentium 4 processor, probably because its pipeline is deeper and thus has more to gain from longer basic blocks. Similarly high speed-ups are found on the VLIW Itanium processor, where longer basic blocks give the compiler's instruction scheduler many more opportunities to fill the instruction bundles.

Our tools, especially genbc and genvm can be easily applied to different bytecode sets, even pre-existing designs like Java bytecode. genvm can generate boilerplate and repetitive code without relying directly on the C preprocessor, and provides an infrastructure explicitly geared towards performance. genbc's pluggability can be used in most diverse applications, ranging from static code analyzers to just-in-time compilers.

8 Acknowledgements

The author would like to thank prof. Stefano Crespi Reghizzi and Vincenzo Martena for their assistance during the development of this work at Politecnico di Milano–DEI. Many thanks to Stefano Crespi Reghizzi also for proofreading this paper.

References

[1] Richard J. Cichelli. Minimal Perfect Hashing Made Simple. *Communications of the ACM*, 23(1):17–19, December 1980.

[2] M. Anton Ertl. *Implementation of Stack-Based Languages on Register Machines*. PhD thesis, Technische Universität Wien, 1996. URL `http://www.complang.tuwien.ac.at/papers/ertl96diss.ps.gz`.

[3] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen — a generator of efficient virtual machine interpreters. *Software—Practice*

and Experience, 32(3):265–294, 2002. URL `http://citeseer.nj.nec.com/article/ertl02vmgen.html`.

[4] William S. Evans and Christopher W. Fraser. Bytecode Compression via Profiled Grammar Rewriting. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 148–155, 2001. URL `http://citeseer.nj.nec.com/evans01bytecode.html`.

[5] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 2nd edition, 1983.

[6] Jan Hoogerbrugge and Lex Augusteijn. Pipelined java virtual machine interpreters. In *Computational Complexity*, pages 35–49, 2000. URL `http://citeseer.nj.nec.com/hoogerbrugge00pipelined.html`.

[7] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 322–332, 1995. URL `http://citeseer.nj.nec.com/proebsting95optimizing.html`.

[8] Douglas C. Schmidt. GPERF: A Perfect Hash Function Generator. In *Second* USENIX *C++ Proceedings*, April 1990.