Automatic Generation of Compact Programs and Virtual Machines for Scheme

Mario Latendresse
Department of Computer Science
Rice University
latendre@cs.rice.edu

Abstract

Compact programs are not particularly needed on large workstations, but they become a necessity on small embedded systems. For example, smart cards have on the order of 1K of RAM, 16K of non-volatile memory, and 24K of ROM. This is an extreme situation, but many embedded systems also have memory constraints requiring compact code. Virtual machine can be an effective approach to obtain compact programs and bytecode is a common technique for encoding virtual instructions. If the instructions are tailored for a particular language, the resulting virtual programs are compact. We use a combination of techniques to automatically generate new instructions and new compact encodings for virtual instructions. The common bytecode encodings align instructions on byte boundaries. Our encoding does not align instructions, operational codes for instructions are Huffman encoded, and argument lengths are not necessarily a multiple of eight bits. New instructions are generated to replace repetitive sequences of instructions in programs. This process is done using a fixed basic set of instructions and a sample of programs. The virtual machines are automatically generated in C. The resulting compressed programs are interpreted **without** decompression. This approach is general enough to be applied to C, Java and many other languages. We demonstrate it on the Scheme language using several benchmarks. The resulting Scheme virtual machines and programs run efficiently and are compact enough to be ported on small embedded systems.

1 Introduction

Bytecode encodings of virtual instructions use a byte for the operational code (opcode) and multiples of eight bits for the arguments. We do not use this approach. More precisely, we use a combination of four techniques to automatically generate a virtual instruction set capable of compact encoding of programs. These techniques are:

1. Operational codes are Huffman encoded.
2. Repetitive sequences of instructions are replaced by one opcode.
3. Creation of instruction formats having constant arguments or lengths not necessarily a multiple of eight bits.
4. Non alignment of instructions on byte boundaries.

Technique 1 shortens frequently occurring opcodes. This should be done using statistics from program samples.

Technique 2 is similar to the Lempel-Ziv compression technique. A repetitive sequence gets an encoding of its own, that is, such a sequence becomes a new instruction receiving its own opcode. We call them macro-instructions. They may have several parameters.

Technique 3 saves memory space for the encoding of arguments. In many cases, it is advantageous to fix one or several arguments of an instruction. For example, the instruction `pushi 0` can be very repetitive[1]. In such a case, a new parameterless instruction `pushi_0` should be created. This introduces a new opcode. And in most cases, the arguments have various lengths. For example, a program may contain several `pushi` instructions with small arguments and a few with large ones. In that case, it could be preferable to have various formats to encode compactly the small arguments.

Technique 4 is natural since lengths of opcodes and arguments are not a multiple of eight bits. It brings some complications to the specification of branching addresses. In this work, we have adopted the following solution: The instruction following a jump to a subroutine is byte aligned. All other branching instructions specify a bit address.

Once an instruction set is generated, it is important to use an efficient decoder. This is very tedious to do by hand, due to the variable lengths of opcodes and arguments[2]. But very efficient and compact decoders can be generated automatically. We have implemented such tools for the host language C as explained in [15, 16]. The macro-instruction implementations are also generated in C. It is done by concatenating the C code of the basic instructions contained in the macro-instruction.

In this paper we focus on the compression techniques and demonstrate its usefulness on Scheme.

1.1 Motivation

Compact programs are not particularly needed on large workstations, but they become a necessity on small embedded systems.

For example, smart cards, a typical resource-constrained device, have on the order of 1K of RAM, 16K of non-volatile memory (EEPROM or flash), and 24K of ROM. This is an

[1] Suppose that `pushi` is an instruction that pushes an integer on top of a stack.

[2] It could be done easily if a large amount of space in the form of a lookup table is used.

Instruction	Standard Format
br d	(s 24)
bf d	(s 24)
pushi i	(s 24)
pushli i	(u 8)
push i	(u 16)
pushl i	(u 8)
pushg i	(u 16)
pop i	(u 8)
alloc i	(u 8)
storel i	(u 8)
storeli i	(u 8)
storeg i	(u 16)
ret i	(u 8)

Figure 1: Machina **basic instruction formats**

extreme situation, but many embedded systems also have memory constraints requiring compact code.

To tackle such small systems in Java, Sun has taken the approach of defining smaller virtual machines, like the KVM[3] [21], and apply restrictions on the language and the libraries. For the smart cards, this comes with major constraints, where floating-point computation, threads, and garbage collection have been removed. Tools are also provided to reduce memory usage, like Java CodeCompact^{tm}[21], which preloads class files, resolves dynamic links, and generates a complete executable ROM version. But this approach does not reduce the size of bytecoded programs.

The approach taken in our work is to generate a tailored instruction set given a sample of programs. This takes advantage of the fact that many embedded systems have some specific class of programs to run.

1.2 The general VM Machina

To apply the construction of compact programs to Scheme we use a general VM, called Machina. Our Scheme compiler generates instructions for Machina. The compiled programs are used as samples to build a new VM, which we call Schemina. The final executable programs are encoded using Schemina instructions and formats.

Machina is a simple stack machine with forty six basic instructions. It has a stack (S), a heap or memory (M), a pool of global variables (G), and a constant pool (C). The instructions are simple but this simplicity allows an automatic generation of more complex instructions tailored for the programs to execute. The complete instruction set is shown in Figure 7, and the standard formats are presented in Figure 1.

This machine is not tailored for Scheme as its instruction set does not include any particular operation for Scheme. This is intended to demonstrate that from a general VM it is possible to come up with a specific virtual Scheme machine.

1.3 Scheme compilation to Machina

We use the front-end of Gambit-C's Scheme compiler [9] to generate Machina code. The compilation is straightforward but some technical details are important to better understand the generation of macro-instructions.

The dynamic tagging of objects use the three lower bits of a 32 bits word. To tag a word, we use the instruction `pushi`, where its argument is the tag, followed by `or`. To untag, the sequence of instructions (`pushi 7`, `not`, `and`) is used.

In general, there are no run-time checks performed, although most of the required infrastructure is there. For example, a function call pushes the number of arguments on the stack, but the prologue of the function does not verify this value. The resulting run-time system does not include a garbage collector, but objects include a tagged word, and `call/cc` has not been implemented.

Some Scheme basic operations generate long sequences of Machina instructions. For instance, the creation of a rest argument needs a sequence of seventy Machina instructions. In a tailored Scheme implementation, this should take only a few bytes.

Primitives, like `car` and `cons` are implemented using sequences of Machina instructions. If it is assumed that these primitives are not rebound, they are in-lined by the compiler. If they occur frequently enough in the samples, their pattern of instructions are detected by the creation of macro-instructions. This is indeed the case in the experiment of Section 2.1.

Figure 8 presents some of the new instructions that are generated along with their format. The following sections show how this is done.

2 Generation of new instructions

The macro-instructions are the new instructions resulting from a statistical analysis of program samples. A macro-instruction is a sequence of basic instructions, possibly with parameters and control flow. In this section, we explain the general algorithm used, independently of Scheme, and in the next section it is applied to Scheme using our compiler with Machina as the target machine.

A compiler generates the sample programs using some instruction set. These resulting programs are divided in basic blocks. A basic block may contain control flow instructions as long as the resulting macro-instructions can be implemented in the host language. For Scheme, we accept basic block containing control flow instructions and a later stage eliminates all macro-instructions containing a jump instruction outside the macro sequence of instructions.

The creation of macro-instructions is done in three phases. In the first phase, the frequencies of sequences are recorded. These sequences may overlap. In the second phase, a greedy algorithm chooses one sequence at a time to create macro-instructions. This choice is based on space savings of non overlapping applications of the sequences; that is, it is in the second phase that the space saving of sequences is taken into consideration. In the final phase, macro-instructions and basic instructions may be attributed several formats, if it reduces space. A format describes the number of bits for each parameter, and may include one or several constants, fixing the corresponding parameters, thus further reducing space usage.

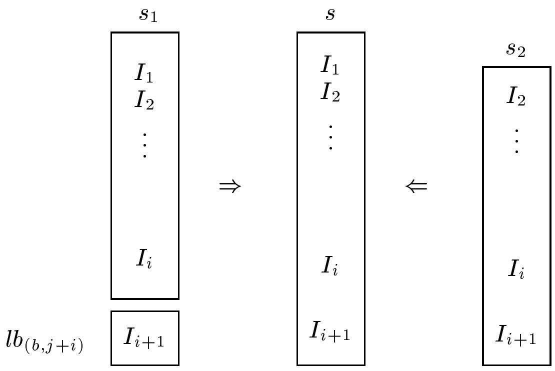
To reduce processing time in the first phase, an inferior frequency threshold is specified under which a sequence should not be considered. This reduces processing time since long sequences are constructed from shorter ones. Moreover, an upper limit on the length of macro-instructions is used[4].

[3]The KVM uses on the order of 128K, including libraries. The virtual machine itself is 40K-80K depending on the compiler and the target platform used.

[4]This eliminates the creation of very long macro-instructions due

For each $(s_1, P_{s_1}) \in S_i$

For each $(b, j) \in P_{s_1}$



$P_s = P_{s_1} \Cap P_{s_2}$, If $|P_s| \geq f_{\min}$ Add (s, P_s) to S_{i+1}

Figure 2: **Generation of sequences s of length $i+1$ from sequences of length $i > 2$**

The sequences of instructions are first considered as if all arguments of these instructions were arguments of the resulting macro-instruction. For example, in the sequence (`pushi 2`, `pushl 3`, `pushi 4`) the macro-instruction has three parameters. It is only after recording all frequencies of all sequences that macro-instructions with fixed parameters are considered and possibly generated. For the preceding sequence, a macro-instruction with one parameter (`pushi 2`, `pushl *`, `pushi 4`), where the star stands for a parameter, could be generated. This is an important case to handle, since many sequences have repetitive arguments.

An overview of the recursive technique used to generate sequences is described in Figure 2. The set of all sequences of length $i+1$ is generated using the set of sequences of length i. The generated sequence is s. The set P_s contains all places (b, j), for basic block b at instruction j, where s occurs. The operation $P_{s_1} \Cap P_{s_2}$ is a set intersection where the couples (b, j) for P_{s_2} is seen as $(b, j+1)$. The value $f_{\min}$ is the lower frequency threshold. In this way, it becomes faster to generate all such sequences and to compute their frequencies since it can be derived from the previous sets. A linear scan of the basic blocks generates the basic sequences of lengths two and three.

In the second phase, the greedy algorithm promotes the sequence with the largest space gain as a macro-instruction. The gain is the space saving of the bytecode minus the space needed to implement the macro-instruction in the VM.

The bytecode saving is the length of replaced opcodes minus the length of the new opcode multiplied by the frequency of the sequence. This requires an evaluation of the Huffman encoding of the new opcode taking into account the new frequencies.

The C code for the macro-instruction is a concatenation of the C code of the basic instructions. So, the space taken for the implementation is based on the space of the implementation of the basic instructions of the sequence and the space taken by a node in the decoder. It is an approximation since the size of the compiled concatenated code might be slightly different than the sum of its compiled part. The selection of sequences as macro-instructions stops when no gain can be obtained.

The final stage does a precise calculation on the space saving by choosing the parameter lengths. This choice may fix some parameters to some constant values.

The opcodes are constructed using canonical Huffman codes as this allows very compact decoders [18, 15].

to very long sequences of basic instructions for which the second phase would prove them useless. For example, if the sample contains a sequence of 200 instructions `pushi` and the lower threshold of repetition is 10, this would result in the creation of macro-instructions of length 2, 3, 4, ..., 191 instructions. But all become almost useless once one of them is chosen in the second phase.

libScheme	R^{4}RS library.
fib	Recursive evaluation of Fib(28).
conform	Type checking of a program.
earley	Generation of a parser.
tak	Evaluation of Tak(18, 12, 4).
qsort	QuickSort of 1000 integers.
mm	Squaring of a 100x100 matrix.
destruct	Operations on lists.

Figure 3: **Benchmarks used in experiments**

2.1 Application to Scheme

The benchmarks used to construct the VM are shown in Figure 3. The library libScheme is a modification of Dubé's library for BIT [6, 7]. We in-lined all primitive operations that existed in our compiler.

All these benchmarks were compiled using our compiler and combined into one list of 58194 Machina instructions. This list was divided in basic blocks and the generation of new instructions was done using the method of the preceding section.

We dub the resulting machine, and its encoded programs, by the name Schemina, an hybrid form of Machina for Scheme. It took about two minutes of cpu time to build the entire instruction set and generate the C code of the VM.

Figure 8, in the Appendix, contains twenty eight of the eighty macro-instructions created. The column 'Format' describes the length of parameters and the fixed constants. For example, macro 17 has format `(s 3 u 3 c 2)` which means that the first argument is a signed three bit integer, the second an unsigned three bit integer, and the third parameter has been removed since it is fixed at constant 2. It would be too long to explain the origin of all of those macro-instructions, but here are some explanations for some of them.

Macro-instruction 1 originated from the calling sequence of functions. The first three instructions remove the closure tag and push on the stack the code address and performs the jump to a subroutine. It has no parameter since constants 7 and 1 have fixed the original parameters.

Macro-instruction 2 allocates a closure with no free variables in the heap and initializes its first component. Macro-instruction 3 removes the three bit tag. Macro-instruction 4 is part of a sequence of instructions for a conditional expression. It generates value 26, for true, or value 10, for false, depending on a boolean value on the top of the stack. It is the conversion of a Machina boolean to a Schemina boolean!

Primitives are also present in these macro-instructions. The primitive `car` is implemented by the compiler as the sequence `pushi 7`, `not`, `and`, `pushi 0`, and `pusha`. By specifying 0 as its second argument, macro-instruction 6 contains this sequence, with the additional instruction `pushl` at the beginning and `pushi` at the end. So, this macro-instruction embodies a more practical instruction, that is a `car` of a local variable followed by the use of an integer. By specifying 1 as the second argument, this macro-instruction implements also `cdr`. Note that the second argument has only two bits.

The long macro-instruction 28 originates from the creation of a closure in the heap and storing its address in a global variable. This is a frequent sequence of instructions found from a series of global Scheme function definitions.

The last phase also created some new formats for the basic Machina instructions. For example, four new instructions for `pushi` were created having formats `(s 5)`, `(s 8)`, `(s 11)`, and `(s 14)`. Maximum formats of arguments of 32 bits were forcefully introduced in the instruction set to allow any standard Machina program to be loaded in this VM.

One important general observation is that it is not obvious how to generate by hand a set of virtual instructions to compactly encode programs. The obvious approach would be to implement the known primitives, the tagging, the untagging, the call to a function, etc., as virtual instructions. What these results show is that there is no such obvious division. As shown, it is better to implement a combination of primitives by one virtual instruction with a very short parameter, add more functionality to a task, for example loading a local variable followed by an integer, etc. And parameter lengths are more complicated to choose by hand. It is more accurate to let a program evaluate that appropriately.

3 Benchmark results

We present two types of results: speed of execution and size of programs.

3.1 Space usage

Figure 4 presents the size of the benchmarks encoded for Schemina as compression factors relative to the size of the bytecode Machina. The bytecode was generated by using an eight bit opcode and the standard instruction formats of Figure 1. Using our techniques, the compression factors range from 18% to 57%[5]. In comparison, gzip compression factors range from 17% to 77%. Of course, the techniques used by gzip are different, and no direct execution would be practically feasible, but this gives a point of comparison. The program gzip is better on large files. This is understandable given the fact that it uses a window technique. Our technique is close to gzip performance and sometimes better.

These results compare the sizes of the compressed codes with Machina bytecodes. But what are the performances compare to other Scheme systems?

A very relevant point of comparison is the BIT system developed by Dubé[6, 7]. It is a very compact implementation of Scheme on a 16 bits micro-controller having a compiler and a tailored VM. This implementation supports integer, char, string, vector, list, and procedure, but not the other basic types of Scheme. It has a real time garbage collector and implements `call/cc`. Its R^{4}RS library uses around 5K bytes. On the other hand, it sacrifices speed of execution for space.

[5]The compression factor is the value c/s where c is the size of the compressed program and s is the size of the uncompressed program.

	Bytecode Size	Schemina **Factors**	gzip **Factors**
libScheme	32040	23%	16%
fib	169	18%	77%
tak	582	26%	37%
earley	26271	31%	19%
conform	28599	23%	17%
mm	2550	30%	29%
destruct	3371	22%	22%
qsort	5827	57%	45%

Figure 4: **Compression factors**

Figure 5 presents the size of the benchmarks for BIT and our system. For Schemina and BIT two versions of each benchmark are presented due to the different compilation techniques used in both systems. In column 'With Library', the specified sizes include the necessary R^{4}RS library code for proper execution. In column 'Without Library', the sizes do not include the R^{4}RS library code. In this case, the code sizes give the direct memory space used by the code as if the whole R^{4}RS library were available in the VM. The bytecode sizes of MzScheme [10] were obtained using distribution 101, and without any part of the R^{4}RS library.

The fib program is very small for Schemina in both versions. It is much larger in BIT with the library. This is due to the compilation technique. The BIT compiler relies on the library to do fixnum arithmetic. But our compiler in-lined the code for them, not relying on any part of the library. For conform, the versions with library are very close in size, but it becomes higher for Schemina without the library code. All Schemina codes are smaller than MzScheme and in many cases quite smaller.

Using our tools the VM can be generated with various decoders ranging in size and speed of decoding. If we compile a VM having an 8 bit canonical decoder for the Pentium, the resulting executable is 29K. If we use a 6 bit canonical decoder, it falls to 27K. Note that part of the R^{4}RS library exists in the VM, since the macro-instructions cover some of those functions and are directly implemented in C in the VM.

3.2 Speed of execution

The Schemina benchmarks have similar speed compared to other interpreted Scheme systems. Figure 6 shows the execution times for BIT, Gambit, MzScheme, and Schemina, where garbage collection times have been subtracted for Gambit and MzScheme. We use Gambit interpreter version 3.0 and MzScheme bytecode compiler. We use an 8 bit canonical decoder for Schemina. For two benchmarks, BIT could not terminate with 64K of heap.

This comparison is not intended to be precise enough to draw some conclusions on the speed benefits of the techniques used by Schemina. There are too many differences between BIT, Schemina, Gambit, and MzScheme[6]. It is rather a comparison showing the practicality of the approach when considering speed of execution.

All Schemina programs are faster than BIT. This is due to two major reasons: the library coding technique, and the run-time mechanism used by BIT. Schemina can also

[6]In particular, Gambit interpreter has a single stepping facility built-in and points to source code when an error occurs.

	BIT With Library	BIT Without Library	Schemina With Library	Schemina Without Library	MzScheme
fib	1372	115	31	31	234
tak	1363	209	152	152	247
earley	6217	4613	8155	6947	7031
conform	6492	3722	6482	4007	10692
mm	1749	409	762	489	797
destruct	1894	555	755	497	958
qsort	4318	2943	4370	3355	10054

Figure 5: **Size of benchmarks for** BIT, Schemina, **and** MzScheme

	BIT	Gambit	MzScheme	Schemina
fib	15.05	6.27	1.67	1.29
tak	14.93	4.58	1.61	1.80
earley	–	1.86	0.69	1.48
conform	79.42	6.74	2.66	13.03
mm	56.97	10.22	4.26	4.55
destruct	4.45	1.69	0.68	0.77
qsort	–	3.30	1.02	4.47

Figure 6: **Execution times in seconds**

have slow execution time due to the library, and this shows quite well for **conform** where a large part of the library is used. Overall, the execution time of Schemina programs are comparable to tailored Scheme systems.

4 Related Work

Patterson and Henessy manually designed a compact native instruction set by studying sample programs generated from C code [19].

Wilner [23] was a early study of compressing program code using Huffman encoding. The decoding was done at the microprogramming level on a Burroughs B1700. This computer was an ideal candidate since it could handle streams of bits. The decoding technique used was not efficient since it was done bit by bit.

Baker [1] made a study of techniques to find similarities in bytecodes. Although it was mainly geared towards finding pairs of similar code segments, not counting the occurrences like we do in this work.

Ernst *et al.* [8] compress native code coming out of a C compiler. A tailored VM is generated for a C program. The intermediate representation is compressed using macro-instructions and fixing parameters. Instructions are aligned on byte boundaries. It is similar to Proebsting's [20] work. Their technique is competitive with gzip on native code. But it is not reported if the compression obtained is due to the use of the VM or the compression of the virtual program. Moreover, no timing of the execution of compressed programs is reported, although they show that the intermediate form can be compiled efficiently.

Several works compress native programs, using Huffman codes, doing decompression at the hardware level [13, 17, 2]. Decompression occurs between memory and cache and is mostly transparent to the processor. The advantage of this approach is the use of hardware to decompress, but this advantage comes with an increase hardware complexity. The compression factors are somewhere around 80% on native code.

Cooper and McIntosh [4] reduce program size by replacing repetitive sequences of instructions with a branching instruction. Suffix trees are used to identify repetitions in the native executable code. The code saving is on average 5%. This work differs from ours since it is done on native code where it is not possible to create new instructions. Debray *et al.* [5] propose a similar approach but it is done at the compiler level which brings more opportunity for compaction. They obtain an average 78% factor of compression.

Hoogerbrugge *et al.* [11] have very good results in producing compact code and in spirit it is one of the closest work to ours. It is similar to the ideas found in the Thumb and MIPS16 processors [22, 14] where only a part of the program is compressed. It gives a faster execution by compressing only the less used parts. A tailored VM is automatically generated given a C program. They obtain a 70% factor of compression when comparing the native codes.

Closer to Scheme, Scheme 48[12] is a compact implementation that was successfully used in a medium size M68000 based system [3] using 512K of RAM and 256K of EPROM. The VM executable uses about 24K of EPROM , with an initial heap image of 80K.

5 Conclusion

We have presented the use of four techniques to encode compactly virtual instructions: Huffman encoding of opcodes, replacement of repetitive sequences of basic instructions, instruction formats having argument lengths non multiple of eight bits, and non alignment of instructions on byte boundaries. These techniques are supported by tools to generate the necessary macro-instructions, decoders, and portable implementation allowing experimentation by the developer to generate a tailored instruction set given a sample of programs.

We have demonstrated their usefulness on Scheme, by showing that they can create particular instructions from general basic ones resulting in very compact programs and virtual machines.

In particular, it was shown that designing an instruction set by replacing repetitive sequences of elementary instructions and performing a precise evaluation of the parameter lengths can result in non trivial virtual instructions capable of good compression of programs for Scheme.

Our benchmarks demonstrate the compactness results and the speed of execution.

6 Acknowledgments

Thanks to Marc Feeley for helpful comments and revision of this paper. This work has been funded in part by Ericsson.

References

[1] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Proc. Second IEEE Working Conference on Reverse Engineering*, pages 86–95, July 1995. Received IEEE Outstanding Paper Award.

[2] Martin Benes, Andrew Wolfe, and Steven M. Nowick. A high-speed asynchronous decompression circuit for embedded processors. In *Proc. Conf. on Advanced Research in VLSI*, September 1997.

[3] Rodney A Brooks. *A robust programming scheme for a mobile robot.* Languages for Sensor-based control in Robotics, ed Ulrich Rembold Klaus Hormann, NATO ASI series, Springer-Verlag, 1987, 1987.

[4] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. Conf. on Programming Languages Design and Implementation*, 1999.

[5] Saumya Debray, William Evans, and Robert Muth. Compiler techniques for code compression. In *Workshop on Compiler Support for System Software*, 1999.

[6] Danny Dubé. Un système de programmation Scheme pour micro-contrôleur. Master's thesis, Université de Montréal, April 1996.

[7] Danny Dubé. BIT: A very compact Scheme system for embedded applications. In Matthias Felleisen, editor, *Proceedings of the Workshop on Scheme and Functional Programming*, Rice Technical Report 00-368, September 2000.

[8] Jens Ernst, Christopher W. Fraser, William Evans, Steven Lucco, and Todd A. Proebsting. Code compression. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 358–365, June 1997.

[9] Marc Feeley, James S. Miller, Guillermo J. Rozas, and Jason A. Wilson. Compiling higher-order languages into fully tail-recursive portable c. Technical Report 1078, Université de Montréal, DIRO, August 1997.

[10] Matthew Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice, 1997.

[11] Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, and Rik van de Wiel. A code compression system based on pipelined interpreters. *Software - Practice and Experience*, 29(11):1005–1023, September 1999.

[12] Richard Kelsey and Jonathan Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.

[13] T.M. Kemp, R.M. Montoye, J.D. Harper, J.D. Palmer, and D.J. Auerbach. A decompression core for PowerPC. *IBM Journal of Research and Development*, 42(6), November 1998.

[14] K. Kissell. *MIPS16: High-density MIPS for the Embedded Market.* Silicon Graphics MIPS Group, 1997.

[15] Mario Latendresse. *Génération de machines virtuelles pour l'exécution de programmes compressés.* PhD thesis, Université de Montréal, May 2000.

[16] Mario Latendresse and Marc Feeley. Fast and compact decoding of Huffman encoded virtual instructions. (In preparation).

[17] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. Improving code density using compression techniques. In *Proc. Int'l Symp. on Microarchitecture*, December 1997.

[18] Alistair Moffat and Andrew Turpin. On the implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, October 1997.

[19] D. Patterson and J. Hennessy. *Computer Architecture, a Quantitative Approach.* Morgan Kaufmann, 2nd edition, 1996.

[20] Todd A. Proebsting. Optimizing a ANSI C interpreter with superoperators. In *Proc. Symp. on Principles of Programming Languages*, pages 322–332, 1995.

[21] Sun. *Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices.* Sun Microsystems, May 2000.

[22] J. L. Turley. Thumb squeezes ARM code size. *Microprocessor Report*, 9(4), March 1995.

[23] W. T. Wilner. Burroughs B1700 memory utilization. *AFIPS FJCC*, 41:579–586, 1972.

A Machina basic instruction set and some generated Schemina instructions

`add`	$P[sp-1] \leftarrow P[sp] + P[sp-1];\ sp \leftarrow sp-1$
`sub`	$P[sp-1] \leftarrow P[sp-1] - P[sp];\ sp \leftarrow sp-1$
`mul`	$P[sp-1] \leftarrow P[sp] \times P[sp-1];\ sp \leftarrow sp-1$
`div`	$P[sp-1] \leftarrow P[sp-1]$ div $P[sp];\ sp \leftarrow sp-1$
`rem`	$P[sp-1] \leftarrow P[sp-1]$ rem $P[sp];\ sp \leftarrow sp-1$
`and`	$P[sp-1] \leftarrow P[sp] \wedge P[sp-1];\ sp \leftarrow sp-1$
`or`	$P[sp-1] \leftarrow P[sp] \vee P[sp-1];\ sp \leftarrow sp-1$
`asl`	$P[sp-1] \leftarrow P[sp-1] << P[sp]; sp \leftarrow sp-1$
`asr`	$P[sp-1] \leftarrow P[sp-1] >> P[sp]; sp \leftarrow sp-1$
`lsr`	$P[sp-1] \leftarrow (\text{unsigned})P[sp-1] >> P]; sp \leftarrow sp-1$
`eq`	$P[sp-1] \leftarrow$ If $P[sp] = P[sp-1]$ Then true Else false; $sp \leftarrow sp-1$
`neq`	$P[sp-1] \leftarrow$ If $P[sp] \neq P[sp-1]$ Then true Else false; $sp \leftarrow sp-1$
`gt`	$P[sp-1] \leftarrow$ If $P[sp] < P[sp-1]$ Then true Else false; $sp \leftarrow sp-1$
`lt`	$P[sp-1] \leftarrow$ If $P[sp] > P[sp-1]$ Then true Else false; $sp \leftarrow sp-1$
`not`	$P[sp] \leftarrow \neg P[sp];$
`exg`	$tmp \leftarrow P[sp];\ P[sp] \leftarrow P[sp-1];\ P[sp-1] \leftarrow tmp$
`dup`	$P[sp+1] \leftarrow P[sp];\ sp \leftarrow sp+1$
`pushli` i	$P[sp] \leftarrow P[sp-(P[sp]+i)]$
`pushi` i	$P[sp+1] \leftarrow i;\ sp \leftarrow sp+1$
`push` i	$P[sp+1] \leftarrow \&C_i;\ sp \leftarrow sp+1$
`pushl` i	$P[sp+1] \leftarrow P[sp-i];\ sp \leftarrow sp+1$
`pushg` i	$P[sp] \leftarrow G[i];\ sp \leftarrow sp+1$
`pusha`	$P[sp-1] \leftarrow M[P[sp] \times 4 + P[sp-1]];\ sp \leftarrow sp-1$
`pushac`	$P[sp-1] \leftarrow M[P[sp] + P[sp-1]];\ sp \leftarrow sp-1$
`pop` i	$sp \leftarrow sp - i;$
`popv`	$sp \leftarrow sp - (1 + P[sp]);$
`storea`	$M[4P[sp] + P[sp-1]] \leftarrow P[sp-2];\ sp \leftarrow sp-3$
`alloc` i	$hp \leftarrow (P[sp]+hp)^i;\ P[sp+1] \leftarrow hp; sp \leftarrow sp+1$
`storeac`	$M[P[sp] + P[sp-1]] \leftarrow P[sp-2];\ sp \leftarrow sp-3$
`storel` i	$P[sp-i] \leftarrow P[sp];\ sp \leftarrow sp-1$
`storeli` i	$P[sp-(P[sp]+i)] \leftarrow P[sp]; sp \leftarrow sp-2$
`storeg` i	$G[i] \leftarrow P[sp];\ sp \leftarrow sp-1$
`br` d	$pc \leftarrow pc + d$
`bf` d	$sp \leftarrow sp-1$; If $P[sp+1] =$ false Then $pc \leftarrow pc + d$
`jsr`	$tmp \leftarrow P[sp];\ P[sp] \leftarrow pc;\ pc \leftarrow tmp$
`ret` i	$pc \leftarrow P[sp];\ sp \leftarrow sp - i - 1$
`writec`	Output character $P[sp]$; $sp \leftarrow sp-1$
`stop`	Stop program execution

Figure 7: Machina **instruction set**

	Sequence	Format
1	(pushi *) (not) (and) (dup) (pushi *) (pusha) (jsr)	(c 7 c 1)
2	(pushi *) (alloc *) (dup) (pushi *) (exg) (pushi *) (storea) (dup)	(c 8 c 8 c 3 c 0)
3	(pushi *) (not) (and)	(c 7)
4	(bf 2) (pushi *) (br 1) (pushi *)	(c 26 c 10)
5	(pushi *) (pushl *)	(c 18 c 2)
6	(pushl *) (pushi *) (not) (and) (pushi *) (pusha) (pushi *)	(u 3 c 7 s 2 s 5)
7	(dup) (pushi *) (exg) (pushi *) (storea) (dup)	(c 3 c 0)
8	(pushi *) (alloc *) (dup)	(c 8 c 8)
9	(pushl *) (pushi *) (not) (and) (pushi *) (pusha) (pushi *)	(u 3 s 4 s 1 s 2)
10	(pushi *) (exg) (pushi *) (storea)	(c 34 c 1)
11	(pushi *) (storea)	(c 0)
12	(pushl *) (pushi *)	(c 2 c 1)
13	(exg) (pushi *) (storea)	(c 2)
14	(pushl *) (pushl *)	(c 0 u 2)
15	(pushi *) (not) (and) (pushi *) (pusha) (pushi *)	(c 7 c 0 c 7)
16	(push *) (exg) (pushi *) (storea)	(u 8 c 1)
17	(pushi *) (pushl *) (pushi *)	(s 3 u 3 c 2)
18	(pushi *) (pushl *) (pushi *)	(s 7 u 2 s 2)
19	(pushi *) (pushl *)	(c 18 c 1)
20	(pushi *) (pushl *)	(c 0 u 3)
21	(pushi *) (pushg *)	(c 18 u 7)
22	(pushi *) (or)	(c 3)
23	(dup) (pushi *)	(c 0)
24	(push *) (exg) (pushi *) (storea)	(u 6 c 1)
25	(pushi *) (pushg *)	(s 3 u 3)
26	(pushi *) (alloc *)	(c 12 c 8)
27	(pushi *) (pushi *)	(c 10) (c 10)
28	(storea) (dup) (push *) (exg) (pushi *) (storea) (pushi *) (or) (storeg *) (pushi *)	(u 6 c 1 c 3 u 8 s 5)

Figure 8: **Some** Schemina **instructions, that are macros generated for** Scheme **using** Machina