An Instruction for Direct Interpretation of LZ77-compressed Programs

Christopher W. Fraser
cwfraser@microsoft.com

September 2002

Technical Report
MSR-TR-2002-90

Abstract

A new instruction adapts LZ77 compression for use inside running programs. The instruction economically references and reuses code fragments that are too small to package as conventional subroutines. The compressed code is interpreted directly, with neither prior nor on-the-fly decompression. Hardware implementations seem plausible and could benefit both memory-constrained and more conventional systems.

The method is extremely simple. It has been added to a pre-existing, bytecoded instruction set, and it added only ten lines of C to the bytecode interpreter. It typically cuts code size by 30%; that is, the compression ratio is 0.70x. More ambitious compressors are available, but they are more complex, which retards adoption. The current method offers a useful trade-off to these more complex systems.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
USA

http://research.microsoft.com

Introduction

Code compression can be used to save network, disk, memory, and even cache space. The various scenarios require different techniques. Modem bottlenecks, for example, can justify compressors and decompressors too costly for use with disk compression software.

At the other end of the spectrum is directly interpretable code, either machine code or interpretive code. For these systems, the scarcity is at the lowest levels of the memory hierarchy, so there is no room to decompress the code at all. The choices are either decompression into an internal hardware instruction cache, queue, or buffer [Abali et al; Benveniste et al; Ditzel et al; Game and Booker; Lefurgy et al, 1999; Lefurgy et al, 2000; Larin and Conte] or interpretation without decompression [Debray et al; Evans and Fraser; Proebsting].

These constraints are less common on current personal computers or servers than on embedded systems such as cellular phones, where market pressures require maximizing features while minimizing ROM. Features that don't need the fastest responses—such as those associated with key presses—are candidates for optimizations that trade time for space, including interpretation.

This paper augments the toolbox for this scenario. Its method compresses a bytecode by about 30% (that is, a compression ratio of 0.70x), but it adds only ten lines of C code to the bytecode interpreter. It starts with a well-studied method from general-purpose data compression and adapts the method to the problem of compressing a program that must be interpreted directly, without decompression.

Background: LZ77 compression

LZ77 compression [Ziv and Lempel] accepts a stream of characters—typically bytes—and produces a stream that interleaves *literals* and *pointers*. Each pointer indicates a *phrase* in the previous N characters and has two parts: a displacement and a length. The displacement gives the distance back to the phrase, and the length identifies the number of characters in the phrase. For example, the byte string

```
Blah blah.
```

compresses to

```
Blah b5,3.
```

where the underlined material denotes a pointer. The displacement is five, and the length is three, because the next three bytes repeat those back five bytes.

The actual encoding must distinguish pointers from literals, and there are a variety of ways to do so [Bell, Cleary, and Witten]. One method inserts bit masks. For example, an encoding of the compressed sequence above might use these bytes:

```
11111101
'B'
'l'
'a'
'h'
' '
'b'
0x30
0x05
'.'
```

The initial byte above is a bit mask and displayed as such above; it reports the fact that there are six leading literal bytes ("`Blah b`"), followed by one pointer, followed by one last literal byte (the period). The literal bytes and pointer components follow the bit mask immediately, followed by the next bit mask, and so on. In the example above, a twelve-bit displacement (`005`) follows a four-bit length (`3`).

More elaborate encodings offer multiple length or pointer sizes and thus widen the fields in the bit mask above to two or three bits [Fenwick]. These fields are thus roughly analogous to an instruction opcode, and the length, displacement, and literal data—though separated somewhat from the mask—are roughly analogous to instruction operand fields. LZ decompressors interpret these "instructions" in a single pass over the input; the method presented below interprets them as part of an interpreter for conventional instructions.

The Echo instruction

The current work adds LZ77 pointers to conventional instruction sets. The assembler instruction

```
echo .-5,3
```

commands the (hardware or software) interpreter to fetch and execute three instructions starting five bytes back from the `echo` instruction. Some readers might find it helpful to think of echo instructions as like C's `#include` directives but with the substitution happening dynamically in an instruction cache or buffer. The assembler encoding suggested above is, of course, a readable version of the binary encoding seen by the interpreter.

In pseudo-code, the semantics for an echo instruction are simple:

1. Save the PC.
2. Subtract the contents of the echo instruction's displacement field from the PC.
3. Set N to the contents of the echo instruction's length field.
4. Fetch and execute the instruction at the address in the PC.
5. Decrement N and go back to Step 4 if the result exceeds zero.
6. Restore the PC and bump it past the echo instruction.

Some echo instructions will point back to phrases that include other echo instructions. This feature obliges the interpreter to maintain a stack of program counters and lengths. As the interpreter bumps the former, it subtracts one from the latter. When a length drops to zero, the interpreter pops the stack and thus resumes interpretation of the containing echo instruction. The stack is no problem for software interpreters, but hardware implementations might benefit from a small upper bound on the nesting level, which could be enforced easily by compilers or post-processors that create the echo instructions.

Compression improves if echo instructions can also reference *fragments* of the streams represented by earlier echo instructions. Consider this example:

```
echo .-100,4
echo .-200,4
```

and assume that these two echo instructions reference only primitive or base instructions, not other echo instructions. Suppose now that the program later repeats the last six of the eight base instructions. This six-instruction phrase might not appear anywhere in the compressed program, because the two four-instruction phrases above aren't adjacent. Such a program would benefit from an extended echo instruction, which adds a field that tells how many leading primitive (that is, non-echo) instructions to skip. For example,

```
echo .-10,6,2
```

interprets six instructions back ten bytes, but the interpreter skips the first two instructions yielded up by the instruction phrase there, which is guaranteed to start with another echo instruction. Those two instructions don't count toward the six. When it is necessary to distinguish between echo instructions with and without offset fields, the qualifiers "extended" and "basic" are used below.

The semantics above for echo instructions work for straight-line code but fail for phrases with internal control flow:

- First, the static instruction counts in the echo instructions can get out of sync with the number of instructions executed dynamically. Suppose, for example, that a program has exactly one repeated phrase, which includes a conditional branch over one or more instructions also within the phrase. Then an execution of the sole echo instruction could take the branch and cause the loop above to execute unwanted instructions past the end of the first instance of the phrase.
- Second, Step 6 above can quash valid jumps. Suppose, for example, that the PC is changed by an unconditional jump executed by Step 4 above. Suppose also that N happens to be one at this point, so the loop is about to exit. Then Step 6 will clobber the new value in the PC before it can be used, nullifying the jump before it can take effect.

Implementations may solve this problem by forbidding echo instructions to point back to phrases that embed control transfers. That is, when recognizing a repeated phrase, the compressor must truncate the extension of the phrase when the next operator is found to be a branch or jump. A more complicated semantics might avoid this problem by, for example, exiting all nested echo instructions at jumps; such semantics are a natural subject for future research.

The implementation below invokes a fresh copy of the interpreter (with its own PC) at each call, so it is not obliged to treat calls and returns like branches and jumps and thus allows echo instructions to reference phrases with calls and returns.

Labels present another complication. If the phrase referenced by an echo instruction spans a label, then jumps to such labels would need to skip the phrase elements before that label. In order to meet this requirement, such jumps would need to be augmented with a field encoding this offset, and such fields would naturally hurt compression. Worse, indirect jumps would need either a distinct offset field for each potential target or a restriction that all targets share a common offset. To avoid these complications, echo instructions have to date been constrained to reference no labels. That is, the LZ compressor stops extending a phrase when it encounters a label.

The echo instruction complements typical call or subroutine-jump instructions. Both allow reuse of common code fragments, but they make different trade-offs and thus benefit different situations. Table 1 contrasts the trade-offs. A key difference for compression is that conventional subroutines—even light-weight ones [Cooper and McIntosh; Fraser, Myers, and Wendt; Kunchithapadam and Larus]—need a return instruction, which makes some small fragments uneconomical and precludes "falling into" the first copy of each fragment.

Table 1. Call version Echo

Call instructions …	*Echo instructions …*
… use subroutine prologues and epilogues, which add at least one instruction, namely a return.	… omit prologues and epilogues, allowing reuse of fragments too small to pay for the overhead of prologues and epilogues.
… must replace each copy of the fragment with a call.	… can "fall into" the first copy for free.
… delimit fragments in the *callee*, namely with the prologue and epilogue.	… delimit fragments in the *caller*, namely with the echo instruction.
… save state in visible registers and memory.	… save state only in invisible registers, namely the stack of active echo instructions.
… permit arbitrary internal control flow and labels.	… forbid internal control flow and labels.

The initial bytecode

The initial implementation of echo instructions was built on a pre-existing bytecode interpreter. Its bytecode is a simple postfix encoding of `lcc` trees [Fraser and Hanson] and nearly identical to that used in a recent paper [Evans and Fraser] on a very different, grammar-based method for bytecode compression.

Most operators consist of an un-typed or generic base (such as `ADD`) followed by a one-character type suffix (`I` for integer, `F` for float, etc), which indicates the type of value produced. The appendix lists all of the operators that appear in the bytecode. There are 99 valid operator-suffix pairs, leaving 256-99=157 codes for use in echo instructions.

All operators are encoded by a single byte, but a few are followed by one or more literal bytes. For example, `LIT2` is followed by a two-byte constant, which it simply pushes onto the stack. Branch offsets and global addresses aren't known until after compression, so they are encoded using one level of indirection. That is, the instruction stream includes not the actual address but rather a two-byte literal index into a table that holds the actual address.

The representation has two other elements, namely procedure descriptors and trampolines for inter-operation with existing libraries and conventional, non-bytecoded procedures. These elements are not bytecoded and thus not subject to compression with echo instructions, so they are described elsewhere [Evans and Fraser]. Only two material changes were made to this previous representation. First, the bytecode was formerly separated by procedure but has now been concatenated into one long string, in order to allow echo instructions to reach back and reuse bytecode from other procedures. Second, the previous grammar-based compression has been replaced with compression using echo instructions.

The compressor

The compressor accepts a program in this bytecode and emits an equivalent program in which echo instructions replace repeated phrases. This compressor changes branch offsets and needs to see all labels, so it naturally operates on an assembly-code representation of the bytecode, which uses symbolic labels. After compression, the code is "assembled" into a binary format for execution, though the assembly process is so simple that is accomplished by C macros.

This compressor could adapt any of the methods used by LZ compressors, which range from linear search within the window reachable by the widest displacement, to hash tables, to Patricia trees [Bell, Cleary, and Witten]. This work focuses on compression ratios, not compressor speed, so it uses a simple method that is easily modified during experimentation. Namely, the compressor maintains a hash table that maps each operator to a list that holds the address of each of the last

twenty occurrences of said operator. This limit reduced compression for typical programs only trivially, but it improves compression time significantly.

When the compressor is at position P, it compares the sequence of instructions at P with the sequences at each of these twenty earlier positions. The longest match is the winning phrase. If the echo instruction that encodes that phrase is shorter than the phrase itself, then the compressor emits the echo instruction and skips forward to the end of the phrase. Otherwise, it emits the instruction at P and skips forward by one instruction.

An echo instruction can pay for even one-instruction phrases, if the original instruction includes literal bytes, and if the echo instruction is short enough. For this reason, careful allocation of bytecodes to echo instructions is important.

Encoding echo instructions

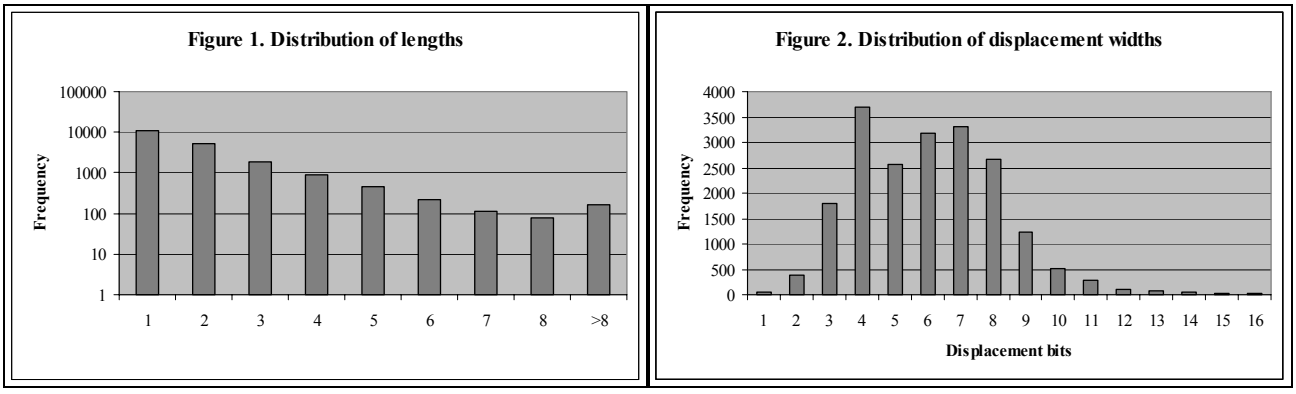
Adding echo instructions to an instruction set requires choosing an encoding. One approach might use Tunstall coding [Tunstall] or perhaps Huffman coding [Huffman] constrained to emit codes with lengths that are multiples of eight bits. The results, however, would make it impossible to separate the effect of the Tunstall or Huffman coding from the effect of the echo instruction. The result also is likely to be harder for the interpreter to decode than the traditional hand-designed instruction set, which is designed with the interpreter in mind. Tunstall and Huffman coding have thus been saved for future work, and a hand-designed echo instruction is used below.

Since the uncompressed bytecode has only 99 distinct operators, it is possible to allocate bytecodes 128-255 to specialized echo instructions and still have a few escape codes left over for longer variations. For example, bytecodes 128-255 might be treated as a seven-bit instruction for phrases that can be referenced with a two-bit length and a five-bit displacement.

Competing for the scarce bytecodes are the length and displacement fields, the offset field for extended echo instructions and the opcodes themselves. To guide the choice for the bytecoded instruction set above, some measurements were taken. The initial round of measurements focused on sizing the length and displacement fields and thus ignored the offset field for extended echo instructions. The bytecode for `lcc` was compiled and compressed assuming—naively—that the first byte might be able hold the opcode, length, and the first seven bits of displacement, and that the opcode could specify whether the displacement added 1, 2, 3, or 4 bytes.

Figure 1 shows the distribution of phrase lengths. Short phrases far outnumber longer ones, so the frequency axis uses a log scale. A 3-bit length field could handle values 1-8, which could handle 99% of the phrases in this sample, a 2-bit field could handle 95%, and a 1-bit field could handle 81%. Even a 0-bit field—that is, an echo instruction specialized to phrases of length 1—could handle 54%.

Figure 2 shows the distribution of the *widths* of displacements. Perhaps surprisingly, one-byte displacements can capture most of the benefit, but it makes sense to allow longer variations as well.



The initial round of measurements, which is summarized in Figures 1 and 2, suggests trying length fields of 0-3 bits and displacements ranging up to 13 bits. It suggests offering short encodings for length fields as narrow as zero bits and displacements as narrow as 4 bits. Thus the input was recompressed using the trial encodings below:

1. Use bytecodes 128-255 for echo instructions with length of one (that is, a zero-bit length field) and a displacement that fits in seven bits. When this won't do, escape to a three-byte form, composed of a one-byte opcode plus two literal bytes that hold a three-bit length and a 13-bit displacement.
2. As above, but the one-byte encodings use a one-bit length and a six-bit displacement.
3. As above, but the one-byte encodings use a two-bit length and a five-bit displacement.

4. Use bytecodes 128-255 plus one literal byte to encode a three-bit length and a twelve-bit displacement. Use no escape codes.

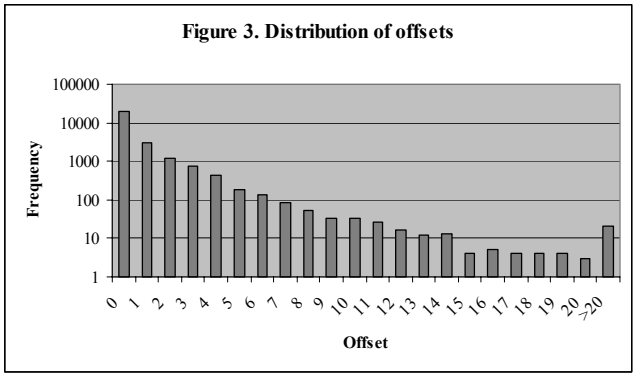
All trials above share a fully general nine-byte echo instruction (a one-byte opcode, two-byte length, two-byte offset, and four-byte displacement) for those few cases with a length or displacement that won't fit in the shorter encodings. Table 2 shows the results of these trial encodings on `lcc`. Encoding 1 wins.

Table 2. Encoding trials

Size	*Ratio*	*Encoding*
194,683	1.000	Uncompressed
128,570	0.660	Encoding 1: zero-bit lengths
130,930	0.673	Encoding 2: one-bit lengths
136,272	0.700	Encoding 3: two-bit lengths
131,744	0.677	Encoding 4: no one-byte forms

Now for the extended echo instructions, which point back to another echo instruction and add an offset field that tells where the current phrase starts in the previous phrase. That is, the offset field specifies the number of primitive (that is, non-echo instructions) to skip from the earlier phrase. In order to size the offset field, `lcc` was recompiled using Encoding 4 above and with the (naïve) assumption that the extra field would require no more bits. Figure 3 shows the resulting distribution of offsets. Short offsets predominate, so the frequency uses a log scale. Offset zero is included to include the frequencies for the basic echo instruction.

For the data in Figure 3, zero offsets outnumber the next most common offset by a factor of seven, so the basic echo instructions (that is, those with zero offsets) should keep the short two-byte encodings assigned above, and the extended echo instructions should use a longer encoding. Three bytes should do, because over 99% of the non-zero offsets fit in a four-bit field. Opcodes 100-115 can be used to encode both the operator and the offset, and two literal bytes can encode the three-bit length and a thirteen-bit displacement.



Performance

Echo instructions are very common in the compressed code. When `lcc` is compressed using the encoding defined above, they are the most common instruction and account for over 40% of the instructions in the final program.

Table 3 gives the compression performance on a set of benchmarks for echo instructions using the encodings above. Unlike conventional compiler benchmarks, which must vary such factors as instruction mix, program size, and demand for working set and instruction and data cache, benchmarks for simple code compression need mainly vary program size and to avoid replicated code. It is important to vary program size because small programs can be hard to compress, and because large inputs are needed to test large phrase displacements. `copt` is a rule-based peephole optimizer for assembly code, and the rest of the benchmarks in the table should be familiar to those experienced with compilers. `lcc`'s automatically generated code generators were removed because they could artificially inflate compression performance.

Table 3. Compression performance

	no compression	*compressed with echo*		*compressed with `gzip`*	
	bytes	*bytes*	*ratio*	*bytes*	*ratio*
8queens	429	262	0.611	185	0.431
copt	2,643	1,851	0.700	1,034	0.391
iburg	12,192	8,387	0.688	4,239	0.348
gzip	45,872	30,448	0.664	15,481	0.337
lcc	194,683	128,570	0.660	60,974	0.313
gcc	1,389,197	943,806	0.679	438,542	0.316

The compression performance for `gzip` [Adler and Gailly], which also uses LZ compression, is included only to suggest the rough cost of requiring byte addressability and of prohibiting control flow in phrases. Any comparison unfairly favors `gzip`, which does not support direct interpretation and thus is free to include control flow in phrases and free to Huffman-code both literals and pointers, since it uses front-to-back decompression and need not support the byte-addressability commonly assumed by bytecode interpreters.

One compressor in the related work [Evans and Fraser] uses the same bytecode and has the same constraint for direct interpretation without decompression. It thus presents an unique opportunity for direct, fair comparison that is unavailable for the other related works below, at least not without access to or reimplementation of their code.

This previous compressor [Evans and Fraser] yielded compression ratios slightly better than `gzip`'s, which means that the echo instruction achieves roughly half of what might be achieved in a directly interpretable bytecode. The previous compressor was, however much more complex, and its interpreter required two levels of interpretation or a much larger interpreter. The echo instruction thus presents a useful alternative. It offers non-trivial compression at a very low cost to the interpreter, where the previous compressor competes with the highly-tuned and less constrained `gzip`, but at the cost of an interpreter that is significantly slower or larger.

Related work

The literature on code compression has grown rapidly in the last few years [van de Weil], but much of it assumes a decompression step (perhaps into a hidden cache) and thus can use techniques that are incompatible with typical bytecode (and machine-code) interpreters, such as Huffman coding. This section thus confines itself to representatives of the methods applicable in this scenario, namely dictionary-based compression that is compatible with direct interpretation.

Liao, Devadas, and Keutzer describe an instruction that is, on the surface, identical to the echo instruction, but it differs in one important way, with the result that the two instructions suit somewhat different scenarios [Liao; Liao, Devadas, and Keutzer]. The instruction

```
CALD offset,len
```

has the same signature as the (basic) echo instruction, and it directs the processor to execute `len` instructions at address `offset`, but it references a separate "dictionary" memory, not the one that holds the `CALD` and the main instruction stream. This modest semantic difference results in very different styles of use, each with its own pros and cons:

- `CALD`'s separate memory permits an important reordering optimization. Namely, a set of useful code fragments can be identified and then ordered to exploit overlap. For example, if one fragment is the triple A-B-C, and another is B-C-D, then the optimizer can store the sequence A-B-C-D in the dictionary, and two `CALD`'s can share the single copy of B-C. Balancing this advantage is the cost of the separate dictionary memory.
- Echo instructions make the symmetric trade-off. They take their fragments from code that's already present in the main memory and thus incur no cost for the dictionary memory, but they can't reorder fragments to improve overlap.

Compression ratios ranging from 0.681-0.919x have been reported for the `CALD` instruction [Liao, Devadas, and Keutzer]. These numbers are, however, not directly comparable to those for the echo instruction above, for a variety of reasons. Chief among these is the fact that `CALD` was applied to optimized machine code and the echo instruction to unoptimized bytecode, which is probably more easily compressed. The relative performance of echo and `CALD` instructions remains an experiment for the future.

Other dictionary-based code compressors [Bird and Mudge; Ernst et al; Evans and Fraser; Latendresse; Lefurgy et al 1997; Lucco; Proebsting] associate a fixed code with the most common sequences in either the program at hand or a training set that is used to generate a code intended to suit all programs. This approach can be particularly advantageous on short programs and at the beginning of all programs, where the echo compressor has little prior context to exploit. On the other hand, a fixed code is naturally less effective when the program at hand is not well represented by the test suite or, in the case of per-program dictionaries, by a single, typically small dictionary. An advantage of LZ compression is that it effectively changes the dictionary as it progresses through the input. It seems likely that some inputs suit a fixed dictionary and others suit the changing dictionary offered by LZ77's sliding window. Thus neither approach seems likely to dominate, and the code compression community benefits from a menu that includes both methods.

Discussion

The echo instruction seems simple enough to implement in hardware, perhaps given limits on nesting depth. Such limits are easily enforced by the compiler, assembler, or post-processor as it creates echo instructions to replace repeated phrases.

Hardware implementations seem likely to worsen compression. For example, two phrases that are identical in postfix bytecode can differ after register allocation or instruction scheduling. On the other hand, even thoroughly optimized code includes repeated phrases, and the cost of the echo instruction appears so low that there's little reason not to exploit it.

The first step toward a hardware implementation is a simulator. Jack Davidson and Kevin Scott are adding an echo instruction to a SimpleScalar [Burger and Austin] simulator for the StrongArm architecture, in order to study both the compression performance and the effect of echo instructions on the performance of the compressed program.

Another natural implementation for echo instructions would be as part of a dynamic translator such as that in Transmeta's Crusoe architecture [Klaiber]. The echo instruction could be added to the emulated instruction set and simply expanded during translation, which would save emulated memory. Such an implementation would be much simpler than changing conventional hardware.

Alternately, the echo instruction could be added to the emulator's internal instruction set. This experiment would be more costly, because it would require changing actual hardware, but the hardware can be changed without changing any legacy software, so this path would offer echo compression to benefit a wider audience.

Acknowledgments

This work benefited from discussions with Jack Davidson, Will Evans, Dave Hanson, Todd Proebsting and Kevin Scott.

References

B. Abali, H. Franke, D. Poff, and T. Smith. Performance of hardware compressed main memory. Research Report RC21799, IBM T. J. Watson Research Center, 7/2000.

M. Adler and J.-l. Gailly. The `gzip` home page. http://www.gzip.org.

T. Bell, J. Cleary, and I. Witten. *Text Compression*. Prentice Hall, 1990.

C. Benveniste, P. Franaszek, and J. Robinson. Cache-memory interfaces in compressed memory systems. Research Report RC21662, IBM T. J. Watson Research Center, 2/2000.

P. Bird and T. Mudge. An instruction stream compression technique. Technical Report CSE-TR-319-96, EECS Department, University of Michigan, 11/96.

D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News 25*(3):13-25, 6/1997. K. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. *PLDI'99*:139-149, 5/99.

S. Debray, W. Evans, R. Muth, and B. de Sutter. Compiler techniques for code compaction. *TOPLAS 22*(2):378-415, 3/2000

D. Ditzel, H. McClellan, and A. Berenbaum. The hardware architecture of the CRISP microprocessor. *Proceedings of the International Symposium on Computer Architecture*, 6/1987.

J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting. Code compression. *PLDI'97:*358-365, 6/97.

W. Evans and C. Fraser. Bytecode compression via profiled grammar rewriting. *PLDI'01:*148-155, 6/2001.

P. Fenwick. Ziv-Lempel encoding with multi-bit flags. *DCC'93*:138-147, 1993.

M. Franz and T. Kistler. Slim binaries. *Communications of the ACM 40*(12): 87-94, 12/97.

C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation.* Addison Wesley Longman, 1995.

C. Fraser, E. Myers, and A. Wendt. Analyzing and compressing assembly code. *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction*:117-121, 6/84.

M. Game and A. Booker. CodePack: Code compression for PowerPC processors. Technical report, IBM Microelectronics Division.

J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel. A code compression system based on pipelined interpreters. *Software Practice and Experience 29*(11):1005-1023, 1999.

D. Huffman. A method for the construction of minimum redundancy codes. *Proceedings. of IRE 40:*1098-1101, 1952.

A. Klaiber. The technology behind Crusoe processors. Transmeta Corporation, 2000. http://www.transmeta.com/about/press/white_papers.html.

K. Kunchithapadam and J. Larus. Using lightweight procedures to improve instruction cache performance. Computer Sciences Technical Report #1390, University of Wisconsin-Madison, 1/99.

S. Larin and T. Conte. Compiler-driven cached code compression schemes for embedded ILP processors. *Proceedings of the 32nd International Symposium on Microarchitecture*:82-92, 11/99.

M. Latendresse. Automatic generation of compact programs and virtual machines for Scheme. *Scheme and Functional Programming 2000*, Montréal, 9/2000.

C. Lefurgy, P. Bird, I. Chen, and T. Mudge. Improving code density using compression techniques. *Proceedings of the 30nd International Symposium on Microarchitecture*, 12/97.

C. Lefurgy, E. Piccininni, and T. Mudge. Analysis of a high performance code compression method. *Proceedings of the 32nd International Symposium on Microarchitecture*:93-102, 11/99.

C. Lefurgy, E. Piccininni, and T. Mudge. Reducing code size with run-time decompression. *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, 1/2000.

S. Liao. Code generation and optimization for embedded digital signal processors. PhD dissertation, MIT, 1996.

S. Liao, S. Devadas, and K. Keutzer: A text-compression-based method for code size minimization in embedded systems. *TODAES 4*(1):12-38, 1999.

S. Lucco. Split-stream dictionary program compression. *PLDI'00:*27-34, 6/2000.

T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. *POPL'95:*322-332, 1/95

B. P. Tunstall. Synthesis of noiseless compression codes. Ph.D. thesis, Georgia Institute of Technology, 1967.

R. van de Wiel. Code compaction bibliography. 10/2001. http://www.extra.research.philips.com/ccb/.

J. Ziv and A. Lempel, A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*:337-342, 1977.

Appendix: Instruction set.

The table below describes the un-typed or generic operators from the initial instruction set. A superscript denotes the number of literal bytes, if any, after the operator.

The only changes from `lcc` [Fraser and Hanson] are literals, the `LocalCALL` operators (which use literals instead of the stack to identify the callee), and comparisons. `lcc` comparisons accept two comparands and a literal branch address, but the comparisons here accept two comparands and push a flag for `BrTrue`, which accepts the literal branch address.

The type suffixes are: `V` for void or no value, `C` and `S` for char and short, `I` and `U` for signed and unsigned integers, `F` and `D` for single- and double-precision floating-point numbers, `P` for pointers, and `B` for instructions that operate on blocks of memory.

Operator	*Comment*
`ADD DIV SUB MUL`	Arithmetic.
`BAND BOR BXOR`	Bit-wise Booleans.
`BCOM`	Bit-wise negation.
`NEG`	Arithmetic negation.
`CVD`	Convert from double.
`CVF`	Convert from float.
`CVI`	Convert from int.
`CVI1 CVI2`	Sign-extend char, short.
`CVU1 CVU2`	Zero-extend char, short.
`EQ GE GT LE LT NE`	Compare and push 0 or 1.
`LSH MOD RSH`	Shifts, remainder.
`INDIR`	Pop p, push *p.
`ASGN`	Pop p and v, copy v to *p.
`ASGNB`^{2}	Pop p and v, copy the block at *v to *p.
`ADDRF`^{2}	Push address of formal.
`ADDRG`^{2}	Push address of global.
`ADDRL`^{2}	Push address of local.
`JUMP`^{2}	Pop label number, jump.
`ARG`	Top is next outgoing argument.
`RET`	Return value atop stack.
`CALL`	Pop p, call routine at address p.
`LocalCALL`^{2}	Call routine at literal address.
`POP`	Discard top element.
`LIT1`^{1} `LIT2`^{2} `LIT3`^{3}`LIT4`^{4}	Push 1-4 literal bytes.
`BrTrue`^{2}	Pop flag. Jump if true.