Split-Stream Dictionary Program Compression

Steven Lucco
Transmeta
3940 Freedom Circle, Santa Clara, CA 95054
slucco@transmeta.com

ABSTRACT

This paper describes split-stream dictionary (SSD) compression, a new technique for transforming programs into a compact, interpretable form. We define a compressed program as interpretable when it can be decompressed at basic-block granularity with reasonable efficiency. The granularity requirement enables interpreters or just-in-time (JIT) translators to decompress basic blocks incrementally during program execution. Our previous approach to interpretable compression, the Byte-coded RISC (BRISC) program format [1], achieved unprecedented decompression speed in excess of 5 megabytes per second on a 450MHz Pentium II while compressing benchmark programs to an average of three-fifths the size of their optimized x86 representation. SSD compression combines the key idea behind BRISC with new observations about instruction re-use frequencies to yield four advantages over BRISC and other competing techniques. First, SSD is simple, requiring only a few pages of code for an effective implementation. Second, SSD compresses programs more effectively than any interpretable program compression scheme known to us. For example, SSD compressed a set of programs including the spec95 benchmarks and Microsoft Word97 to less than half the size, on average, of their optimized x86 representation. Third, SSD exceeds BRISC's decompression and JIT translation rates by over 50%. Finally, SSD's two-phased approach to JIT translation enables a virtual machine to provide graceful degradation of program execution time in the face of increasing RAM constraints. For example, using SSD, we ran Word97 using a JIT-translation buffer one-third the size of Word97's optimized x86 code, yet incurred only 27% execution time overhead.

Keywords

Compression, virtual machine, runtime system.

1. INTRODUCTION

This research aims to increase our ability to trade program size for program execution time in designing computer systems. Specifically, we want to enable computer system designers to store native or virtual machine programs using a smaller amount of system ROM, RAM or disk space, while incurring an insignificant impact on program execution time.

For example, hand-held organizer products currently popular can have as little as 2 megabytes ROM and 2 megabytes RAM to hold all system software, plus add-on software and data [2]. This limits the number and types of applications suitable for these organizers. Since data competes directly with programs for space, the number of contacts or maps that the device can hold will depend directly on the amount of space the device requires to store its programs. In embedded systems with even tighter constraints on program space, such as MEMS [3], the degree to which one can compress system programs determines the capabilities one can pack into the device.

On desktop systems, we can use program compression to increase system performance by taking advantage of large differences in access time among components of the memory hierarchy. For example, we used the program compression scheme described this paper, split-stream dictionary (SSD) compression, to reduce the number of code pages required to start Microsoft `Word97`. Because SSD yields decompression speed of 7.8 megabytes per second on a 450MHz Pentium II, disk latency dominated decompression time and `Word97` started 14% faster than the same version of `Word97` compiled to optimized x86 instructions.

The effects described in both these examples become more pronounced when computer systems use RISC or VLIW [4] instruction sets. These fixed-length program encodings are less compact than the variable length x86 bytecodes [5]. For example, early compiler implementations suggest that programs compiled for the Intel IA64 (Itanium) architecture [6] will require two to three times the code space of the same program compiled for the x86 processor.

Designers of embedded system processors have attempted to increase program encoding density by introducing 16-bit versions of their instruction sets or by adding complex features to their designs. For example, the ARM [7] computer architecture includes a 16-bit instruction set, called Thumb, which is used to provide program compression. The ARM architecture converts Thumb instructions back to ARM instructions during the decode pipeline stage, sacrificing chip area in an attempt to increase program density. Similarly, the ARM departs from RISC discipline by spending chip area on features, such as auto-increment addressing, designed to reduce code size.

Hence, the current evolution of embedded system processor designs illustrates the pressure that program storage cost exerts on embedded processor architecture. In adding complex features such as the Thumb instruction set or auto-increment addressing, ARM designers implicitly trade program density against program execution time. In contrast to these fixed-hardware approaches, SSD compression can reduce a program's use of ROM, RAM and disk space without significantly increasing a program's execution time.

Specifically, SSD compressed a test suite of programs compiled for the Omniware virtual machine (OmniVM) [8,9], including Microsoft `Word97` and the `spec95` benchmarks, to an average of 47% the size of their optimized x86 representations. When incrementally decompressed, JIT-translated, and executed by the OmniVM, these programs ran an average of 6.6% slower than the optimized x86 versions, demonstrating that SSD supports fast JIT-translation of processor-neutral code. Further, execution-time profiles of these programs revealed that SSD decompression and JIT translation contributed no more than 0.7% to any program's execution time; limitations on JIT-translated code quality accounted for most of the execution time overhead.

Section 3 of this paper provides more details of these experiments; it also reports the impact of varying JIT-translation buffer size on program execution time. The results of this experiment show that SSD compression supports graceful degradation of program execution times as JIT-translation buffers shrink. SSD achieves this result by supporting two-phased JIT-translation. In the phase one, the virtual machine loads and decompresses a *dictionary*. The dictionary maps 16-bit indices to sequences of one to four instructions. During phase two, the JIT-translator expands a basic block by copying dictionary entries into a native code buffer. Because phase two translation consists mostly of copying memory blocks, it is fast. For our set of benchmark programs, we measured the average speed of 12.5 megabytes of produced code per second for phase two translation. Once the virtual machine pays the fixed cost of dictionary decompression, it can translate and re-translate parts of the program at this phase two translation speed. This feature enables a virtual machine to achieve reasonable program execution times even when using a native code buffer significantly smaller than the program being executed.

The remainder of this paper is organized as follows. Section 2 places SSD compression in the context of previous approaches to program compression. It then explains in more detail why and how SSD compression works and describes how to incorporate SSD compression into program loaders and virtual machine implementations. Section 3 provides a quantitative assessment of SSD. Finally, section 4 summarizes.

2. SSD COMPRESSION

Many compression techniques encode their input using a dictionary. In general, a compression dictionary stores common input patterns. All or part of a compressed input consists of compact references to the dictionary. When the dictionary does not depend on the input, we call it external. If the dictionary depends on input but does not change during decompression, we call it static; otherwise, we call a dictionary dynamic. Lempel-Ziv compression uses a dynamic dictionary [10]. As LZ decompresses data, it stores each novel sequence of bytes in a dictionary. Items further back in the stream of compressed data can refer to these implicitly generated dictionary entries using a byte offset and a length [11].

Because LZ compression uses a dynamic dictionary, it is stream-oriented. An LZ decompressor can't randomly access and decode a particular basic block or function. Arithmetic coding strategies, which have yielded the most effective archival program compression solutions known to us, share this limitation with LZ compression [12,18,19].

To support fast in-place interpretation or JIT-translation of compressed programs, we need to design a program compression scheme capable of fast decompression at basic block granularity. We designate as interpretable any program compression scheme that fulfills this criterion.

Program	Optimized x86 Size (bytes)	Total Instructions/ Unique Instructions	Average Re-use Frequency for an Instruction	Unique Digrams	Average Re-use Frequency for a Digram	Avg. Re-use Freq. Of Most Common Instruction Sequences (top 10%)
`Word97`	5175500	1427592/124288	11.5	518351	2.8	16.6
`Gcc 2.6.3`	747436	194501/22946	8.4	78413	2.5	12.5
`Vortex`	400040	97931/11828	8.3	34657	2.8	12.8
`Perl`	238950	75270/11664	6.5	34043	2.2	9.5
`Go`	180838	36398/6133	5.9	17568	2.1	10.0
`Ijpeg`	136070	31057/7893	3.9	19207	1.6	8.5
`M88ksim`	119782	21957/5865	3.7	11403	1.9	3.4
`Xlisp`	75942	13414/1860	7.2	5549	2.4	7.4
`Compress`	7234	1411/591	2.4	1032	1.4	5.2

Table 1: Redundancy of instructions occurring in benchmark programs. All columns reflect instruction-matching algorithm that compares sizes but not specific values of pc-relative branch targets. The last column reports the average re-use frequency for the 10% of instruction sequences (lengths 2-4) that were most common.

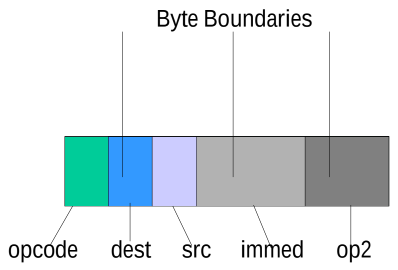

Figure 1: Instruction semantics often do not match byte boundaries.

We can clarify further the class of interpretable program compression schemes by describing why some related efforts don't fit into this category. Java class files [13] are directly interpretable, but are not compressed; they are often larger than the native-compiled version of a given Java class [14]. Further, Java class files can not efficiently represent programs written in many other programming languages, such as C++. ANDF programs [15] and slim binaries [16] represent programs at a high level of abstraction, similar to abstract syntax trees (ASTs) [17]. Hence, they represent programs in a form that requires significant further compilation following decompression. For this reason, AST representations such as these are not examples of interpretable program compression.

Among previous approaches to interpretable program compression, the BRISC program format is the most effective known to us. BRISC compresses programs to about 61% the size of their optimized x86 representation and supports JIT translation at over five megabytes per second [1]. Like the best stream-oriented program compression methods [12], BRISC excels by considering non-byte-aligned quantities in its input stream. Compression methods such as LZ are byte-oriented; they assess similarities among input patterns in terms of byte comparisons. However, most of the information within a virtual or native machine language program, for example opcodes or register numbers, is not aligned on byte boundaries (see Figure 1 for an illustration). Program compression methods that consider the individual fields within instructions are called *split-stream* methods [1]. BRISC and other split-stream compression techniques conceptually split the input stream of instructions into separate streams, one for each type of instruction field. In contrast to BRISC, SSD uses a split-stream method to compress not the entire program but only a dictionary of instruction sequences derived from the program. SSD compresses a program into two parts: a split-stream compressed dictionary and a stream of SSD items.

SSD is significantly simpler to implement than BRISC. BRISC requires the generation and maintenance of a corpus-derived set of instruction patterns designed to capture common opportunities for combining adjacent opcodes and for specializing opcodes to reflect frequently occurring instruction-field values. A virtual machine implementing BRISC will have to load and decode this external dictionary of instruction patterns (approximately 2000 instruction patterns or 150 kilobytes of data). Also, systems implementing BRISC must maintain a separate program to generate the external dictionary of instruction patterns from a training corpus of representative programs.

Input	*P*: a sequence of instructions
Outputs	*D*: an SSD dictionary
	E: a sequence of references to entries in *D*
Variables	*Cur*: a sequence yields of instructions
	Entry: a dictionary entry
	Target: pointer to branch target instruction
Operators	Ntail(*S*,*n*): If sequence *S* has length L_S, returns the suffix of *S* with length L_S-*n*
	Head(*S*): returns first element of sequence *S*
	Append(*S*,*e*): appends *e* to end of sequence *S*
	GetEntry(*D*,*s*): returns dictionary entry matching instruction sequence *s*; creates entry if necessary.
	NewRef(*entry*,*tgt*): returns structure containing reference to dictionary *entry* and branch target *tgt*
	GetBranchTarget(*S*,*L*): return branch target, if any, of L^{th} instruction in sequence *S*

Table 2: Variables used in Algorithm 1.

Further, BRISC's compression effectiveness depends on the applicability of the training corpus. In contrast, SSD embeds an input-specific dictionary into each compressed program. When its input is large (30 kilobytes or more), SSD is not only simpler than BRISC but also compresses programs more effectively (see Section 3 for details).

SSD achieves this combination of simplicity and effectiveness by exploiting the surprisingly high frequency with which large programs re-use small sequences of instructions. Table 1 summarizes single instruction re-use frequency for our set of benchmark programs. These measurements show that our

```
1. Make each unique instruction in P a base entry of D
2. Cur=P; E=the empty sequence
3. while (Cur not empty)
     a. find the longest sub-sequence of instructions
        s, with length L, such that:
          i.   Cur contains at least L instructions
               and L<=4
          ii.  s matches the first L instructions in
               Cur
          iii. s occurs at least twice in P
          iv.  s is contained within a single basic
               block of P
     b. if L>=2 then
          i. Entry=GetEntry(D,s)
     c. else
          i. Entry=GetEntry(D,Head(Cur))
     d. target=GetBranchTarget(Cur,L)
     e. Append(E,NewRef(Entry,Target))
     f. Cur=Ntail(Cur,L)
```

Algorithm 1: Dictionary Generation

benchmark programs re-use each of their instructions an average of 2.4 to 11.5 times. Further, all programs whose x86 optimized code is at least 150 kilobytes in length re-use each of their instructions an average of 5.9 to 11.5 times. The data in Table 1 reflects an instruction-matching algorithm that compares the sizes but not the specific values of pc-relative branch targets. All other instruction fields must match exactly. Section 2.1 explains why SSD handles branch targets in this way. Table 1 shows that re-use frequencies drop off for sequences of two instructions; however, it also shows that the benchmark programs rabidly re-use their favorite two- to four-instruction idioms.

2.1 Overview of SSD

SSD takes advantage of this phenomenon by constructing a dictionary that contains two types of entries. First, the dictionary contains an entry for each individual instruction that occurs in the program; we call these base entries. Second, the dictionary contains an entry for each two- to four-instruction sequence that occurs two or more times in the input program; we call these sequence entries.[1] If an input program were to avoid re-using any instructions, the dictionary would be identical to the input program and SSD's output would be larger than the input. Fortunately, the measurements in Table 1 show that large programs make extensive re-use of single instructions and short instruction sequences.

Once it has constructed the dictionary for a given program *P*, SSD matches the instructions in *P* against the dictionary. Call the first instruction in *P i1*, the next *i2*, etc. SSD matches the sequence <*i1,i2,i3,i4*> against all four-instruction sequences in the dictionary. If it finds a match with sequence entry *e* it outputs an SSD item that refers to *e* and then continues matching with instruction *i5*. Otherwise, SSD tries to match <i*1,i2,i3*> against all three-instruction sequences in the dictionary, and so on. Finally, if no sequence entries match the current input, SSD will output an SSD item that refers to the base entry matching *i1*; then SSD will continue matching with instruction *i2*. SSD continues matching input instructions against the dictionary and generating SSD items until the input is exhausted.

SSD items refer to dictionary entries using 16-bit indices. A dictionary of 2^{15} entries proved sufficient for all benchmark programs except `Word97`; `Word97` required 281,107 entries. If a dictionary requires more than 2^{16} entries, SSD partitions the dictionary into a common dictionary that applies to the entire compressed program, and a series of sub-dictionaries that apply only to parts of the compressed program.

In addition to a 16-bit index, an SSD item may also contain a pc-relative offset representing an intra-function branch target. A dictionary entry contain at most one branch instruction. In sequence entries, the branch instruction is always the last instruction of the sequence; no dictionary entry spans more than one basic block.

For two reasons, SSD represents intra-function branch targets as pc-relative offsets in the stream of SSD items rather than as absolute instruction addresses inside dictionary entries. First, pc-relative offsets are more compact than absolute addresses. Second, this enables SSD to ignore pc-relative offset values when comparing branch instructions during dictionary construction. Instead of matching the exact value of pc-relative offset fields, SSD matches only the size of pc-relative offsets. This choice sharply reduces dictionary size but requires that the stream of SSD items explicitly represent pc-relative offsets. For our set of benchmark programs, this choice yielded compressor output an average of 6.2% smaller than the output of a compressor configured to represent branch targets as absolute values within dictionary entries.

[1] We'll see below that the definition of sequence entries is slightly more complex. The SSD dictionary will not contain a *k* instruction sequence s_1 if every occurrence of s_1 is subsumed by a sequence s_2 that is *k*+1 or more instructions long.

2.2 SSD Details

The previous sub-section provided an overview of SSD compression. In this sub-section, we describe in more detail the three main algorithms of SSD: dictionary construction, dictionary compression and SSD item generation.

Given an input program *P*, SSD uses Algorithm 1 to generate a dictionary *D* and a sequence *E* of dictionary entries. Algorithm 2 (detailed below) converts a dictionary entry sequence *E* to a sequence of SSD items.

To implement Algorithm 1, we use two hash tables and an additional pass over the input. The first hash table (H_I) contains individual instructions; the second (H_D) contains digrams of adjacent instructions. Before executing Algorithm 1, our SSD implementation reads the entire program, constructing these two hash tables. To implement step 1 of Algorithm 1, our implementation makes each element of table H_I a base entry of dictionary *D*. The remainder of Algorithm 1 constitutes a second pass through the input program *P*. Conceptually, the algorithm matches prefixes of lengths 2-4 of the remaining instructions (*Cur*) against the entire program (*P*), attempting to find a sequence of instructions (*s*) that matches the prefix and occurs at least twice in *P*.

To accomplish this operation, our implementation matches the prefix of length 2 against the digram hash table (H_D). For each digram *d* occurring at least twice in *P*, H_D contains a list of all the program addresses at which *d* occurs. To implement step 3.a, our implementation traverses this list, matching the instructions at the front of Cur against up to four of the instructions found at each location of the matched digram *d* within the program *P*. The implementation compares the longest match, if any have length >= 2, with the sequence entries already in *D*. If *D* does not already contain a sequence entry for matching instruction sequence *s*, step 3.b.i creates a new sequence entry and adds it to *D*.

When a match is found, then step 3.f of Algorithm 1 sets *Cur* to begin at the next instruction after the matched prefix. This step yields a greedy algorithm, because by skipping over instructions once it has found a match, Algorithm 1 ignores the possibility of finding a longer match beginning at one of the other instructions in the matched prefix. In any case, step 3.e appends to output sequence *E* the dictionary entry (*entry*) obtained during steps 3.a and 3.b.

It is important to emphasize that in the case of branch instructions our method for comparing instructions is more complex than simple equality. Two branch instructions *a* and *b* will match when their pc-relative branch target fields are equal in size and all other fields are exactly equal. A dictionary entry e_b containing a branch instruction specifies only the size sz_b in bytes of e_b's target. Each SSD item referring to e_b supplies a pc-relative branch target of size sz_b.

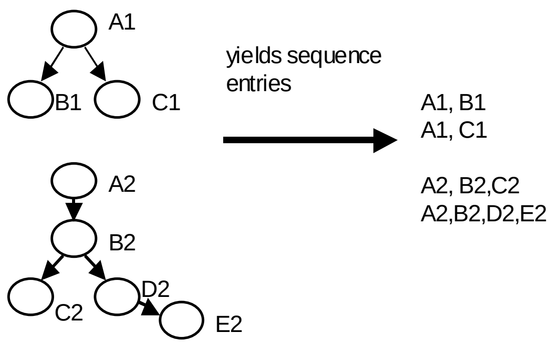

Figure 2: Tree representation of four sequence entries.

2.2.1 Base Entry Compression

To compress the base entries of dictionary *D*, SSD first sorts the entries by opcode, creating an *instruction group* for each opcode. Within each instruction group, SSD sorts the entries by the largest instruction field for that group's opcode. For example, SSD sorts `call` instructions by target address, but sorts arithmetic-immediate instructions (e.g. `add r1,r2,45`) by their immediate field. Of course, the details of this sorting step depend on the particular instruction set of the input program. For the measurements reported in Section 3, the input instruction set was the OmniVM virtual machine instruction set.

Within an instruction group, each instruction field is output as a separate stream. For example, for an add immediate instruction group (with pattern `add reg1,reg2,imm`), our implementation would first sort the group by the `imm` field and then output all `imm` fields followed by all `reg1` fields followed by all `reg2` fields.

We tried two techniques to further compress the base entries. First, we encoded the sorted field (in our example, the `imm` field) using *delta coding*. Delta coding expresses each value as an increment from the previous value (with suitable escape codes for occasional large deltas). All other fields are output literally. Second, we concatenated all of the sorted instruction groups and then applied a simple form of LZ compression to the result. This latter approach was simpler and yielded better compression. It is used for all experiments reported in Section 3.

2.2.2 Sequence Entry Compression

SSD compresses the sequence entries of a dictionary *D* by constructing a forest of trees, one tree for each instruction *i* that can start a sequence. A given tree t_i represents all of the sequences in *D* that start with *i*. If two such sequence entries in *D* share a common prefix *p* of length *L*, their representation in t_i will share the first *L* nodes. Figure 2 depicts a simple forest of sequence entries. SSD stores each tree as a sequence of 16-bit indices that refer to base entries of *D*. SSD stores these indices in prefix order. If *D*'s base entries number 2^{15} or fewer, SSD represents the tree structure using the high-order bit of each index. If *D* has more than 2^{15} base entries, SSD uses a special index value to mark upward tree traversal.

2.2.3 SSD Item Generation

Algorithm 2 describes the SSD algorithm for emitting SSD items. In practice, we need to perform some extra bookkeeping to

Input	*E*: a sequence of pairs <*R*,*t*> where *R* refers to a dictionary entry and *t* is a branch target
Output	*I*: a sequence of SSD items, one for each element of *E*
Variables	*Ref*: a pair <*R*,*t*> as described above
	Tgt: a branch target
Operators	GetIndex(*R*): returns 16-bit index corresponding to dictionary entry referred to by *R*
	NewItem(*indx*,*tgt*): given an index indx and a branch target, *tgt*, creates an SSD item
	IsBranch(*tgt*): returns true if *tgt* is a valid branch target
	ConvertTarget(*I*,*tgt*): given a branch target *tgt*, converts it to a branch target expressed relative to the end of SSD item sequence *I*

Table 3: Variables used in Algorithm 2

support step 3 of Algorithm 2. For each forward branch processed in step 2.b.i, Algorithm 2 must create and save a *relocation item*. Each relocation item points to an SSD item br_i in *I*. The relocation item also contains the intended target of the forward branch br_i in terms of Algorithm 2's input sequence *E*. Then, in step 3, Algorithm 2 traverses its list of relocation items, overwriting the pc-relative branch target values once their target addresses in *I* are known. To compute these target addresses, Algorithm 2 maintains a forwarding table that maps items in sequence *E* to items in sequence *I*. Algorithm 2's ConvertTarget operator immediately looks up backward branches in this forwarding table, but for forward branches, it creates a relocation item.

2.2.4 JIT Translation

In this sub-section, we describe SSD decompression; we also show how to incorporate SSD decompression into virtual machine (VM) systems that incrementally translate compressed programs into native instructions.

We divide SSD decompression into two phases, the first called the *dictionary decompression* phase, and the second called

```
1. Cur=E
2. while (Cur not empty)
       a. Ref=Head(Cur)
       b. If (IsBranch(Ref.t)) then
              i. Tgt=ConvertTarget(I,Ref.t)
       c. else
              i. Tgt=null
       d. Append(I,NewItem(GetIndex(Ref.R),Tgt))
3. Fix branch targets for forward branches
```

Algorithm 2: SSD Item Generation

Inputs	I*buf*: buffer containing SSD items
	Start: address of first item to translate
	End: addess just past last item to translate
	Itab: instruction table produced by dictionary decompression
Output:	*Jbuf*: JIT-translation buffer containing native instructions
Variables	*Ptr*: pointer to current SSD item
	Copylen: number of instruction bytes to copy
	Iptr: pointer into instruction table
	Jptr: pointer into JIT translation buffer
Operators	GetLength(*itab,item*): use *itab* to find length in bytes of instructions to be copied for *item*
	GetPointer(*itab,item*): return pointer to instructions to be copied
	IsBranch(*itab,item*): returns true if item refers to instruction sequence ending with branch

Table 4: Variables used in Algorithm 3.

the *copy* phase. During dictionary decompression, the VM first reconstructs the base entries of the compressed program's dictionary, reversing the compression steps described in Section 2.2.1. Second, the VM reconstructs the sequence entries of the dictionary by traversing the tree that represents these entries.

If the original input to SSD contained virtual machine instructions, then the VM performs additional work during the first of these dictionary decompression steps. As the VM generates base entries, it converts them from virtual machine instructions to native instructions. This type of conversion is appropriate only for virtual machine instruction sets (like OmniVM) that accommodate optimization, since the conversion is done by translation of individual instructions, rather than optimizing compilation. Of course, the VM can take a hybrid approach by further optimizing each function once it has generated the native code for that function. For example, the OmniVM can optionally perform machine-specific basic block instruction scheduling on its generated native code.

The organization of the base dictionary entries facilitates rapid conversion from virtual to native instructions. Since SSD arranges these entries into instruction groups sorted by opcode and largest field value, much of the work needed to translate a particular instruction can be shared among the instructions in a group.

SSD dictionary decompression produces an *instruction table* of native instructions organized to support the copy phase of SSD decompression. The instruction table maps the 16-bit indices found in SSD to sequences of native instructions. Each entry in the instruction table begins with a 32-bit tag. The tag provides the length of the ensuing instruction sequence. If the instruction sequence ends with a branch instruction *b*, the tag provides a negative offset from the end of *b*; this offset indicates where within *b* to copy the pc-relative branch target *t* that will be supplied by the SSD item. Instruction *b*'s opcode determines *t*'s size.

```
1.  ptr=start;jptr=jbuf
2.  while (start<end)
        a.  item=ibuf[ptr]
        b.  copylen=GetLength(itab,item);
            iptr=GetPointer(itab,item)
        c.  copy copylen bytes from iptr to jptr
        d.  jptr=jptr+copylen
        e.  if (IsBranch(itab,item) then
                i.  get branch target from item
                ii. if forward branch or function call
                    then create relocation item for
                    branch target field else convert
                    branch target to pc-relative offset
                    and overwrite target field in copied
        f.  ptr=ptr+size of item in ibuf
3.  Apply relocation items to fix up forward branches and
    call targets
```

Algorithm 3: Copy Phase of SSD Decompression

Algorithm 3 details the copy phase of SSD decompression. This phase of SSD decompression can take place incrementally. The Omniware virtual machine implementation uses SSD decompression to perform JIT translation one function at a time. In Algorithm 3, this would correspond to setting *start* to point to the beginning of the function and *end* to point just past the function. There are three paths through step 2 of Algorithm 3, depending on whether the translated SSD item contains a forward branch or call, a backward branch, or only non-branching instructions. The latter path occurs most frequently and requires only 7+*n* x86 machine instructions to complete, where *n* is the number of bytes of native instructions copied.

3. PERFORMANCE EVALUATION

The decompression algorithms described in the previous section are designed to support rapid, incremental decompression and JIT translation of highly compressed programs. In this section, we provide a quantitative evaluation of how well SSD achieves these goals. We will present results from three sets of experiments. In the first experiment, we compared in size SSD-compressed, optimized OmniVM versus optimized-x86 representations of a set of benchmark programs, including the `spec95` benchmarks and `Word97`. Second, we measured the impact of SSD decompression and JIT translation on the execution time of our benchmark programs. Finally, we limited the size of the buffer used to hold JIT-translated native instructions and measured the impact of this limitation on `Word97` execution times.

We performed all of these experiments on a 450MHz Pentium II processor with 128 megabytes of memory, running Microsoft Windows NT 4.0 service pack 5. We used Microsoft Visual C++ 5.0 at its highest level of optimization to compile our

Program	Optimized x86 Size (bytes)	Ratio of SSD Compressed Size to Optimized x86 Size	Ratio of BRISC Compressed Size to Optimized x86 Size	SSD Execution Time Overhead	SSD JIT Translation and Decompression Execution Time Overhead	SSD Overhead Due to Reduced Code Quality
`Word97`	5175500	0.45	0.69	3.2%	0.7%	2.5%
`Gcc 2.6.3`	747436	0.49	0.57	9.1%	0.4%	8.7%
`Vortex`	400040	0.37	0.55	7.7%	0.4%	7.3%
`Perl`	238950	0.57	0.85	8.6%	0.3%	8.3%
`Go`	180838	0.42	0.60	5.5%	0.2%	5.3%
`Ijpeg`	136070	0.50	0.60	8.1%	0.5%	7.6%
`M88ksim`	119782	0.41	0.49	7.4%	0.3%	7.1%
`Xlisp`	75942	0.43	0.59	5.1%	0.2%	4.9%
`Compress`	7234	0.58	0.57	4.3%	0.2%	4.1%
Average	**786866**	**0.47**	**0.61**	**6.6%**	**0.4%**	**6.2%**

Table 5: Compression effectiveness of SSD compared to BRISC. Also, execution time performance of SSD. All execution time overhead values computed relative to optimized x86 program execution time.

Buffer Size (including dictionary size) / Optimized x86 Code Size	Megabytes JIT-Translated (including re-translation)	Buffer Hit Rate
0.2	208.0	91.31
0.25	119.1	94.35
0.275	53.2	99.83
0.3	13.5	99.87
0.325	9.3	99.89
0.35	7.4	99.89
0.4	6.5	99.93
0.45	6.1	99.95
0.5	5.3	99.96

Table 6: Megabytes JIT-Translated and Buffer Hit Rate Versus Buffer Size for `Word97`

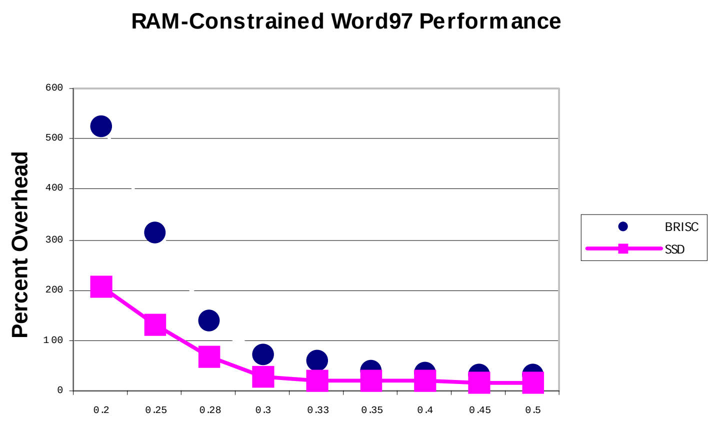

Figure 3: RAM – Constrained `Word97` Performance

benchmark programs. To measure execution time for the `spec95` benchmarks we used the standard benchmark input sets; for `Word97,` we used a performance test suite that includes the `Word97` auto-format, auto-summarize and grammar check.

Table 5 shows SSD compressed the OmniVM benchmark programs to less than half the size, on average, of their optimized x86 versions. Table 5 also compares SSD compression to BRISC compression, illustrating that SSD compresses programs more effectively than BRISC.

In addition, Table 5 lists execution times for our benchmark programs. The measurements demonstrate that SSD decompression does not significantly impact program execution time; execution time overhead averaged just 6.6%. Table 5 breaks this overhead into components, measured using execution time profiling, showing that most of the execution time overhead was due to reduced quality of the JIT-translated native code rather than to decompression overhead. Decompression overhead contributed less than 0.5%, on average, to the total execution time of the benchmarks.

Finally, Figure 3 graphs performance of `Word97` as a function of JIT-translation buffer size, using both BRISC and SSD compression. We varied buffer size from 0.2 to 0.5 times the size of `Word97`'s optimized x86 code. The figure plots execution time overhead against buffer size. In these measurements, the buffer size is computed as the sum of the JIT translation buffer size plus the size of either the SSD dictionary or, for BRISC, the BRISC external dictionary. Also, the infrastructure required to discard and to re-generate JIT-translated code (including a level of indirection for function calls) increases to 14.1% the minimum execution time achievable (versus the JIT-translate-once overhead of 3.2%).

To perform this experiment, we used a buffer space replacement policy that combines round-robin and LRU concepts[20]. The replacement policy breaks the JIT translation buffer into a *permanent* and a *round-robin* area. During program execution, functions that are large and frequently used[2] are moved to the permanent area. To reduce fragmentation, functions smaller than 512 bytes also reside in the permanent area. Table 6 shows that this policy achieves excellent hit rates. For example, when we ran `Word97` using a JIT translation buffer 0.4 times the size of native `Word97`, 99.3% of all function calls were to functions already residing in the buffer. Despite this excellent hit rate, `Word97` required re-translation of 6.5 megabytes of code during the benchmark, yielding a re-translation throughput of 12.5 megabytes per second. These measurements show that the efficiency of SSD copy-phase decompression enables a graceful degradation in program performance, even when the virtual machine must re-generate several megabytes of code during program execution.

4. SUMMARY

Embedded systems can use this graceful degradation of program performance to compactly store system programs in ROM but execute them at near-native performance in a small amount of RAM. Desktop and server systems can use SSD compression to reduce application startup latency. In general, we have demonstrated that SSD compression is a simple but powerful tool that increases our ability to trade program size for program execution time in designing computer systems.

5. ACKNOWLEDGEMENTS

This research was conducted at the Microsoft Bay Area Research Center, San Francisco, CA. Thanks to William Evans, Jim Gray, and the PLDI 2000 Program Committee for their suggestions in revising earlier drafts of this paper.

6. REFERENCES

[1] J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting, “Code compression,” PLDI ’97:358-365, 6/97.

[2] http://www.palm.com/home.html

[3] J. Kahn, R.H. Katz, K.Pister, “MOBICOM challenges: mobile networking for ‘Smart Dust’,” ACM MOBICOM Conference, Seattle, WA, 8/99.

[4] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach,* Addison-Wesley, ISBN 1-55860-329-8.

[5] Intel Corp., *Pentium Processor User’s Manual Volume 3: Architecture and Programming Manual*, Intel Literature Sales, ISBN 1-55512-195-0.

[6] http://developer.intel.com/design/ia64/microarch_ovw/index.htm.

[7] S. Furber, *ARM System Architecture*, Addison-Wesley, ISBN 0-201-40352-8.

[8] S. Lucco, O. Sharp, and R. Wahbe, “Omniware: a universal substrate for web programming,” in *Fourth International World Wide Web Conference,* Boston, Massachusetts, 12/95. http://www.w3.org/Conferences/WWW4/Papers/165/.

[9] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe, “Efficient and language-independent mobile programs,” PLDI ’96:127-136, 6/96.

[10] A. Lempel and J. Ziv, “On the complexity of finite sequences,” *IEEE Transactions on Information Theory* 22(1):75-81, 1/76.

[11] J. Ziv and A. Lempel, “Compression of individual sequences via variable-rate coding,” *IEEE Transactions on Information Theory* 24(5):530-536, 9/78.

[12] C. Fraser, “Automatic inference of models for statistical code compression,” PLDI ’99:242-246, 5/99.

[13] K. Arnold and J. Gosling, *The Java Programming Language,* Addison-Wesley, ISBN 0-201-63455-4.

[14] W. Pugh, “Compressing java class files,” PLDI ’99:247-258, 5/99.

[15] “Architecture Neutral Distribution Format: a white paper,” Open Software Foundation, 11/90.

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

[17] M. Franz, “Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile-object systems,” TR 97-04, Dept. of Information and Computer Science, University of California, Irvine, 2/97.

[18] I. Witten, R. Neal, and J. Cleary, “Arithmetic coding for data compression,” *Communications of the ACM 30*(6):520-540, 6/87.

[19] T. Yu, “Data compression for PC software distribution,” *Software-Practice & Experience 26*(11):1181-1195, 11/96.

[20] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley, ISBN 0-471-94148-4.

[2] Specifically, a function is moved to the permanent area when the product of its size and the number of times it has been translated is greater than the size of the round-robin area.