Code Optimization for Code Compression

Milenko Drinić
Computer Science Department
University of California
Los Angeles, CA 90095
milenko@cs.ucla.edu

Darko Kirovski
Microsoft Research
One Microsoft Way
Redmond, WA 98052
darkok@microsoft.com

Hoi Vo
Microsoft Research
One Microsoft Way
Redmond, WA 98052
hoiv@microsoft.com

Abstract

With the emergence of software delivery platforms such as Microsoft's .NET, reduced size of transmitted binaries has become a very important system parameter strongly affecting system performance. In this paper, we present two novel pre-processing steps for code compression that explore program binaries' syntax and semantics to achieve superior compression ratios. The first preprocessing step involves heuristic partitioning of a program binary into streams with high auto-correlation. The second preprocessing step uses code optimization via instruction rescheduling in order to improve prediction probabilities for a given compression engine. We have developed three heuristics for instruction rescheduling that explore tradeoffs of the solution quality versus algorithm run-time. The pre-processing steps are integrated with the generic paradigm of prediction by partial matching (PPM) which is the fundament of our compression codec. The compression algorithm is implemented for x86 binaries and tested on several large Microsoft applications. Binaries compressed using our compression codec are 18-24% smaller than those compressed using the best available off-the-shelf compressor.

1. Introduction

Compression ratio of program binaries is a parameter that directly impacts several applications with restricted bandwidth and storage resources. Software delivery platforms such as Microsoft's .NET [8], depend tremendously on compression of binaries for two reasons. First, client performance is largely governed by the delay and bandwidth of the communication channel that links the delivery server. Second, the workload on the server is highly impacted with the number of service requests. Both performance parameters are improved with decreased compression ratios of the communicated data: the program binaries. Efficient code compression also benefits applications with restricted memory resources. Traditionally, in most embedded systems ROM and RAM size tremendously impact the system cost [5], thus, posing a balancing factor between system functionality and price. Similarly, wireless systems that extensively use mobile code have energy savings directly proportional to the decrease of the code compression ratio as power consumption due to the device's transceiver activity commonly dominating the energy bill of wireless devices [27].

Compression ratios of program binaries yielded by the best off-the-shelf compression algorithms have their lower bound set at approximately 0.50. Significantly better compression ratios are obtained when, for example, text or multimedia data are compressed. The reason behind such difference in the quality of compression is in the difference in the structure and the nature of data in program binaries as opposed to textual or multimedia data. Textual files usually contain far less than nominal 256 possible characters used in most text encoding schemes. Thus, compression algorithms leverage on inherent redundancy of the encoding to achieve low compression ratios. Furthermore, there is a strong correlation among symbols being encoded in textual files which makes symbol prediction probabilities more accurate. Multimedia data is commonly compressed using lossy compression algorithms, taking advantage of the limited human audio and visual perception.

A glimpse on the limits of compressing binaries can be obtained by compressing raw source code. Although source code gives a sufficient description of program's functionality, it does not map the program to a given architecture abstraction. Nevertheless, such a compression results in ratios that ranges within 0.10 to 0.25 with respect to the compiled binary. As opposed to textual files, program binaries can be changed in such a way that their functionality remains the same while their structure is altered significantly. There exist a long list of possible transformations that can be applied which includes associativity, distributivity, common subexpression elimination, etc. Even finding the optimal list of transformations that minimize uncompressed

Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)
0-7695-1913-X/03 $17.00 © 2003 **IEEE**

IEEE COMPUTER SOCIETY

code size is an NP-hard problem [14]. Program binaries are heterogeneous files, files that contain different types of data interlaced throughout binaries, which makes their compression a hard task yielding poor compression ratios. Intra-correlations of different data types are hard to extract and analyze in a computationally inexpensive fashion. It is important to stress that a number of architectures, including x86 which is the target architecture of our compression algorithm, are already using a form of Huffman encoding for code compaction by a variable length of instructions. The most frequently used instructions such as `PUSH, POP,` and `RET` are encoded with a single byte, while less frequently used instructions such as floating point division are encoded with multiple bytes [16].

In this paper, we present compression mechanisms for program binaries that explore their syntax and semantics to achieve superior compression ratios. The fundament of our algorithm is prediction by partial matching (PPM) [4]. We combine this powerful compression paradigm with the following pre-processing steps:

(*i*) SPLIT^2-STREAM. We have developed an algorithm that initially breaks a program into a large number of sub-streams with high auto-correlation and then, heuristically merges certain sub-streams to: (*a*) achieve the benefits provided by classical split-stream [11] and (*b*) reduce the compression ratio increase which typically occurs when a PPM-like algorithm compresses small amounts of data.

(*ii*) INSTRUCTION RESCHEDULING. We have developed three heuristics that reschedule instructions while preserving their dependencies with an aim to maximize the correct predictions of a PPM predictor. The three algorithms explore global vs. local heuristic objectives that reflect on solution quality vs. algorithm run-time respectively.

We implemented the compression algorithm for x86 binaries and tested its performance on a benchmark constituted of several large Microsoft applications. Binaries compressed using our algorithm were 18-24% smaller than files created using off-the-shelf PPMD, the best available compressor [12].

2. Related Work

We trace the related work along two directions: program binary compression and generic PPM compression variants.

Code compression has been addressed through dictionary based techniques that compute the dictionary using set-covering algorithms [21]. Lucco proposed a split-stream format for JIT interpretation of compressed code that halves program-size at the expense of a 27% decrease in program performance [22]. Random access decompression has been enabled by resetting the statistical model used for compression for each block of data [20].

Compression of programs' intermediate representation (IR) has resulted in impressive compression ratios. However, software publishers are hesitant to deliver code in IR format as its reverse engineering is a significantly simpler task [26]. Techniques have been developed for machine independent compressed code with adaptive compression of syntax trees [9] and a "wire-code" compression format that can be interpreted on-the-fly without decompression [7]. Fraser explored the bounds on the compression ratio for the IR of binaries by using machine learning techniques to automatically infer a decision tree that separates IR code into streams that compress better than the undifferentiated code [10]. Techniques for reducing code size include procedural abstraction and generalization of cross-jumps [11] and common instruction merger into super-operators [24]. A good survey of transformations for reduced code size is outlined in [6].

Architectures for minimized code size have been explored using instruction selection [21]. Operating systems and CPU support for the execution of compressed code on existing and modified architectures has been detailed in [19, 29].

The PPM compression paradigm introduced by Cleary and Witten [4], has set the bar on compression ratio performance that no other algorithm has been able to reach to date. Moffat's improvement, PPMC [23], set the benchmark for over a decade, until recently, when Howards' variant, PPMD [15], with improved computation of escape symbols was recognized as the best overall compression scheme [12].

3. PPM Overview

In this section, we detail the generic PPM engine as a backbone for our code compression algorithm presented in Sections 5 and 6. Lets denote the input data-stream to be compressed as $x \in \{\mathcal{A}\}^N$, where x is a sequence of N symbols from an alphabet $\mathcal{A}$. A *context of order* K is defined as a sequence of K consecutive symbols of x. For a current symbol x_i, its context of order K, $C(x_i, K)$, is the sequence of symbols $C(x_i, K) = \{x_{i-K}, \ldots, x_{i-1}\}$. We denote as $P_C(s)$, the probability that symbol $s \in \mathcal{A}$ follows context C.

While both compressing and decompressing, PPM builds a model of the input that aims at estimating the probability that a certain symbol occurs after a certain context. PPM encodes a symbol with the amount of information proportional to the probability that the symbol appears after its current context of certain order. The maximum referenced context order is constant. It has been shown that increasing the maximum order beyond five improves the compression ratio only negligibly [1]. A PPM model has entries for all limited-length contexts that have occurred in the processed data-stream. The model counts the symbols that occur after

IEEE COMPUTER SOCIETY

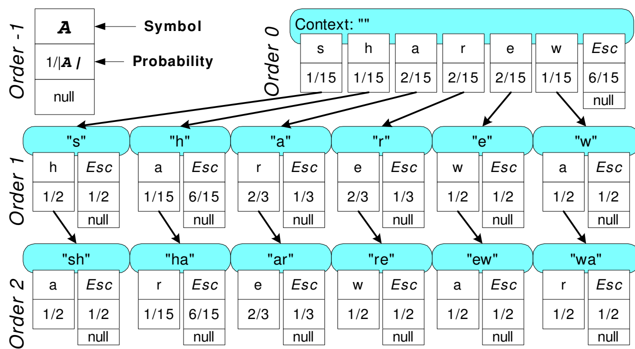

Figure 1. A PPM model with maximum order of two after processing the string "shareware".

each recorded context. The counts are used for calculation of symbol occurrence probabilities. For each context, there is a special *escape* symbol ε, used to resolve the case when a new symbol occurs after a recorded context.

While encoding a symbol, PPM initially considers its longest context. If the symbol is not found in the longest context, ε is emitted and the order of the current context is decremented. Since the decoder maintains the same model, the ε symbol signals to the decoder that it should switch to a shorter context. A special context of order -1 contains all symbols from the used alphabet. For this context, the probability of occurrence is uniform for all symbols $s \in \mathcal{A}$: $P_{-1}(s) = \frac{1}{|\mathcal{A}|}$. After the first occurrence of a symbol in the input data-stream, PPM switches back to the order-(-1) model and encodes the symbol accordingly.

An example of a PPM model with a maximal order of two after processing the string "*shareware*" is illustrated in Figure 1. The model records the occurrence of each symbol in its order-0 context. For each new symbol, the *escape* counter of the order-0 model is incremented. Central to a PPM implementation is the digital search tree or trie [28]. A new node (i.e. context) is added to the trie upon encountering a new symbol after a certain context. Each new node is initialized with two entries: one, for the new symbol that created this context, and another, for the *escape* symbol.

4. Why PPM Does Not Perform Well for Program Binaries?

To the best of our knowledge, PPM variants are by far the best generic compression techniques for program binaries [12]. However, while performing prediction by partial matching, a PPM algorithm does not take into account the fact that a program binary is being compressed. By definition, all PPM variants explore exclusively the localized correlations in sequential neighborhoods of the input data-stream (i.e. context-based correlation). In that light, there are several important questions that argue the efficacy of traditional PPM compression of code. Before we propose our solutions to these problems, in this section we elaborate the key questions related to the performance of PPM algorithms for program binaries.

Q1. For a given program binary format, which data-fields in the stream have high correlations? A PPM model is aware only of sequential local correlations between input symbols. Since most PPM compressors operate with 8-bit symbols (e.g. text characters), a PPM model can observe only the correlations that happen at byte boundaries. However, the x86 code[1] has a significantly richer structure than text. An x86 instruction has variable-length between 1 and 16 bytes. An instruction may contain the following fields: an optional *instruction prefix*, an *opcode*, a *displacement* (if required), and *immediate data* (if required). Details of the x86 instruction set can be found in [16]. Instruction format is shown in the following table:

Field	Prefix	Opcode	Displa-cement	Immediate data
Size (bytes)	0,1,2,3,4	1,2,3,4	0,1,2,4	0,1,2,4

An improvement to compression ratios can be made by making PPM aware of correlations that exist within the input. For example, in the case of XML compression, Cheney has demonstrated that general text compressors combined with the knowledge of the XML structure have superior performance [3]. We recognize two types of correlations that occur in a program binary: *horizontal* and *vertical*. Horizontal correlations occur among fields of the same instruction. Vertical correlations occur across all instructions among the data (fields) of the same type. We demonstrate how vertical and horizontal correlation affect the compression ratio of PPMD using two experiments. We extract the register and register addressing (RAR) fields from all instructions (experiment A) and all 4 byte displacements of CALL instructions (experiment B). Then, we compress separately the two resulting files (the extraction and the remainder). The results are presented in Table 4 where the third and the last column represents the relative change in compressed file size from experiments A and B respectively compared to the compressed file size in bytes of the original files. We observe that experiment A results in an *increase* in the compression ratio, clearly indicating that horizontal correlation of the RAR field is stronger than the vertical. On the other hand, the improved compression ratio in experiment B demonstrates slightly higher correlation among extracted displacements with respect to their correlation with the instructions where they are extracted from. Modeling these types of correlations is especially difficult for architectures with variable-length instructions. Any technique

[1] With no loss of generality, we restrict our work in this paper to x86 code. All techniques presented can be applied to different instructions sets in a straightforward manner.

0-7695-1913-X/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

that explores vertical correlations for such binaries needs to disassemble programs with respect to instruction length and fields.

While building its model, PPM observes and addresses only horizontal correlations among neighboring fields. In Section 5, we present SPLIT^2-STREAM, a technique that explores the trade-offs between considering both horizontal and vertical correlations while compressing code.

Table 1. Two experiments that demonstrate horizontal and vertical correlation among instruction fields.

Program name	Compressed [bytes]	Exp. A. %	Exp. B. %
Compiler CC1	416 218	+22.76	-2.79
MsAccess	1 919 666	+8.74	-3.07
WinwordXP	3 919 572	+7.34	-2.72
ExcelXP	3 807 300	+6.60	-2.82
PowerpntXP	2 140 822	+7.65	-3.96
Winword2000	3 225 381	+7.25	-2.71
VisualFoxPro6L	1 976 977	+7.32	-3.29
VisualFoxPro7	2 101 326	+7.32	-3.17

Q2. Which code transformations improve the compression ratio of program binaries? Prior to compression, a program binary can be transformed almost arbitrarily using a set of functionally isomorphic transformations. In general, all transformations that reduce the resources used by a program at little or no expense to code size, benefit compression as fewer symbols for resources (i.e. their encodings according to the instruction set) are referenced in the binary. An example of such a transformation is an optimization for register assignment and allocation - where the reduction of the number of used registers directly lowers program entropy. In this work, we assume that the compiler already performs a number of optimizations that improve code compression as a side effect: register allocation, reducing the number of computationally expensive instructions, etc. However, particularly for PPM, we have developed three heuristic algorithms which reschedules instructions in order to improve the prediction rate of the PPM model. The new schedule preserves the original dependencies among instructions. Details of the algorithms are presented in Section 6.

Q3. Are there global correlations in program binaries? In large applications, it is possible to explore self-similarities or even replication of procedures or small functional blocks for compression. Shorter multi-instruction constructs - superoperaters - have already been proven to improve the conciseness of a program [24]. Typically for large programs, code is commonly replicated mainly due to: inefficiency in programming as large groups of programmers write the source code and due to functional similarities between program constructs (e.g. loops, `case-switch` and `if-then-else` statements) and different procedures (e.g. sorting, data structure manipulation, finding a max/min). However, finding this type of similarity is a task beyond the scope of this paper. In this work, we explore only local correlations and trace our future work along identifying techniques for exploring global correlations within a program binary.

5. Split^2-Stream

The initial step in the compression process is to split the program binary into separate sub-streams that have strong auto-correlation. The procedure for isolating sub-streams needs to balance the *horizontal* and *vertical* correlation between instruction fields. Simultaneously, it needs to enforce that the isolation induces minimal overhead. We have developed an approximate disassembler which provides information to the decoder side about the length of each instruction and its corresponding fields as a binary gets decompressed. This information is sufficient to perform full decompression without injecting any additional data into compressed binary. Each of the sub-streams uses its own PPM model for prediction of upcoming symbols. The actual encoding/decoding for all sub-streams is performed into/from a single file.

As experiment A in Section 4 shows, extraction of some of sub-streams results in an increase of the file size. However, the extraction of the majority of sub-streams results in stronger auto-correlation which leads to better prediction of symbols during the encoding procedure, and, consequently to better compression ratios. As the auto-correlation grows when sub-streams contain a narrower set of data, the length of each of the sub-streams is reduced. Unfortunately, the compression ratio of a stream is getting better as the stream becomes longer. This behavior is typical for all compression engines including PPM. In order to illustrate such behavior, we have performed a simple experiment where we have extracted one MB of each binary from our benchmark suite and compressed it. We have also split the extracted code into 100 fragments of 10KB, and 10 fragments of 100KB. Then we have compressed each fragment separately. The average compression ratios obtained through this experiment are shown Table 2. Better compression ratios are obtained with longer fragments. This results in a fundamental trade-off: the higher the number of sub-streams, the stronger the auto-correlation which reflects in better compression ratios; at the same time, compression ratios are improved as the length of each sub-stream is increased.

To address all of these issues we have developed an algorithm for sub-stream isolation: SPLIT^2-STREAM. We

IEEE COMPUTER SOCIETY

Table 2. An example of PPMD compression ratios improvement with increase in size of input.

Program name	Average compression ratio		
	100×10K	10×100K	1×1M
Compiler CC1	0.4600	0.4086	N/A
MsAccess	0.4927	0.4292	0.3858
WinwordXP	0.5416	0.4862	0.4434
ExcelXP	0.5502	0.4928	0.4496
PowerpntXP	0.4587	0.3980	0.3650
Winword2000	0.5423	0.4875	0.4464
VisualFoxPro6L	0.4974	0.4502	0.4145
VisualFoxPro7	0.5022	0.4549	0.4221

demonstrate its features on x86 code. The key characteristic of x86 code is that instruction's prefix and opcode uniquely determine instruction's length as well as size of its fields. Thus, separating certain sub-streams that correspond to fields is an easy task that needs no additional information in the compressed file to perform the assembly. However, field separation must be fully governed by the exerted horizontal and vertical correlations. In general, since PPM is unable to capture vertical correlations in its model, we adopt the following separation policy: a sub-stream that corresponds to a certain field is separated from the program if its vertical correlation is stronger than its horizontal correlation.

Before we present the SPLIT^2-STREAM algorithm, we introduce certain definitions related to the *program stream* (*p-stream*) to be compressed. An *atomic-stream* (*a-stream*) of a p-stream is defined as a sequence of field values for all instructions in the p-stream for fields that correspond to a single operand and instruction type. An a-stream cannot be partitioned any further. A *molecular-stream* (*m-stream*) of a p-stream is defined as a sequence of field values for all instructions in the p-stream for fields that correspond to a set of operand and instruction types. M-streams can be partitioned into a-streams. An a-stream is by default an m-stream. A union of n (where $n > 1$) a- or m-streams is an m-stream that encompasses field values for all field and operation types that correspond to the argument m-streams. Figure 2 illustrates an example p-stream with two a-streams created by: isolation of 1-byte indexing constants (a-stream a) and 1 byte constants from arithmetic operations (a-stream b); and an m-stream created as $a \cup b$. Note that the order of the bytes in the union corresponds to the byte sequence in the containing p-stream.

We introduce two functions: $split(\cdot)$ and $merge(\cdot)$. Function $split(a, b)$ determines whether vertical correlation of an m-stream a is greater than its horizontal correlation

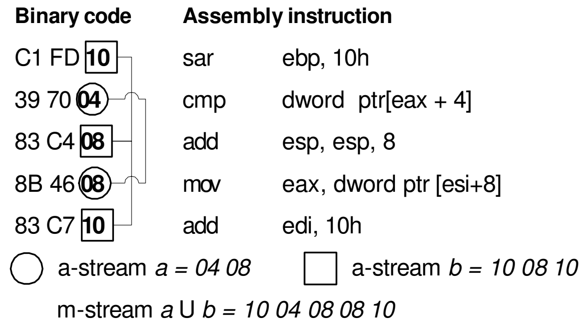

Figure 2. An example of a two *a-streams* that can be merged into a single *m-stream*.

with respect to its containing m-stream $s, a \subset s$. It is defined as:

$$\text{split}(a, s) = \ln\left(\frac{\varrho(s-a) + \varrho(a)}{\varrho(s)}\right), \quad (1)$$

where $s - a$ is an m-stream that represents an exclusion of a from s with respect to their containing p-stream, i.e. $((s - a)|s = (s - a) \cup a)$, and $\varrho(x)$ is a function that returns the size of the PPM-compressed argument stream x. Function $merge(a, b)$ on two m-streams a and b contained by the same p-stream, evaluates the effect of their merger on the resulting compression ratio. It is defined as:

$$\text{merge}(a, b) = \ln\left(\frac{\varrho(a \cup b)}{\varrho(a) + \varrho(b)}\right). \quad (2)$$

The input to the SPLIT^2-STREAM algorithm is a p-stream p and a set of a-streams A extracted from p such that:

$$(\forall a_i \in A)\ \text{split}(a_i, p) > 0. \quad (3)$$

Based on p and A, the algorithm initially creates the program's *core-stream* (*c-stream*) as: $c = p - \bigcup_{i=1}^{|A|} a_i$. The set of operand- and instruction-type fields that identify the set A is determined experimentally for a target instruction set. In general, set A is created by: first, considering the set of all a-streams extracted for each type of a constant and a control-flow displacement field that appears in instructions of a particular type, and then, filtering this set using Equation 3. An example that shows the two types of a-streams, filtered and preserved, is presented in Table 4.

The goal of the algorithm is to create a partitioning B of the starting set of streams $A' = \{A, c\}$ into M non-empty sets $b_i, i = 1 \ldots M$, such that: $\sum_{i=1}^{M} \varrho(b_i)$ is minimized. Function $\varrho(b_i)$ returns the file-size of the compressed union of all m-streams from A' in b_i. The number of all possible ways to partition A' into M streams, $S(M, |A| + 1)$, can be computed using the Stirling number of the second kind:

$$S(M, m) = \frac{1}{(m)!} \sum_{i=0}^{m-1} (-1)^i \begin{pmatrix} m \\ M \end{pmatrix} (m - i)^M \quad (4)$$

IEEE COMPUTER SOCIETY

$B = \{a_1, a_2, \ldots, a_{|A|}, c\}$
Repeat
Find $a, b \subset B$ s.t. $(\forall c, d \in B) d, e \neq a, b$ and $merge(a,b) \leq merge(d,e)$
Create $f = a \cup b$, remove a, b from B, add f to B
Until $|B| = M$

Figure 3. Pseudo-code of the Split^2-Stream procedure.

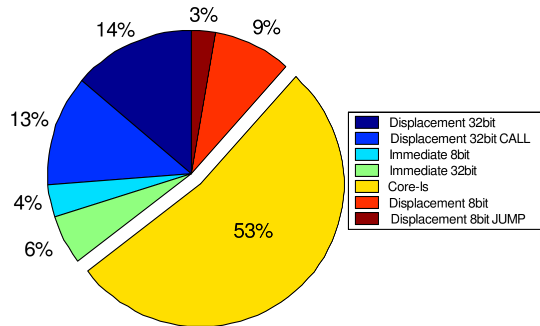

Figure 4. Relative size of the resulting sub-streams created using Split^2-Stream with $M = 7$ averaged over the benchmark suite.

where $m = |A| + 1$. For commonly considered $|A| = 25$ and $5 < M < 10$, exhaustive search is computationally too expensive, hence, we opt to use the following greedy heuristic.

Initially, we set $B = A'$. The heuristic iteratively performs the following step until $|B| = M$. It finds a pair of m-streams $a, b \subset B$ with minimal $merge(a,b)$. Then, we merge a and b into a single stream f and replace m-streams a, b with f in B. Pseudo-code for the SPLIT^2-STREAM algorithm is given in Figure 3.

Figure 4 illustrates the average relative size of m-streams in B for $M = 7$ generated over our benchmark of applications (see Table 4). Streams dominated with constants and control-flow displacements, are commonly compressed at low rates. However, the m-stream dominated with *core-I*s (where a core-I is an instruction with removed constants, and short and long control-flow displacements), is the performance bottleneck: it accounts for more than half the total data to be compressed and it is compressed at a rate significantly higher than the overall compression ratio. Hence, in the next section we present three algorithms that aim at reducing the compression ratio of the c-stream. Note that the partitioning of a-streams can be pre-computed and fixed at compression time. In the generated experimental results in Section 7, we use a fixed a-stream partitioning B presented in Figure 4.

6. Instruction Rescheduling

In this section, we present three algorithms for instruction rescheduling (IR) that improve the prediction rate of PPM. The essence of our approach is to alter the order of instructions such that dependencies among instructions are preserved, and PPM is predicting more accurately while using its model. To the best of our knowledge, code optimizations aimed at the improvement of instruction correlations such that modeling of a compression algorithm is enhanced, have not been developed to date. We have restricted our attention to IR within a basic block to attenuate the effect of this post-compilation optimization on execution speed of super-scalar processors.

IR is performed after the binary is split into m-streams using SPLIT^2-STREAM. IR operates only on the c-stream. Changes in the order of instructions in the c-stream must be propagated to other m-streams for structure consistency. There are three algorithms that we propose. All three algorithms first identify the set of unique core-Is along with their frequencies[2]. This set represents the symbol alphabet $\mathcal{A}$ for the input stream to PPM. We denote the cardinality of $\mathcal{A}$ with L. The input to PPM is a vector of N symbols $x \in \{\mathcal{A}\}^N$. We define a *dependency set* $D(x)$ of x as a set of instruction pairs, where a pair $d(i,j) = \{x_i, x_j\} \in D(x)$ denotes that instruction x_i can be rescheduled with preserved dependencies such that x_i immediately precedes instruction x_j. The goal of IR is to create a permutation $\pi(x)$ such that:

- all connected instruction pairs $x_a x_b \in \pi(x)$ are found in $D(x)$, i.e. $d(a,b) \in D(x)$, and
- a PPM prediction engine of order K emits minimal number of *escape* symbols while compressing $\pi(x)$ according to its model presented in Section 3.

Algorithm A1.
This algorithm aims at building a solution that improves PPM's context matching at the global level by maximizing the number of most frequent symbol-to-symbol occurrences throughout the entire binary. $A1$ performs iteratively the following two steps. In the first step, the algorithm computes the connectivity matrix of the target c-stream. A *connectivity matrix* of a stream x denoted as $\mathcal{M} = \{\mathcal{I}\}^{L \times L}$ is a matrix of non-negative integer numbers where each element $m(i,j) \in \mathcal{M}$ equals the count of all pairs $d(k,l) \in D(x)$ such that $x_k = a_i$ and $x_l = a_j$, where a_i and a_j are possibly equal symbol values from $\mathcal{A}$. An example of the format of this matrix is illustrated in Figure 5.

In the next step, $A1$ finds the largest element $m(i,j)$ of $\mathcal{M}$. Next, $A1$ reorders instructions such that the count of all concatenated symbols $x_k x_{k+1}$ with values $x_k = a_i$ and $x_{k+1} = a_j$, is maximized throughout the entire stream.

[2] With no loss of generality, we assume that the c-stream consists of core-Is.

0-7695-1913-X/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

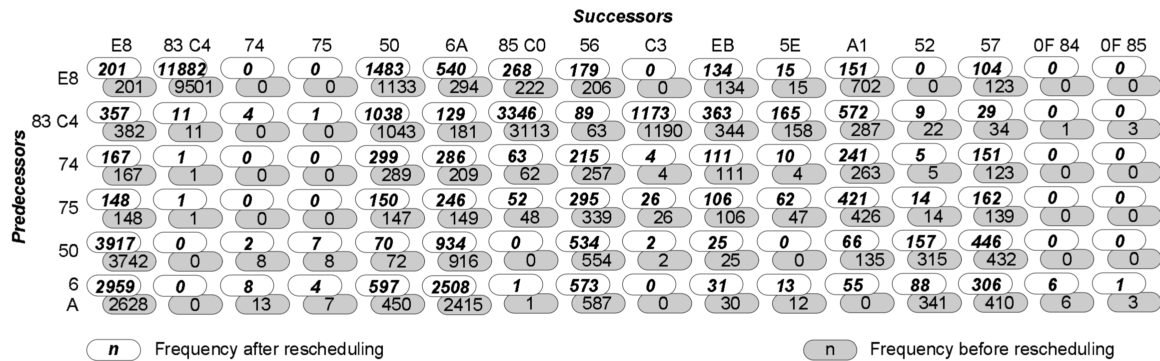

Figure 5. An example of the connectivity matrix $\mathcal{M}$ for the top 6×15 most frequent unique core-Is in the benchmark program CC1. Shaded and white cells specify the frequency of occurrence of a $x_i x_j$ sequence before and after applying the algorithm $A1$.

Then, element $m(i, j)$ in $\mathcal{M}$ is permanently set to 0. All symbols throughout the reordered stream, that are concatenated as $x_k x_{k+1}, x_k = a_i, x_{k+1} = a_j$, are tagged such that subsequent iterations of $A1$ cannot insert any other core-I between them. In the subsequent iteration, the tags are considered when recomputing $\mathcal{M}$ by updating the dependency set $D(x)$ correspondingly. The two steps of $A1$ are repeated until all elements of $\mathcal{M}$ are smaller than 2. An example how the number of sequential occurrences of the most frequent core-Is in CC1 changes after several steps of $A1$, is illustrated in Figure 5.

Although $A1$ restricts IR within a basic block, it explores the possible concatenation of symbols $x_k x_{k+1}$ if they are found in consecutive code basic blocks, and there are no dependencies preventing rescheduling of x_k at the end of its basic block and x_{k+1} at the beginning of the consecutive basic block. The schedule of two instructions is then fixed as in the case when both instructions are found inside of a single basic block. An example of IR over two consecutive code basic blocks is shown in Figure 6.

$A1$ increases globally the probability that certain symbols appear after a given symbol. Actually, it is straightforward to prove that $A1$ is optimal by construction for a PPM model of order 1. For higher orders, $A1$ is a greedy heuristic that performs well. The heuristic goal that $A1$ aims to achieve for higher order PPM models is sequencing of core-Is into common contexts. The hope is that if common contexts exist in the program binary, $A1$ enforces their appearance.

Algorithm A2.

Algorithm $A1$ has a strong deficiency; it is rather slow and memory-consuming for large binaries. The complexity of algorithm $A1$ is $O(L^2 N)$, which may be unacceptable for certain applications. To address this issue, we have developed algorithm $A2$ that aims at finding local core-I schedules that improve the prediction of the PPM model according to its current state. The advantage of $A2$ over $A1$ is that it is significantly faster and that it adapts the instruction schedule to fit the current PPM model regardless of PPM's order. The disadvantage is that it fails to recognize the global correlation of instruction sequences.

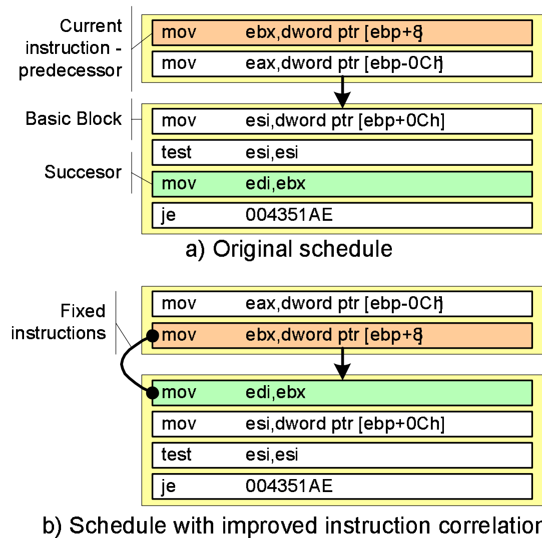

Figure 6. An example of instruction rescheduling when x_k (predecessor) and x_{k+1} (successor) are found in consecutive code basic blocks.

$A2$ reschedules non-control-flow core-Is only within their basic block. For each basic block, $A2$ reschedules the core-Is exhaustively such that PPM emits the minimal number of *escape* symbols starting from its current model state. An example of how core-Is can be rescheduled within a basic block is illustrated in Figure 7. Since the number of different schedules in a basic block may potentially grow exponentially with respect to the number of instructions, we partition large basic blocks into mini-blocks of limited car-

IEEE COMPUTER SOCIETY

	Instruction	Possible position						
		1	2	3	4	5	6	7
1	mov esi, [eax+8]	⊗	×	×	∗			
2	mov ecx, [esp+1Ch]	×	⊗	×	×	∗		
3	mov edx, [eax+4]	×	×	⊗	×	∗		
4	or esi, 100h		×	×	⊗	×		
5	cmp ecx, edx			∗	×	⊗	×	
6	mov [eax+8], esi			∗	∗	×	⊗	
7	je LOOP							⊗

Figure 7. An example of IR within a basic block. ⊗ - current position; × - possible position; ∗ - possible position if at least one other core-I is moved as well.

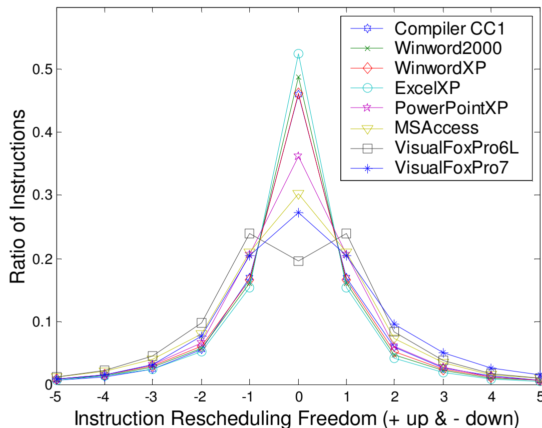

Figure 8. Ratio of total instructions that can be moved up (positive x-axis) and down in a basic block for our benchmark suite.

dinality (typically 10-15 core-Is). Although this step puts an upper bound on algorithm complexity, it actually has little effect on the final results as instructions can rarely be moved significantly up and down in x86 code because of a small register file. IR freedom for our benchmark suite of programs is presented in Figure 8. In general, the results that $A2$ obtains are comparative to the results of $A1$ within 0.5% difference in the final compression ratio. Hence, the experimental results that we report in Section 7, are obtained using $A2$.

Algorithm A3.

Algorithm $A3$ is a compromise between algorithms $A1$ and $A2$ in terms of speed and the compression ratio. While it is significantly faster then $A1$, it achieves better compression ratios than $A2$. $A3$ aims at building a solution that improves PPM's context matching on both global and local levels by sorting instructions within a basic block. Instructions are sorted by their binary codes of the corresponding core-I. Starting from the beginning of a basic block, PPM is more likely to encounter core-Is that have lower binary code. As it progresses through the code, the likelihood of instructions

	Binary Code	Assembly	
1	**8B 50** 04	mov	edx, [eax+4]
2	**8B 4C 24** 1C	mov	ecx, [esp+1Ch]
3	**8B F3**	mov	esi, ebx
4	**3B CA**	cmp	ecx, edx
5	**89 70** 08	mov	[eax+8], esi
7	**74** 14	je	LOOP

a) Schedule before ordering

1	Binary Code	Assembly	
2	**8B 4C 24** 1C	mov	ecx, [esp+1Ch]
3	**8B 50** 04	mov	edx, [eax+4]
4	**3B CA**	cmp	ecx, edx
5	**8B F3**	mov	esi, ebx
6	**89 70** 08	mov	[eax+8], esi
7	**74** 14	je	LOOP

b) Schedule after ordering

Figure 9. An example of IR within a basic block. Instructions are sorted by their core-Is (bold) under the constraint of dependencies among instructions.

with higher binary codes increases. Sorting also helps code structure on the global level. Since a particular order of instructions is now fixed, whenever a pair of instructions is encountered, they will appear in the same order throughout the binary. Consider the example shown in Figure 9. The first two instructions do not have dependencies. The sorting of core-Is will switch their positions in the input stream. Since this holds for the whole binary, in the PPM model there will be no **8B 4C 24** in the context of **8B 50**. This means that the instructions that do appear in the context of **8B 50** will be predicted with higher probability, and consequently with fewer bits. Also, dependencies from $D(x)$ will make instructions that are correlated through those dependencies next to each other. This further increase correlation among instructions which enables PPM to encode the stream more efficiently. An illustration of such cases are the rescheduling of **8B F3** next to **89 70** 08, and **3B CA** next to **8B 50** 04.

The sorting procedure slightly differs from standard sorting procedures. $A3$ needs several passes through the basic block to find an optimal ascending order under constraint of instructions' dependencies. This approach proved to be much faster than the calculation of all possible schedules which can become highly computationally expensive when the number of instructions within a basic block exceeds 10. After each pass, the algorithm inspects the basic block in order to determine wether an instruction swap enabled a schedule previously not feasible. The program terminates because of the monotonicity of the sorting procedure.

IEEE COMPUTER SOCIETY

Table 3. Compression ratio comparison for a benchmark of large applications: bzip2 vs. PPMD vs. our technology PPMexe. File-size reported in bytes. Our technology used the $A2$ algorithm, $M = 7$ m-streams, PPM model of order $K = 4$ with $L_1 = L_2 = 256$ symbols for each alphabet. C. Ratio - compression ratio.

Program Name	Uncompressed File Size	bzip2 C. Ratio	PPMD C. Ratio	Split^2-stream C. Ratio	PPMexe C. Ratio	Improvement PPMD [%]	Improvement bzip2 [%]
Compiler CC1	872588	0.5126	0.4770	0.4173	0.3917	17.9	23.6
Winword2000	6137358	0.5669	0.5255	0.4522	0.4273	18.7	24.6
WinwordXP	7535800	0.5614	0.5201	0.4647	0.4206	19.1	25.1
ExcelXP	7303411	0.5624	0.5213	0.4630	0.4215	19.1	25.0
PowerPointXP	4449093	0.5212	0.4812	0.3941	0.3660	23.9	29.8
MSAccess	3942060	0.5275	0.4870	0.4018	0.3887	20.8	26.3
VisualFoxPro6L	3769781	0.5662	0.5244	0.4349	0.4061	22.6	28.3
VisualFoxPro7	3941408	0.5752	0.5334	0.4395	0.4167	21.9	27.6

7. Experimental Results

We tested the effectiveness of our code compression technology on a representative benchmark of large applications that consisted of Microsoft's OfficeXP suite, Microsoft Word 2000, and a C compiler. It is important to stress that we have removed from each program the gluing data blocks that exist in x86 binaries. These blocks can be compressed with ratios incomparably low with respect to the remainder of the data in the executable (less than 0.25). The results are presented in Table 4. Size of uncompressed programs excludes the data blocks. We have used the powerful Vulcan technology to manipulate x86 binaries [25].

We have compared our technology, named *PPMexe* in the table, with two most competitive compression technologies today: bzip2 [2], which uses the Burrows-Wheeler transform and PPMD [15]. PPMexe has demonstrated superb performance with respect to bzip2 and PPMD, outperforming them on the average for 26.3% and 20.4% respectively. Our decompression software reported 600Kb/s throughput on a 300MHz Pentium III, i.e. several times larger than the bandwidth of a generous DSL link between an SD server and a client. The decompression speed of our software is in the same range as PPMD. The overhead associated with reassembling of instructions is compensated with better PPM modeling which is a consequence of the techniques we have developed and described throughout this paper. The rule of the thumb for the PPM algorithm is that the better the compression rate of a file is (i.e. the better its PPM model is), the faster is its decompression speed. PPM as a compression paradigm is inherently inferior with respect to its compression/decompression speed to bzip2. Hence, bzip2 yielded higher decompression throughput ranging around 1100Kb/s on the given benchmark suite.

Impact of SPLIT^2-STREAM and INSTRUCTION RESCHEDULING to the compression ratio is not orthogonal. Rescheduling of the instructions without breaking them into streams has a limited effect to the compression ratio because horizontal and vertical correlations are opaque to the compression algorithm. However, when the instructions are split, horizontal and vertical correlations of streams are exposed. Even on the architecture with typically small basic block size (such as x86), instruction rescheduling improves exposed correlations, enables better modeling for the compressor, and impacts the compression ratio significantly. The sole contribution to the improvement of the compression ratio of SPLIT^2-STREAM is shown in Table 7. The effect of INSTRUCTION RESCHEDULING without SPLIT^2-STREAM is not commeasurable with its full effect when both techniques are applied, thus, it is not shown separately. SPLIT^2-STREAM and INSTRUCTION RESCHEDULING contributed to the overall improvement approximately with a 2:1 ratio.

We now discuss effects of IR on the execution speed. Advanced architectures provide an environment for IR that affects the execution speed only negligibly. For example, Pentium III architecture utilizes an advanced dataflow analysis which creates an optimized, reordered schedule of instructions analyzing data dependencies between instructions [17]. Similarly, Pentium 4 Advance Dynamic Execution engine provides a view of 126 instructions simultaneously which enables deep out-of-order speculative execution [18]. It follows that IR for code compression minimally affects the execution speed of program binaries. In our IR algorithms presented in Section 6, we have restricted IR within basic blocks which further reduces the impact of IR on the execution speed. We have conducted a set of execution speed experiments on our benchmark suite. The experiments showed that execution speeds of binaries optimized for code compression are within 1% of execution speeds of binaries with the original schedule of instructions.

IR for code compression introduces a potential slowdown in execution speed on architectures that do not support out-of-order execution. This potential slowdown is a consequence of *structural* and *data hazards*([13]) that may arise when instructions are rescheduled such that resources conflicts and data dependencies are exposed in a given architecture. Furthermore, IR can cause a functionally incorrect execution on certain architectures. In such cases, IR

IEEE COMPUTER SOCIETY

for code compression must include additional constraints which reflect the limitations of the underlying architecture.

8. Conclusion

Systems such as software delivery platforms, embedded systems, and mobile code have imposed strong requirements for high compression rates of binaries. In this work, we have explored compression algorithms that focus exclusively on program binaries. We have adopted as a fundament of our algorithm, prediction by partial matching (PPM) - a compression paradigm that has demonstrated superior performance with respect to other compression techniques: Lempel-Ziv, Burrows-Wheeler transform, and Huffman coding. In this paper, we have proposed a compression mechanism that uses pre-processing steps to PPM that significantly improve the achieved compression ratios. The pre-processing steps include: heuristic partitioning of a binary into highly correlated sub-streams and re-scheduling of instructions to improve prediction rates.

We have demonstrated the effectiveness of the developed techniques by building a compression codec for x86 binaries. The tool has been tested on a benchmark constituted of several large Microsoft applications. Binaries compressed using our technology, were 18-24% smaller than files created using off-the-shelf PPMD, the best available compression tool. While obtaining superior compression ratios, our code optimization algorithms for code compression have negligibly affected the execution speed of program binaries.

References

[1] S. Bunton. Semantically motivated improvements for ppm variants. *The Computer Journal*, 40(2/3), 1997.

[2] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. *Technical report - Digital Equipment Corporation*, 1994.

[3] J. Cheney. Compressing xml with multiplexed hierarchical models. *Data Compression Conference*, pages 163–172, 2001.

[4] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transacitons on Communications*, 32(4):396–402, 1984.

[5] J. Debardelaben, V. Madisetti, and A. Gadient. Incorporating cost modeling into embedded-system design. *IEEE Design & Test of Computers*, 14(3), 1997.

[6] S. Debray, W. Evans, and R. Muth. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.

[7] J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting. Code compression. *Programming Languages Design and Implementation*, pages 358–365, June 1997.

[8] C. Farley. Microsoft .net vs. j2ee: How do they stack up? *Available on-line at http://java.oreilly.com/news/*, 2001.

[9] M. Franz and T. Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, 1997.

[10] C. Fraser. Automatic inference of models for statistical code compression. *Programming Languages Design and Implementation*, pages 242–246, 1999.

[11] C. Fraser, E. Myers, and A. Wendt. Analyzing and compressing assembly code. *ACM SIGPLAN Symposium on Compiler Construction*, 19:117–121, 1984.

[12] J. Gilchrist. The archive compression test. *Available on-line at http://compression.ca*, 2000.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, San Francisco, Ca., second edition, 1995.

[14] I. Hong, D. Kirovski, and M. Potkonjak. Potential-driven statistical ordering of transformations. *Design Automation Conference*, pages 347–352, 1997.

[15] P. Howard and J. Vitter. Design and analysis of fast text compression based on quasi-arithmetic coding. *Data Compression Conference*, pages 98–107, 1993.

[16] Intel architecture software developer's manual, volume 2: Instruction set reference manual. *http://developer.intel.com/design/processor*.

[17] http://www.intel.com/design/pentiumiii.

[18] http://www.intel.com/design/pentium4.

[19] D. Kirovski, J. Kin, and W. H. Mangione-Smith. Procedure based program compression. *Int'l Symp. on Microarchitecture*, 1997.

[20] H. Lekatsas and W. Wolf. Random access decompression using binary arithmetic coding. *Data Compression Conference*, pages 306–315, 1999.

[21] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Instruction selection using binate covering for code size optimization. *Int'l Conf. on Computer-Aided Design*, pages 393–399, 1995.

[22] S. Lucco. Split-stream dictionary program compression. *Programming Languages Design and Implementation*, pages 27–34, 2000.

[23] A. Moffat. Implementing the ppm data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, 1990.

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

[25] A. Srivastava and H. Vo. Vulcan: Binary transformation in distributed environment. *Micorosft Research Technical Report*, MSR-TR-2001-50, Apr. 2001.

[26] T. Systa, P. Yu, and H. Muller. Shimba-an environment for reverse engineering java software systems. *Software Practice and Experience*, 31(4):371–394, 2001.

[27] T. Truman, T. Pering, R. Doering, and R. Brodersen. The infopad multimedia terminal: A portable device for wireless information access. *IEEE Trans. on Computers*, 47(10):1073–1087, 1998.

[28] I. Witten, A. Moffat, and T. Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, San Francisco, Ca., 1999.

[29] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. *Proc. Int'l Symp. on Microarchitecture*, 1992.

0-7695-1913-X/03 $17.00 © 2003 **IEEE**

IEEE COMPUTER SOCIETY