PPMexe: Program Compression

MILENKO DRINIĆ, DARKO KIROVSKI, and HOI VO
Microsoft Research

With the emergence of software delivery platforms, code compression has become an important system component that strongly affects performance. This paper presents PPMexe, a compression mechanism for program binaries that analyzes their syntax and semantics to achieve superior compression ratios. We use the generic paradigm of prediction by partial matching (PPM) as the foundation of our compression codec. PPMexe combines PPM with two pre-processing steps: (*i*) instruction rescheduling to improve prediction rates and (*ii*) heuristic partitioning of a program binary into streams with high auto-correlation. We improve the traditional PPM algorithm by (*iii*) using an additional alphabet of frequent variable-length super-symbols extracted from the input stream of fixed-length symbols. In addition, PPMexe features (*iv*) a low-overhead mechanism that enables decompression starting from an arbitrary instruction of the executable, a property pivotal for run-time software delivery. We implemented PPMexe for x86 binaries and tested it on several large applications. Binaries compressed using PPMexe were 18-24% smaller than files created using off-the-shelf PPMD, one of the best available compressors.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Software Distribution; D.3.4 [**Programming Languages**]: Optimization for Compression; E.4 [**Coding and Information Theory**]: Data Compaction and Compression

General Terms: Algorithms, Performance

Additional Key Words and Phrases: instruction scheduling, prediction by partial matching, random access compression, software compression, software distribution.

1. INTRODUCTION

Compression ratio[1] of program binaries is a parameter that directly impacts several applications with restricted bandwidth and storage resources. Relatively high cost of improving the bandwidth of the "last-mile"[2] of the global net, points to compression of program binaries as effective means to improving performance of software delivery platforms [Mohney 2003]. These platforms depend significantly on compression of binaries for two reasons. First, client performance is largely governed by the delay and bandwidth of the communication channel that links to the delivery server. Second and more important, the workload on the server is highly

[1]We define compression ratio as a ratio of the compressed file size and the original file size.
[2]The "last mile" typically refers to the link that connects households to the global communication framework.

impacted with the number of service requests. Both performance parameters are improved with decreased compression ratio of the communicated data: the program binaries. Efficient code compression also benefits applications with restricted memory resources. Wireless systems that extensively use *mobile code* have energy savings directly proportional to the decrease of the compression ratio as power consumption due to device's transceiver activity, commonly dominates the energy bill of wireless devices [Truman et al. 1998].

In this paper we present compression mechanisms for program binaries that analyze their syntax and semantics to achieve superior compression ratios. Although the techniques presented in this paper relate to the x86 instruction set, they can be applied to any other instruction set (Java byte code, MSIL, MIPS, ARM, etc.). Nevertheless, by targeting the x86 instruction set, we aim at compressing binaries for the most prolific computing platform today. While software delivery platforms typically provide programming environments centered around virtual machines, most large applications are still distributed in native code for:

—**performance** - instruction interpretation introduces non-trivial execution overhead [Romer et al. 1996], and

—**protection of intellectual property** - most virtual machines have been created without the constraint that byte-code disassembly should be made a difficult task [Baker and Manber 1998] - historically, the x86 instruction set has been attributed with difficult disassembly.

In this work we focus our attention to the following questions that are crucial for creating an efficient compression scheme. We address these questions in more detail in Section 2.

Q1. *For a given binary format, which data fields have high correlations?* Program binaries are heterogeneous data collections. A number of data streams (e.g., instruction opcodes, displacements, immediate data fields, and data) is interlaced in a complex manner. Capturing the correlations among these streams can significantly improve compression.

Q2. *Which code transformations improve the compression ratio of program binaries?* Program binaries can have a large number of functionally equivalent representations. We investigate a set of code transformations whose target is to enable better compression rates.

Q3. *How can a basic PPM engine be modified to benefit from the knowledge that it is compressing a particular program binary format?* Modifying a compression scheme toward a specific data set can further enhance quality of compression.

We chose an effective general purpose compression scheme, prediction by partial matching (PPM) [Cleary and Witten 1984], as a fundament of our algorithm. We combined this powerful compression paradigm with several pre-processing steps in order to address the posed questions:

(*i*) Instruction Rescheduling. We 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 complexity.

(*ii*) SPLIT^2-STREAM. We have developed an algorithm that initially partitions 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 [Fraser et al. 1984] and (*b*) reduce the increase in compression ratio which typically occurs when a PPM-like algorithm compresses small amounts of data.

In addition, we have improved the traditional PPM algorithm using a:

(*iii*) DUAL ALPHABET PPM. Our version of PPM operates with two alphabets: (*a*) the original alphabet of fixed-length (8-bit) input symbols and (*b*) an alphabet of common variable-length super-symbols (multi-byte). The latter alphabet is extracted from the binary and represents a list of most frequent unique instructions in the program.

Minimized compression ratio is not the only requirement for a code compression technology aiming at efficient software delivery. Namely, the computation system may decompress a program in two ways: entirely before execution and partially, caching the most frequently used program pages at run-time [Kirovski et al. 1997]. To address the latter case, we introduce three different PPM variants for:

(*iv*) RANDOM ACCESS DECOMPRESSION. The variants are: model copying, model undoing, and model stopping. They all enable decompression starting from an arbitrary instruction of the executable, a feature pivotal for run-time software delivery. However, they respond differently to the decompression speed vs. compression ratio trade-off.

We implemented the proposed compression algorithm for x86 binaries and tested its performance on a benchmark that encompassed several large applications extracted from Microsoft's suite of programs. Binaries compressed using our algorithm (*i-iii*), were 18-24% smaller than files created using off-the-shelf PPMD, one of the best available compressors [Gilchrist 2000; Witten et al. 1999]. Using techniques described in (*iv*), we were able to facilitate random access decompression at a negligible increase in file-size.

1.1 Software Delivery Objectives

The developed compression algorithm targets software delivery systems. In a typical software delivery environment, a client is linked to a server via a relatively low-bandwidth Internet connection (e.g., modem or DSL). To run a program, the client originally downloads a small subset of all the functionalities offered in the application. For example, the initial code should be sufficient to run the kernel of the application. Then, as the client invokes functions not yet downloaded, service requests are sent to the software delivery server. A service request is typically an address offset within the binary. The server replies with a block of data containing the target address. The distributed blocks may represent procedures or groups of procedures (i.e., funclets[3]). Upon reception, the client stores the funclet and executes the invoked procedure. Such a system enables richer software pricing and

[3]In the remainder of the paper, we assume that funclets of variable size are distributed via the software delivery link.

metering models, greatly facilitates software updates and bug fixes, and improves software integrity and availability. Performance evaluation of such a framework is beyond the scope of this paper. Thus, we assume that the amount of transferred data over the software delivery link significantly limits both server and client performance.

We review the typical requirements for a compression algorithm through the prism of software delivery:

A. *Encoding speed.* Commonly, there are no time-lines for program compression in a software delivery framework. An exception is the case of individualized software. An example of a such type of software is the Microsoft Windows Media Player whose individualized Digital Rights Management component is uniquely obfuscated for each instance of the binary. In such an event, the software distributor needs to either use a fast compression algorithm or maintain a reserve of compressed unique program copies or use a technology that creates unique compressed program copies from multiple versions of compressed funclets.

B. *Decoding speed.* This is one of the ultimate requirements for a code compression algorithm that targets software delivery. Since decompression can be pipelined with the actual download of the funclet, minimum decompression throughput equals the highest downlink bandwidth a software delivery system may encounter. We have set a target decompression speed of 1MB/s on a 2.8GHz Pentium IV, performance achievable by an off-the-shelf PPMD [Howard and Vitter 1993].

C. *Compression ratio.* This parameter is the key to improving system performance. For different compression paradigms, it ranges within the following neighborhoods: 0.75 for Huffman codes [Huffman 1952], 0.65 for LZ techniques [Ziv and Lempel 1978], 0.55 for BWT-based compressors [Burrows and Wheeler 1994], and 0.5 for PPM variants [Howard and Vitter 1993]. As the desired decoding speed can be easily reached on many computing platforms for most PPM algorithms, this parameter may be traded off only with:

D. *Decompressor memory size.* Although irrelevant for PC platforms, memory size plays an important role for embedded systems. There are two main memory consumers for PPM-based compressors: decompression software and the statistical model built during compression. It is important to identify the trade-off between memory consumption of a given PPM-based algorithm and its compression ratio.

2. WHY GENERAL PURPOSE COMPRESSION SCHEMES DO NOT PERFORM WELL FOR PROGRAM BINARIES?

The most successful general purpose compression schemes are adaptive. They build a model of processed data, predict unprocessed data, and efficiently encode differences between actual data and predictions [Witten et al. 1999]. When compressing program binaries, these algorithms do not take into account the type of data being processed. They are unable to capture the intricacies of the heterogeneous structure of program binaries and complex correlations of different data streams within binaries. In this section, we discuss several key observations used as guidelines while creating an effective compression scheme for program binaries. We focus our attention to PPM and its variants, which yields the best compression ratio [Gilchrist

2000]. We present more details about PPM paradigm in Appendix A. 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 among 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 sequentially at byte boundaries. However, the x86 code[4] has a significantly richer structure than text. Instructions in the x86 instruction set have variable length from 1 to 16 bytes. An instruction may contain the following fields: an optional *instruction prefix*, an *opcode*, a *displacement* (if required), and *immediate data* (if required). Certain opcodes include bits that specify the addressing mode and/or the scale index base. Details of the x86 instruction set can be found in [Intel Corp. 1999b].

Program name	Compressed [bytes]	Experiment A %	Experiment 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

Table I. Two experiments that demonstrate horizontal and vertical correlation among instruction fields. Column 2 presents program size after compression using PPMD and columns 3 and 4 show the relative change in file size after compressing separately register and register addressing fields and the program remainder (experiment A) and compressing separately the displacement of CALL instructions and the program remainder (experiment B).

An improvement to the compression ratio can be made by making PPM aware of correlations that exist within the input. 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 among the same field type and across all instructions. 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) and compress separately the two resulting files (the extraction and the remainder). Although both strategies A and B seem viable for compression improvement, one

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

actually results in decreased compression performance. From the results presented in Table I, we observe that A results in an *increase* in the compression ratio indicating that horizontal correlation of the RAR field is stronger than the vertical. Similarly, the improved compression ratio in B demonstrates slightly higher vertical correlation of extracted displacements. Modeling such correlations is especially difficult for architectures with variable-length instructions. Any technique that explores vertical correlations for such binaries must disassemble programs with respect to instruction length and fields.

While building its model, PPM observes and addresses only horizontal correlations among neighboring fields. In Subsection 3.1, we present SPLIT^2-STREAM, a technique that explores the trade-offs between considering both horizontal and vertical correlations while compressing code.

Q2. Which code transformations improve the compression ratio of program binaries? Prior to compression, a program binary can be transformed almost arbitrarily[5] using a functionally isomorphic set of 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 algorithms that reschedule 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 Subsection 3.2.

```
Assembly:     mov byte ptr[ebp-1Ch],0CCh
Binary code: |C6 45| E4 CC ← Immediate data
                ↑     ↑
             Core-I  Displacement
```

Figure 1. An example of a x86 instruction and its components: its *core-I*, an 8-bit displacement, and 8-bit immediate data.

Q3. How can a basic PPM engine be modified to benefit from the knowledge that it is compressing a particular program binary format? An all-purpose PPM implementation commonly processes an input stream using fixed-size symbols (typically 8-bit). However, correlations in a program happen at the field level. Fields may be longer or shorter than 8-bits. Consider a stream of core instructions (i.e., core-I), where a *core-I* is an instruction with removed constants (immediate operands) and short and long control-flow displacements (fields

[5]Applications rarely limit the usage of transformations, e.g., code obfuscation may limit certain transformations to reduce exposure of program functionality.

of instructions used to calculate the next value of the program counter). An example of a core-I is shown in Figure 1. Core-Is are usually between 1 and 4 bytes long in the x86 instruction set.

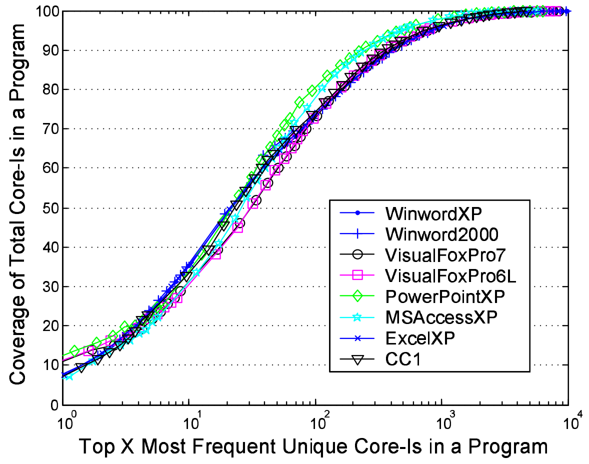


Figure 2. Percentage of all Core-Is of a program (y-axis) covered with a set of x-axis most frequent unique core-Is in the program.

We make the following observation for x86 core-Is in relatively large programs[6]: *regardless of the number of unique core-Is in a program, more than 85% of all the core-Is of the program belong to a set of top 256 most frequent unique core-Is.* Figure 2 illustrates how the coverage (y-axis) of program's core-Is increases as the most frequent unique core-Is (x-axis) are considered. For a suite of functionally different applications such as Microsoft Office and a compiler, the observation holds as true. Note that different versions of applications (Winword2000 and WinwordXP, VisualFoxPro6L and VisualFoxPro7) as well as CC1 are compiled with different compilers. This validates our observation across different compilers on x86 platform.

We explore this observation in the following way. We use two alphabets: $\mathcal{B}$ - with variable-length symbols that represent the $L_1 = |\mathcal{B}|$ (typically L_1=256) most frequent unique core-Is, and $\mathcal{A}$ - which is the standard alphabet of all $L_2 = 256$ 8-bit symbols. By giving higher preference to the $\mathcal{B}$ alphabet when processing the input stream, we reduce the number of predictions the PPM model makes and improve the modeling of correlations among data-fields.

Consider the example shown in Figure 3. An identical byte (**39** Hex) appears in two different unique core-Is. Although there is a significant semantic difference between the two occurrences, traditional PPM treats them as equivalent events while updating its statistical model. However, in the dual alphabet PPM model, under the assumption that symbols 39 46 Hex and 89 39 Hex belong to $\mathcal{B}$, two separate corresponding entries in the PPM model are updated yielding a qualitatively more

[6]Programs larger than 10^4 instructions.

accurate model. Details on how the two alphabets are used to create a PPM model are given in Subsection 3.3.

Binary code	**Assembly code**
$\boxed{\mathbf{39}\ 46}$ 04	cmp dword ptr [esi+4], eax
$\boxed{89\ \mathbf{39}}$	mov dword ptr [ecx], edi

Figure 3. An example of two instructions with different unique core-Is (framed) that have been selected as symbols in $\mathcal{B}$.

3. CODE COMPRESSION USING PPM

In this section, we describe the technical details behind our algorithm for code compression. The algorithm has two pre-processing steps: Split^2-Stream and instruction rescheduling; followed by a PPM-like compression process.

3.1 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 *balances the horizontal and vertical correlation* among instruction fields. Each sub-stream is compressed individually. The length of a sub-stream is governed by the number of sub-streams as funclet size has little variance in a software delivery system. This results in a fundamental trade-off: the higher the number of sub-streams versus the compression ratio of PPMD. The higher the number of sub-streams results in the stronger the auto-correlation of sub-streams. U nfortunately, the compression ratio for PPMD increases as the amount of compressed data decreases as presented in this example in Table II.

Average Compr-ession Ratio	Program Name							
	Compiler CC1	Win-Word 2000	Win-Word XP	Excel XP	Power-Point XP	MS Access	Visual Fox-Pro6L	Visual Fox-Pro7
10K	0.4600	0.5423	0.5416	0.5502	0.4587	0.4927	0.4974	0.5022
100K	0.4086	0.4875	0.4862	0.4928	0.3980	0.4292	0.4502	0.4549
1M	N/A	0.4464	0.4434	0.4496	0.3650	0.3858	0.4145	0.4221

Table II. An example of PPMD compression ratio decline with increase in input size. One MB of code and data extracted from an application is compressed as one hundred 10KB and 10 100KB files and as the original file. Compression ratio averages are reported.

To address all of these issues we have developed an algorithm for sub-stream isolation: Split^2-Stream. We demonstrate its features on x86 code. In x86 code, 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. After splitting into streams PPM captures vertical correlations in its model. Because of that, 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 a field that corresponds 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 or other m-streams. A *union* of $n > 1$ 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 4 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 bytes in the union corresponds to the byte sequence in the containing p-stream.

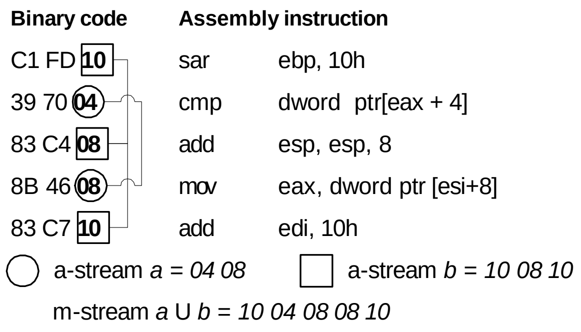


Figure 4. An example of two *a-streams* that can be merged into a single *m-stream*.

We introduce two functions: split(·) and merge(·). Function split(a, s) determines whether vertical correlation of an m-stream a is greater than its horizontal correlation with respect to its containing m-stream $s, a \subset s$. It is defined as:

$$\text{split}(a, s) = \frac{\varrho(s - a) + \varrho(a)}{\varrho(s)}, \tag{1}$$

where $s - a$ is an m-stream that represents an exclusion of a from s with respect to their containing p-stream, 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:

$$\text{merge}(a, b) = \frac{\varrho(a \cup b)}{\varrho(a) + \varrho(b)}. \tag{2}$$

Both split(·) and merge(·) functions can be greater or smaller than 1. We decide to split or to merge two streams if the result of the functions is smaller than 1, i.e., the compression ratio is improved. Both functions assume that when either splitting or merging, the order of operands in any molecular stream is preserved, i.e., when the decompressor is assembling the binary, the result is functionally correct.

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:

$$A = \{a_i \mid \text{split}(a_i, p) < 1\}. \tag{3}$$

Based on p and A, the algorithm initially creates program's *core-stream* (*c-stream*) as: $c = p - \bigcup_1^{|A|} a_i$.[7] 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 Eqn. 3. An example that shows the two types of a-streams, RAR fields (filtered) and CALL displacements (preserved), is presented in Table I.

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 $b_i, i = 1 \dots M$, 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:

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

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.

```
$B = \{a_1, a_2, \dots, a_{|A|}, c\}$
while $|B| > M$
   Replace $a \in B$ and $b \in B$ with $(a \cup b) \in B$ where
      $(a, b) = \arg\min_{c,d \in B} merge(c, d)$
end
```

Figure 5. Pseudo-code of the SPLIT^2-STREAM procedure.

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 5.

[7]Commonly, the c-stream corresponds to a stream of core-Is.

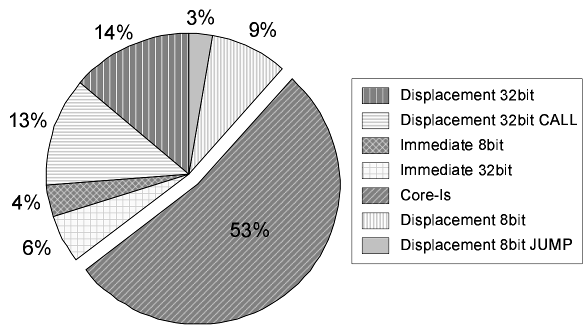


Figure 6. Relative size of the resulting sub-streams created using Split^2-Stream with $M = 7$. Legend specifies the most dominating a-stream in the respective final m-stream. Results are averaged over our benchmark suite from Table I.

Figure 6 illustrates the average relative size of m-streams in B for $M = 7$ generated over our benchmark of applications (see Table I). Streams dominated with constants and control-flow displacements, are commonly compressed at low ratios. However, the m-stream dominated with core-Is, is the performance bottleneck: it accounts for more than half the total data to be compressed and it is compressed at a ratio significantly higher than the overall compression ratio. Hence, in the next two subsections we present three techniques 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 5, we use fixed a-stream partitioning B presented in Figure 6.

We applied the Split^2-Stream algorithm to the programs listed in Table I resulting in the sub-stream choices listed in Figure 6. We have determined values for split$(\cdot)$ and merge$(\cdot)$ functions using the compression engine that is used later on for actual compression and decompression. While this approach enables high fidelity of quantified correlations, it requires a significant amount of computation and has to be recalculated for each potential sub-stream, thus significantly affecting compression speed. A similar approach was presented by [Fraser 1999] where the quantification of correlations is approximated with zero order frequency count of symbols.

3.2 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. We determine basic block boundaries by using the Vulcan framework [Srivastava and Vo 2001]. Information about basic block boundaries is needed only during compression phase when IR is performed. Finally, it is important to stress that

IR has negligible performance effect on program execution because most modern processors are capable of executing instructions out of order.

IR is performed after the binary is split into m-streams using SPLIT^2-STREAM. Decisions to reorder are based 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[8]. This set represents the symbol alphabet $\mathcal{A}$ for the input stream to PPM. We denote the cardinality of $\mathcal{A}$ with $|A|$. The input to PPM is a vector of N symbols $x \in \mathcal{A}^N$.

DEFINITION 1. *A dependency set $D(B)$ of a binary B is a set of instruction pairs, where a dependency pair $d(x_i, x_j) \in D(B)$ denotes that instructions x_i and x_j can be scheduled such that x_i immediately precedes x_j and the result of execution of the binary is not changed for all values of input data when x_i precedes x_j.*

A dependency set is formed based on three standard instruction dependency categories: "Read-After-Write", "Write-After-Write", and "Write-After-Write" [Hennessy and Patterson 1995]. It provides a set of constraints that limits IR such that the functionality of a binary B is preserved.

The goal of IR is to create a permutation of instructions in a binary B, $\pi(B)$ such that:

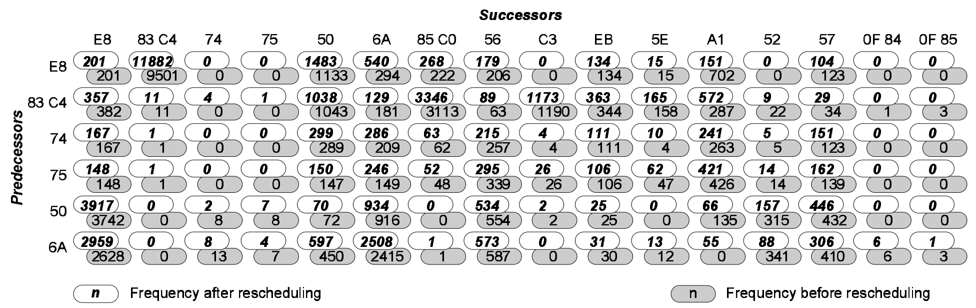
(a) all instruction pairs $x_a x_b \in \pi(B)$ are also found in $D(B)$, i.e., the functionality of basic block B is not changed by the instruction rescheduling, and

(b) a PPM prediction engine of order K emits minimal number of *escape* symbols while compressing $\pi(B)$ according to its model presented in Appendix A.

Condition (a) enforces that the rescheduled representation of a binary has the same functionality. Note that condition (a) does not imply that the instruction x_a immediately precedes x_b. Other instructions can be scheduled between x_a and x_b, but x_a is always scheduled before x_b. Condition (b) states the heuristic goal of instruction rescheduling algorithms presented in this section. In terms of PPM modeling, condition (b) aims to minimize the overhead incurred by emitting *escape* symbols, and consequently improving the final compression ratio.

3.2.1 **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 two symbol context occurrences throughout the entire binary. For the set of all contexts considered by PPM, by skewing their count distribution we heuristically aim at encoding of the input stream with fewer bits. The pseudo-code for this algorithm is shown in Figure 8. $A1$ performs iteratively the following two steps. In the first step, the algorithm computes the connectivity matrix of the target c-stream. The *connectivity matrix* $\mathcal{M}$ of a stream $x = x_1, x_2, \ldots x_N$ is the $|A| \times |A|$ matrix of non-negative integers $\mathcal{M}_{i,j}$, each denoting the number of occurrences of $a_i a_j$ in $\pi(x)$. A sample of a connectivity matrix is given in Figure 7. It illustrates how our algorithm enhances correlation for the majority of the most frequent core-Is. For example, the frequency count of the pair of core-Is $\boxed{\textbf{E8}}$ - $\boxed{\textbf{83 C4}}$ increases by more than 25% (from 9501 to 11882).

[8]For the x86 instruction set, the c-stream consists of core-Is only.



Successors

Predecessors	E8	83 C4	74	75	50	6A	85 C0	56	C3	EB	5E	A1	52	57	0F 84	0F 85
E8	***201***	***11882***	***0***	***0***	***1483***	***540***	***268***	***179***	***0***	***134***	***15***	***151***	***0***	***104***	***0***	***0***
	201	9501	0	0	1133	294	222	206	0	134	15	702	0	123	0	0
83 C4	***357***	***11***	***4***	***1***	***1038***	***129***	***3346***	***89***	***1173***	***363***	***165***	***572***	***9***	***29***	***0***	***0***
	382	11	0	0	1043	181	3113	63	1190	344	158	287	22	34	1	3
74	***167***	***1***	***0***	***0***	***299***	***286***	***63***	***215***	***4***	***111***	***10***	***241***	***5***	***151***	***0***	***0***
	167	1	0	0	289	209	62	257	4	111	4	263	5	123	0	0
75	***148***	***1***	***0***	***0***	***150***	***246***	***52***	***295***	***26***	***106***	***62***	***421***	***14***	***162***	***0***	***0***
	148	1	0	0	147	149	48	339	26	106	47	426	14	139	0	0
50	***3917***	***0***	***2***	***7***	***70***	***934***	***0***	***534***	***2***	***25***	***0***	***66***	***157***	***446***	***0***	***0***
	3742	0	8	8	72	916	0	554	2	25	0	135	315	432	0	0
6A	***2959***	***0***	***8***	***4***	***597***	***2508***	***1***	***573***	***0***	***31***	***13***	***55***	***88***	***306***	***6***	***1***
	2628	0	13	7	450	2415	1	587	0	30	12	0	341	410	6	3

n Frequency after rescheduling

n Frequency before rescheduling

Figure 7. An example of the connectivity matrix $\mathcal{M}$ for the top 6×16 most frequent unique core-Is in the benchmark program CC1. Shaded and white cells specify the frequency of occurrence of a $a_i a_j$ sequence of symbols in the binary before and after applying the algorithm $A1$.

```
Compute connectivity matrix $\mathcal{M}$
while $\max(\mathcal{M}) \geq 2$
  Find $(i, j)$ such that $\mathcal{M}_{i,j} = \max(\mathcal{M})$
  Reorder instructions to increase occurrence of pairs $x_k x_{k+1}$
    such that $d(x_k, x_{k+1}) \in D(B)$ and $x_k = a_i$ and $x_{k+1} = a_j$
  Tag $x_k x_{k+1}$ inseparable
  Update $\mathcal{M}$ and $D(B)$
  Set $\mathcal{M}_{i,j} = 0$
end
```

Figure 8. Pseudo-code for the instruction rescheduling algorithm A1. The algorithm increases the count of the most frequent instruction pairs in an attempt to reduce the count of *escape* symbols in the compressed stream.

In the next step, $A1$ finds the largest element $\mathcal{M}_{i,j}$. 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.[9] Then, element $\mathcal{M}_{i,j}$ 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 $D(B)$ correspondingly. The two steps of $A1$ are repeated until all elements of $\mathcal{M}$ are smaller than 2. An example of how the number of sequential occurrences of the most frequent core-Is in CC1 changes after several steps of $A1$, is illustrated in Figure 7.

$A1$ 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

[9]Indices of a_i and a_j correspond to the selected $\mathcal{M}_{i,j}$.

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.

	Instruction		Possible position						
			1	2	3	4	5	6	7
1	mov	esi, [eax+8]	$\otimes$	$\times$	$\times$	$*$			
2	mov	ecx, [esp+1Ch]	$\times$	$\otimes$	$\times$	$\times$	$*$		
3	mov	edx, [eax+4]	$\times$	$\times$	$\otimes$	$\times$	$*$		
4	or	esi, 100h		$\times$	$\times$	$\otimes$	$\times$		
5	cmp	ecx, edx			$*$	$\times$	$\otimes$	$\times$	
6	mov	[eax+8], esi			$*$	$*$	$\times$	$\otimes$	
7	je	LOOP							$\otimes$

Figure 9. An example of IR within a basic block. Possible position of each core-I is illustrated as $\otimes$ for current position; $\times$ for possible position; and $*$ for possible position if at least one other core-I is moved as well.

3.2.2 **Algorithm A2**. Algorithm $A1$ has a strong deficiency; it is rather slow and memory-consuming for large binaries. The complexity of algorithm $A1$ is $\mathcal{O}(|A|^2N)$, 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. Figure 10 illustrates the pseudo-code of algorithm $A2$. 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.

```
mark $x_1$ as "scheduled" and $x_2, \ldots, x_N$ as "not scheduled"
for $i = 1, \ldots, N-1$
  repeat until $x_i x_{i+1}$ are not "scheduled"
    let $S$ be a set of symbols succeeding $C(x_i, K)$ in the current PPM model
    let $F$ be a set of instructions that can succeed $x_i$
    if $F \cap S \neq \emptyset$
      $x_j = \arg\max_{x_k \in F \cap S}(P_C(x_k))$
      schedule $x_i x_j$ $(x_{i+1} = x_j)$ and mark $x_j$ as "scheduled"
    else $K = K - 1$
  end
  update PPM model
end
```

Figure 10. Pseudo-code for instruction rescheduling algorithm A2. The algorithm searches for the instruction that corresponds to the best match in the current PPM context in order to concatenate it to the current instruction. The **repeat** loop terminates always since the next instruction in the binary can be found at least in context of order-(-1).

$A2$ reschedules non-control-flow core-Is only within their basic block. For each basic block, $A2$ reschedules the core-Is with an aim to reduce the number of emitted *escape* symbols and to maximize the prediction accuracy of the PPM engine. The complexity of this problem is governed by the fact that the number of different schedules in a basic block may potentially grow exponentially with respect to the number of encompassed instructions. An example how core-Is can be rescheduled within a basic block is illustrated in Figure 9. IR freedom for our benchmark suite of programs is presented in Figure 11.

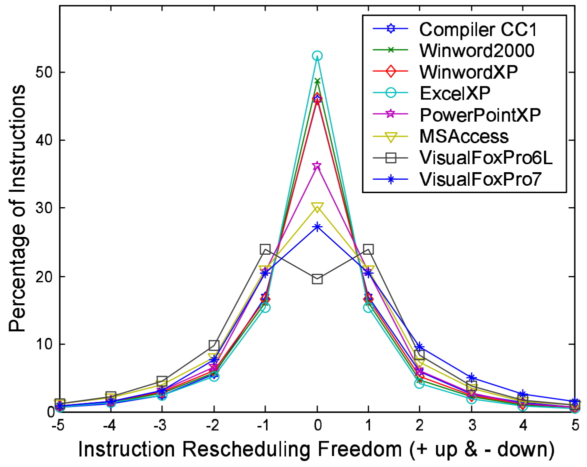


Figure 11. Percentage of instructions that can be moved up (positive x-axis) and down in a basic block for our benchmark suite.

$A2$ processes the input binary B one instruction at a time. Initially, it marks the first instruction in the binary x_1 as "*scheduled*" and all others as "*not scheduled*". Then, for the current instruction x_i, which is x_1 in the first iteration, it finds the set $F = \{x_j \mid d(x_i, x_j) \in D(B) \wedge x_j \text{ "not scheduled"}\}$ of unscheduled instructions, which can succeed x_i according to $D(B)$. In the next step, for the current context length K, the preceding context $C(x_i, K)$, and the set S of succeeding symbols after $C(x_i, K)$, $A2$ selects the instruction x_j from $F \cap S$ which has the highest likelihood of occurrence after $C(x_i, K)$. If $F \cap S = \emptyset$, an *escape* symbol must be emitted. Hence, $A2$ decrements K and repeats the previous step. This action is iterated until instruction x_j which results in the best possible prediction accuracy is identified. Ultimately, best fit must be found in the context of order-(-1). Once the earliest best fit x_j is found, $A2$ schedules x_j immediately after x_i, marks x_j as "*scheduled*", and continues to the next iteration.

In general, the results that $A2$ obtains are comparative to the results of $A1$ within 0.5% difference in the final compression ratio at a computational complexity of $\mathcal{O}(N)$. Due to this favorable encoding speed vs. compression ratio balance, the experimental results that we report in Section 5, are obtained using $A2$.

3.2.3 **Algorithm A3**. Algorithm $A3$ is a step further from $A1$ and $A2$ in significantly speeding up compression at the expense of slightly increased compression ratios with respect to $A1$ and $A2$, while still exploring IR as an optimization for software compression. Due to its conceptual simplicity, $A3$ achieves an encoding speed that is comparable to the corresponding performance of an off-the-shelf PPM algorithm. Figure 12 outlines the main steps of $A3$ in a pseudo-code format.

```
Given a block $\beta = \{x_i x_{i+1} \dots x_j\}$ of $j - i + 1$ instructions
Mark all instructions in $\beta$ as "not scheduled"
for $k = i, \dots j - 1$
   $\beta_k = \{x_l \mid x_l$ is "not scheduled" $\wedge\, d(x_{k-1}, x_l) \in D(B)\}$
   find $x_l$ such that $(\forall x_m \in \beta, l \neq m)\ x_l = f(x_m, x_l)$
   schedule $x_l$ after $x_{k-1}$ and mark $x_l$ as "scheduled"
end
schedule the one remaining instruction in $\beta$
```

Figure 12. Pseudo-code for the instruction rescheduling algorithm A3 at the position of scheduling an instruction at the position x_{k+1} after x_k is scheduled. The ordering function of two instructions x_a and x_b is denoted as $f(x_a, x_b)$.

$A3$ aims at sorting instructions within a basic block such that interdependencies among instructions are preserved while improving heuristically PPM's context matching on both global and local levels. For example, if sorting is performed in an ascending order of instructions' opcodes, then starting from the beginning of a basic block, PPM is more likely to encounter instructions that have a higher binary code. Sorting also enforces more structured instruction sequences on the global level. Since a particular policy for ordering instructions (e.g., ascending order) is permanent throughout the entire binary, the entropy of a symbol appearing after a certain context $P_C(x_i)$ is likely to decrease as the likelihood of instructions with opcodes higher and closer to the opcode of x_i is higher than in the original binary.

A single basic block $\beta = \{x_i x_{i+1} \dots x_j\}$ of $j - i + 1$ instructions is sorted in the following way. Initially, all instructions in β are tagged as "*not scheduled*". Starting from the instruction x_{i-1} that precedes x_i, we identify a pool β_i of "*not scheduled*" instructions from β that can succeed x_{i-1} with preserved instruction dependencies, i.e., $(\forall x_k \in \beta_i)\ d(x_{i-1}, x_k) \in D(B)$. The fundamental component of the sorting procedure is a comparison function $f(x_a, x_b) \in \{x_a, x_b\}$ that determines the order between instructions x_a and x_b. $A3$ sorts instructions in the increasing order of the binary codes of their core-Is. $A3$ concatenates to x_{i-1} the one instruction x_k from β_i such that $(\forall x_l \in \beta_i, l \neq k)\ x_k = f(x_k, x_l)$. Next, x_k is tagged as "*scheduled*" and the concatenation procedure is repeated for x_k as the last scheduled instruction. After $j - i$ iterations of this step, all instructions within β are scheduled, thus $A3$ proceeds to the next basic block.

Consider the example shown in Figure 13. The first two instructions do not have dependencies, hence, depending on the ordering policy, $A3$ can switch their positions. For example, if the ordering is ascending, at the binary level all pairs of core-Is $\boxed{\textbf{8B 50}}$ - $\boxed{\textbf{8B 4C 24}}$ are switched as $\boxed{\textbf{8B 4C 24}}$ - $\boxed{\textbf{8B 50}}$. As a

	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

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

b) Schedule after ordering

Figure 13. An example of IR within a basic block. Instructions are sorted by their core-Is (bold) under the constraint of dependencies among instructions.

consequence, the PPM model cannot have the $\boxed{\textbf{8B 4C 24}}$ symbol in the context that is ending with the instruction $\boxed{\textbf{8B 50}}$. This implies that fewer instructions appear in the context that ends with the instruction $\boxed{\textbf{8B 50}}$, and they are predicted with higher probability, i.e., with fewer bits.

In our experiments, $A3$ has demonstrated superior compression speed compared to $A1$ and $A2$; $A3$ was only about 30% slower in encoding than off-the-shelf PPMD. At the same time, the produced gain in compression ratio was approximately one half of the improvement achieved with $A2$ (see Table VI).

3.3 Dual Alphabet PPM

We introduce a dual alphabet PPM as a technique to improve the traditional PPM model while processing a program binary. The key idea is to identify the most frequent core-Is and use them as symbols in the PPM model. Thus, there are two alphabets used to build the model:

— $\mathcal{B}$ - symbol set of L_1 most frequent core-Is, where a core-I is a variable-length symbol typically 1 to 4 bytes long. From Figure 2, we conclude $L_1 \geqslant 256$.

— $\mathcal{A}$ - symbol set of all $L_2 = 256$ 8-bit values.

In order to use the two alphabets, the compression codec uses a disassembler to parse the input stream into symbols. Initially, the disassembler extracts the next core-I from the program. If this core-I is found in $\mathcal{B}$, then it is fed to the PPM model as the next incoming symbol. Otherwise, the extracted m-byte core-I is split into m 8-bit symbols, where each of these symbols is individually fed to the PPM model. It is important to stress that there are core-Is that are 8-bits long. The disassembler must separate the two cases (core-I or 8-bit generic symbol) before feeding the appropriate symbol to the PPM model. Due to such a separation, our model has a significantly better potential to describe correlations among instructions and isolate them from the remainder of the stream which is unrelated. An example of such separation is presented in Figure 14 where symbol **5D** Hex in the third core-I is found in $\mathcal{B}$ and core-I **DD 5D** Hex is not found as a symbol in $\mathcal{B}$. Hence, it is fed as two $\mathcal{A}$-symbols **DD** Hex and **5D** Hex to the PPM model.

An additional advantage of the dual alphabet PPM model is that it reduces the number of symbols processed. The improvements in the compression ratio with respect to traditional PPM, presented in Table III, come at the expense of approximately doubling the size of the PPM model (empirically determined). The

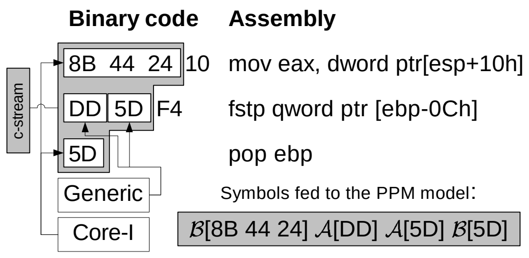

Binary code
Assembly
c-stream
8B 44 24 10
mov eax, dword ptr[esp+10h]
DD 5D F4
fstp qword ptr [ebp-0Ch]
5D
pop ebp
Generic
Core-I
Symbols fed to the PPM model:
$\mathcal{B}$[8B 44 24] $\mathcal{A}$[DD] $\mathcal{A}$[5D] $\mathcal{B}$[5D]

Figure 14. An example of how a disassembler parses the input c-stream.

advantage achieved by using a dual- as opposed to a single-alphabet compressor is established even for relatively small input streams. As an example, Figure 15 illustrates the accumulated compression ratio for two compression algorithm variants of equal parameters ($K = 4$, $L_1 = L_2 = 256$, memory size 20MB) that use a single- and dual-alphabet respectively. The compressors are applied to the $1.35 \cdot 10^6$ instructions of the VisualFoxPro7 binary. One can observe that after only 10^3 processed instructions the improvement achieved due to a dual-alphabet parsing of the input is relatively constant throughout the remainder of the compression time-line.

PPMD {bytes}	Program Name							
	Compiler CC1	Win-Word 2000	Win-Word XP	Excel XP	Power-Point XP	MS Access	Visual Fox-Pro6L	Visual Fox-Pro7
$\{\mathcal{A}\}$	196819	421675	1717025	1654522	859140	818017	876435	918302
$\{\mathcal{B},\mathcal{A}\}$	187935	402661	1627633	1579008	780306	758873	830610	877922
Imp.[%]	4.514	4.509	5.206	4.564	9.176	7.230	5.229	4.397

Table III. Compression ratio comparison of a traditional PPMD with a single alphabet and a dual alphabet PPMD. Columns 2 and 3 quantify the size of the compressed c-stream of a particular benchmark respectively. "Imp." denotes improvement.

4. RANDOM ACCESS COMPRESSION

The key feature of a compression algorithm for run-time decompression is random access: program execution must be diverted to any instruction from any control-flow operation in the binary without decompressing the entire program. This is a feature that straightforward PPM as well as many other decompression formats do not offer by default. Modifying PPM to enable this feature is even more difficult because besides a pointer to the compressed file, a PPM decompressor also needs the state of the PPM model, a message that consumes vast amount of information.

An important feature of a software delivery platform is that it partitions functions into funclets to distribute them to clients. A funclet is a minimal amount of information exchanged between a server and its clients. We recognize that most large applications need a certain code-base (kernel) to run almost any functionality. The size of the base (assumed to be 256KB) is usually much larger than a funclet.

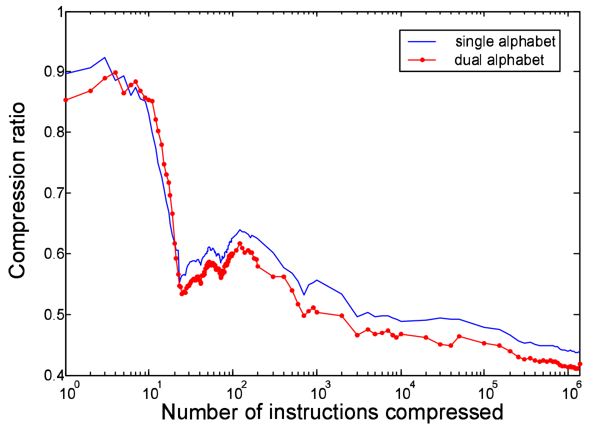


Figure 15. An example of the progress of compression ratio while compressing VisualFoxPro7 using two compression algorithms with equal parameters: $K = 4$, $L_1 = L_2 = 256$, and memory size 20MB, and with and without the Dual Alphabet technique.

A client initially downloads the code-base and then starts running the program. For each missing function, the client sends a service request to the server. The server sends the funclet containing the desired function. Finally, the client decompresses the funclet and maps its content to the corresponding memory location.

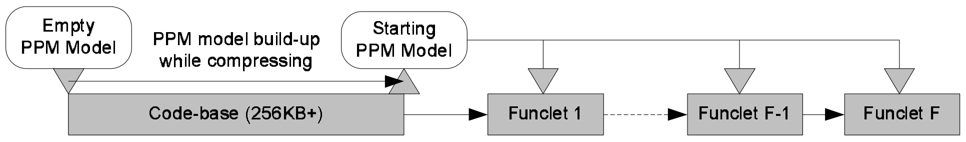


Figure 16. Funclet compression for random access.

To address the decompression requirements of a software delivery platform, we propose three different modifications to the original PPM paradigm that have the same goal. For all three techniques, we assume that the code-base is compressed first and that each funclet is compressed starting from the PPM model built after compressing the code-base. Figure 16 illustrates the process of (de)compression for random access. Any of the F funclets in the program can be decompressed using only the knowledge of the *starting* PPM model.

—Copy model (CM). This technique compresses each funclet from the starting PPM model and continues building the model while compressing. However, since every funclet is compressed using the starting model, it needs to be copied prior to funclet decompression and deleted afterwards.

—Undo model (UM). This technique slightly (5-10%) improves upon the speed of the previous technique as the compressor does not copy the model but rather maintains a log of all changes encountered while compressing the funclet. After

Code-base size	CM/UM			SM
	8KB	16KB	32KB	
64KB	6.13%	5.08%	3.95%	11.9%
128KB	2.41%	2.68%	1.62%	8.51%
256KB	0.85%	0.34%	-0.45%	4.21%

Table IV. Change in compression ratio for different random access enabling methods (CM - CM/UM and SM) for variable funclet and code-base size. Results obtained using the c-stream of the CC1 compiler and averaged over all funclets.

the funclet has been fully decompressed, the log is used to undo the changes induced to the starting PPM model. As funclet size becomes larger, the speed of UM becomes closer to the CM variant.

—Stopped model (SM). SM has strong gains in decompression speed with respect to the previous two techniques as it does not update the starting PPM model during compression and decompression. Statistically, if the program binary is a homogenous stream of symbols with respect to the PPM model, SM should have little effect on the compression ratio under the assumption that the model has been statistically saturated to a level which enables near-maximum performance.

Experimental data presented in Table IV quantifies the impact of CM/UM[10] and SM on compression ratio as code-base and funclet size vary. The compressed data is the c-stream of the CC1 compiler. Note, that for the CM/UM method, code-base and funclet size of only 256KB and 32KB respectively, yields a compression ratio that is within the variance of the compression ratio of a classical PPM model with no random access capability. Actually, in our experiment, we have obtained even a slight improvement. Thus, the random access mechanisms that we propose are effective because PPM models of binaries saturate quickly due to the homogeneity of the target content. In addition, results in Table IV demonstrate that saturation of the PPM model is a factor far more important than funclet size as compression ratio slightly varies with this parameter.

We assume that the random access decompression is performed in the background and processes of copying and undoing of the model do not affect final decompression speed. This effect is dependent of several system parameters including the rate of fetching new code from the server and funclet pre-fetching algorithms. The overall system analysis and possible pre-fetching strategies are beyond the scope of this paper. In cases when the decompression speed is of ultimate importance, SM can be used since its decompression speed shows even a slight improvement over the same compressor that does not enable random access decompression at the cost of 3 to 8% increase in compression ratio.

5. EXPERIMENTAL RESULTS

A lower bound on the compression ratio for program binaries is an undecidable problem [Chaitin 1966; 1969; Kolmogorov 1965]. Even subproblems related to the

[10]CM and UM have equivalent compression ratios, but different decompression run-times.

minimization of program binary size such as eliminating functionally unreachable statements or finding a sequence of program transformation that minimizes program size are NP-hard problems [Hopcroft and Ullman 1979; Hong et al. 1997]. Program binaries have complex intra-correlations that are hard to extract and analyze. In particular, this analysis is computationally expensive.

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. We used the powerful Vulcan framework to manipulate x86 binaries [Srivastava and Vo 2001]. For the benchmark, we used pre-computed and fixed sub-stream separation presented in Subsection 3.1. We removed from each program the gluing data blocks that exist in x86 binaries. These blocks can be compressed with ratios significantly lower than the remainder of the data in the executable.

5.1 Compression Ratio

The experimental results for the compression ratio achieved by our compression algorithm, named **PPMexe** are presented in Table V. It is important to stress that x86 instruction set is designed in such a way that it reduces the size of code. This is accomplished via variable instruction length where the most frequently used instructions are encoded with the shortest code. Size of uncompressed programs excludes the data blocks. We compared our technology with the two most competitive general-purpose compression technologies today: bzip2 [Burrows and Wheeler 1994], which uses the Burrows-Wheeler transform and PPMD [Howard 1993]. All compression mechanisms used 10MB of operating memory. PPMexe demonstrated superb performance with respect to bzip2 and PPMD, outperforming them on the average for 26.3% and 20.4% respectively. SPLIT^2-STREAM, DUAL ALPHABET, and INSTRUCTION RESCHEDULING contributed to the improvement with an approximately 2:1:1 ratio.

Program Name	Uncompressed File Size	bzip2 C. Ratio	PPMD C. Ratio	PPMexe C. Ratio	Improvement [%]	
					PPMD	bzip2
Compiler CC1	872588	0.5126	0.4770	0.3917	17.9	23.6
Winword2000	6137358	0.5669	0.5255	0.4273	18.7	24.6
WinwordXP	7535800	0.5614	0.5201	0.4206	19.1	25.1
ExcelXP	7303411	0.5624	0.5213	0.4215	19.1	25.0
PowerPointXP	4449093	0.5212	0.4812	0.3660	23.9	29.8
MSAccess	3942060	0.5275	0.4870	0.3887	20.8	26.3
VisualFoxPro6L	3769781	0.5662	0.5244	0.4061	22.6	28.3
VisualFoxPro7	3941408	0.5752	0.5334	0.4167	21.9	27.6
Average Improvement					20.5	26.3

Table V. Compression ratio comparison for a benchmark of large applications: bzip2 vs. PPMD vs. our technology PPMexe. File size is reported in bytes. Reported numbers for PPMexe refer to the $A2$ algorithm, $M = 7$ m-streams, and PPM model of order $K = 4$ with $L_1 = L_2 = 256$ symbols for each alphabet. Label "C. Ratio" refers to compression ratio.

Impact of SPLIT^2-STREAM, DUAL ALPHABET and INSTRUCTION RESCHEDULING on the compression ratios is presented in Table VI. The effects of SPLIT^2-STREAM and Instruction Rescheduling on compression ratio are not orthogonal. Instruction rescheduling without splitting them into streams has a limited effect on compression ratio because horizontal and vertical correlations are opaque to the compression algorithm. However, when instructions are split, horizontal and vertical correlations of streams are exposed. Even on an architecture with typically small basic block size (such as x86), instruction rescheduling improves exposed correlations and thus, the compression ratio. The effect of Instruction Rescheduling without Split^2-Stream and Dual Alphabet is not commeasurable with its full effect when all techniques are applied, thus, it is not shown separately.

Program Name	PPMD	S^2-S	S^2-S + DA	PPMexe	Random Access
Compiler CC1	0.4770	0.4311	0.4173	0.3917	0.4022
Winword2000	0.5255	0.4759	0.4522	0.4273	0.4375
WinwordXP	0.5201	0.4724	0.4647	0.4206	0.4306
ExcelXP	0.5213	0.4697	0.4630	0.4215	0.4367
PowerPointXP	0.4812	0.4172	0.3941	0.3660	0.3782
MSAccess	0.4870	0.4143	0.4018	0.3887	0.4017
VisualFoxPro6L	0.5244	0.4552	0.4349	0.4061	0.4178
VisualFoxPro7	0.5334	0.4766	0.4395	0.4167	0.4266

Table VI. Comparison of compression ratios of techniques applied in the PPMexe technology. The columns represent compression ratio without any techniques applied (PPMD), with Split^2-Stream (S^2-S), with S^2-S and Dual Alphabet (S^2-S + DA), with S^2-S, Dual Alphabet, and Instruction Rescheduling (PPMexe), and with all techniques with random access decompression for CM/UM with 128KB code-base size and 16KB funclet size. We used the same maximum size of PPM models across all experiments.

5.2 Compression Ratio vs. Operating Memory

Compression ratio is significantly influenced by the available operating memory used to build the PPM model. This dependency is particularly important for embedded systems, as they commonly use resources that are more restricted than general purpose computing platforms. Figure 17 illustrates the variation in compression ratio as the operating memory available to PPMexe increases from 256KB to 16MB for the set of benchmark applications considered in this manuscript.

5.3 Decompression Throughput

Our decompression software reported minimum 1.2MB/s throughput on a 2.8GHz Pentium IV, i.e., several times larger than the bandwidth of a generous DSL link between a server and a client. The detailed decompression throughput numbers and decompression times are shown in Table VII. The reduction in decompression throughput for CM/UM without any part being executed as a background process ranges from 1080 KB/s for ExcelXP with 256 KB code base size (larger model to copy/undo) and 8 KB of funclet size, to 1270 KB/s for CC1 with 64KB code base size (smaller model to copy/undo) and 32 KB funclet size. On the average, using CM/UM to enable random access decompression produced a 15% reduction,

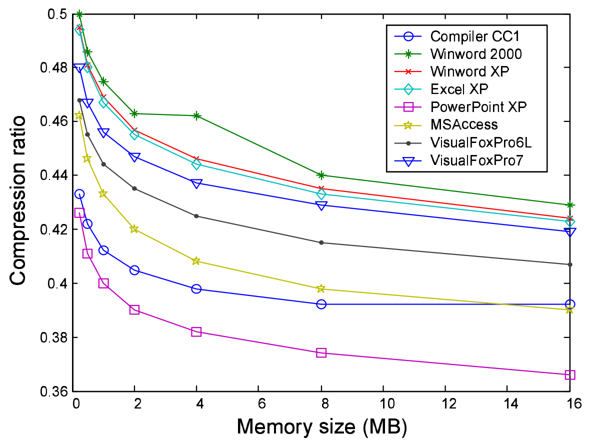


Figure 17. Variation in the compression ratio as operating memory available to PPMexe for building its data models increases from 256KB to 16MB for our set of benchmark applications.

while using SM for the same purpose produced a 10% increase in decompression throughput compared to the case when random access decompression was not enabled (see Table VII). All experiments were performed with 10MB of available operating memory for the PPM model.

PPM as a compression paradigm is inherently inferior with respect to its compression/decompression speed to bzip2. The bzip2 decompression algorithm yielded higher decompression throughput of about 2MB/s on our benchmark suite. Hence, the total download and decompression time favors PPMexe (assuming a 1.2MB/s decompression throughput of PPMexe) for all download links with bandwidth smaller than 3.5Mb/s.

PPM builds the exact same model during both compression and decompression. This results in approximately equivalent run times during both procedures. However, during compression PPMexe uses the Vulcan platform [Srivastava and Vo 2001] to provide pointers to code, data, and other sections of the program binary. The duration of this step depends on the size of the binary. Typically, as a result, PPMexe compression is approximately three times slower than decompression. The importance of this characteristic is minor in most applications as binaries are compressed only once, while decompression occurs on many clients.

Program Name	Compiler CC1	Win-Word 2000	Win-Word XP	Excel XP	Power-Point XP	MS Access	Visual Fox-Pro6L	Visual Fox-Pro7
T-put[KB/s]	1547.7	1332.3	1329.0	1272.3	1471.6	1508.5	1227.9	1250.8
Time[sec]	0.7	4.8	5.6	5.5	3.1	2.6	3.1	3.2

Table VII. Decompression throughput and decompression times for PPMexe on a 2.8GHz Pentium IV platform.

5.4 Impact of Instruction Rescheduling on Performance

Advanced architectures provide an execution environment in which INSTRUCTION RESCHEDULING affects the execution speed only negligibly. For example, the Pentium III architecture utilizes an advanced dataflow analysis which creates an optimized, reordered schedule of instructions by analyzing data dependencies [INTEL CORP. 1999a]. Similarly, the Pentium 4 Advanced Dynamic Execution mechanism analyzes 126 instructions simultaneously in order to enable deep out-of-order speculative execution [INTEL CORP. 2000]. For such architectures, INSTRUCTION RESCHEDULING minimally affects the execution speed of programs since the presented algorithms are restricted within basic blocks that are typically smaller than the number of instructions a processor considers at the same time. Other architectures such as statically scheduled pipelined processors, can be strongly affected by INSTRUCTION RESCHEDULING. In these cases, the algorithm for rescheduling must take into account the scheduling constraints imposed by the optimization for performance.

In order to further demonstrate our claim that the effect of IR on program runtime is negligible, we use Figure 18 to illustrate the histogram of run-times for the `gzip` binary from the SPEC2000 benchmark suite [Weaver 2000]. In our experiment, we executed 40 runs of the original optimized binary using the data-set provided by SPEC, and three runs for each of the 39 different randomly rescheduled versions of the same binary and one version obtained by our INSTRUCTION RESCHEDULING algorithm. During the process of random rescheduling, we considered each pair of instructions that can be rescheduled and randomly decided if instructions should exchange their positions. For each of the randomly rescheduled binaries, we considered the average processor run-time based on the three runs for each version of the binary. We observed that the mean and standard deviation for the two sets of runs did not have any significant statistical difference. The mean and variance for the executions of the original binary were $\mu_1 = 238$ and $\sigma_1 = 4.42$ seconds respectively, and correspondingly $\mu_2 = 237$ and $\sigma_1 = 3.10$ seconds for the rescheduled binaries. This data suggests that the effect of INSTRUCTION RESCHEDULING on program run-time is negligible.

6. RELATED WORK

We trace the related work along two directions: program binary compression and generic PPM compression variants. Code compression has been around in the research community for more than forty years [Korolev 1958]. Research directions can be categorized as: static program compression, compression of intermediate representations, techniques for reducing uncompressed program size, and techniques for system issues involved in running compressed code.

Code compression has been addressed by computing a dictionary using set-covering algorithms [Liao et al. 1995] and expression trees [Araujo et al. 2000]. Lucco proposed a split-stream format for JIT interpretation of compressed code that halves program-size [Lucco 2000]. However, due to program interpretation the program performance decreases by 6.6% given unconstrained memory and by 27% given a JIT translation buffer of one third of the binary code size. Random access decompression has been enabled by resetting the statistical model used for

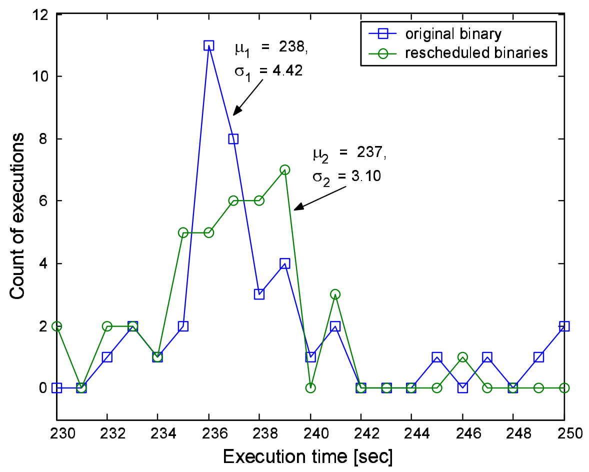


Figure 18. Impact of Instruction Rescheduling on execution time of the `gzip` binary from the SPEC2000 benchmark suite.

compression for each block of data [Lekatsas and Wolf 1999].

Compression of program's intermediate representation has resulted in impressive compression ratios [Pugh 1999]. However, software publishers are hesitant to deliver code in this format due to concerns about reverse engineering [Systa et al. 2001]. Techniques have been developed for machine independent compressed code with adaptive compression of syntax trees [Franz and Kistler 1997] and a "wire-code" compression format that can be interpreted on-the-fly without decompression [Ernst et al. 1997]. Fraser explored the bounds on the compression ratio for the intermediate representation of binaries by using machine learning techniques to automatically infer a decision tree that separates intermediate representation code into streams that compress better than the undifferentiated code [Fraser 1999].

Techniques for reducing code size include procedural abstraction and generalization of cross-jumps [Fraser et al. 1984] and common instruction merger into super-operators [Proebsting 1995]. Lau et al. introduced a technique for embedded systems that replaces code segments that are repeated throughout a binary with parameterized *echo* instructions [Lau et al. 2003]. A good survey of transformations for reduced code size is outlined in [Debray et al. 2000].

Architectures for minimized code size have been explored using instruction selection [Liao et al. 1995] and binding operand identifiers [Hoevel and Flynn 1977]. Operating system and processor support for execution of compressed code on existing or modified architectures has been detailed in [Wolfe and Chanin 1992] and [Kirovski et al. 1997]. Storage management in systems with compressed binaries has been addressed in [Murtagh 1991], [Liao 1996], and [Rao and Pande 1999]. Compression of program execution traces involves techniques for both compressing executed code as well as data associated with the execution [Burtscher et al. ; Zhang and Gupta 2005].

The PPM compression paradigm introduced by Cleary and Witten [Cleary and Witten 1984], has set the bar on compression ratio performance that no other algorithm has been able to reach to date. Moffat's improvement, PPMC [Moffat 1990], set the benchmark for over a decade, until recently, when Howards' variant, PPMD [Howard 1993], with improved computation of escape symbols was recognized as one of the best overall compression schemes [Gilchrist 2000].

Acknowledgments

We thank Dr. Mark N. Wegman and the anonymous reviewers for comments that have improved the presentation of this manuscript.

7. CONCLUSION

Systems such as software delivery platforms, embedded systems, and mobile code have imposed strong requirements for low compression ratios of binaries. In this work, we have explored compression algorithms that focus exclusively on program binaries. We have adopted as a foundation of our algorithm, prediction by partial matching (PPM) - a compression paradigm that demonstrates 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 several pre-processing steps to PPM as well as several modifications to the generic PPM technology that significantly improve the achieved compression ratio. The pre-processing steps include: rescheduling of instructions to improve prediction rates and heuristic partitioning of a binary into highly correlated substreams. We improve the traditional PPM algorithm by extracting a common alphabet of variable-length super-symbols from the input stream of fixed-length symbols. The key mechanism for enabling dynamic decompression of program binaries is random-access. We introduce several techniques that enable this feature with different effects on the decompression speed vs. compression ratio trade-off.

The developed techniques are generic in the sense that our implementation can be slightly modified to compress binaries for other instruction sets, programs represented using intermediate formats, or software patches. We demonstrated the effectiveness of the developed techniques by building a compression codec for x86 binaries. The tool was tested on a benchmark consisting of several large applications. PPMexe yielded compression rates over the benchmark suite that were 18-24% better than rates achieved by off-the-shelf PPMD, one of the best available compression tools [Gilchrist 2000; Witten et al. 1999].

REFERENCES

Araujo, G., Centoducatte, P., Azevedo, R., and Pannain, R. 2000. Expression tree based algorithms for code compression on embedded RISC architectures. *IEEE Transactions on VLSI Systems 8,* 5, 530–3.

Baker, B. S. and Manber, U. 1998. Deducing similarities in Java sources from bytecodes. *Proceedings of the USENIX Annual Technical Conference*, 179–190.

Bunton, S. 1997. Semantically motivated improvements for PPM variants. *The Computer Journal 40,* 2/3, 76–93.

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

Burtscher, M., Ganusov, I., Jackson, S. J., Ke, J., Ratanaworabhan, P., and Sam, N. B. The VPC trace-compression algorithms. *IEEE Transactions on Computers 54,* 11.

Chaitin, G. J. 1966. On the length of programs for computing finite binary sequences. *Journal of the ACM (JACM) 13,* 4, 547–69.

Chaitin, G. J. 1969. On the length of programs for computing finite binary sequences: statistical considerations. *Journal of the ACM (JACM) 16,* 1, 145–59.

Cleary, J. and Witten, I. 1984. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications 32,* 4, 396–402.

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

Ernst, J., Evans, W., Fraser, C., Lucco, S., and Proebsting, T. 1997. Code compression. *ACM SIGPLAN Programming Languages Design and Implementation*, 358–65.

Franz, M. and Kistler, T. 1997. Slim binaries. *Communications of the ACM 40,* 12, 87–94.

Fraser, C. 1999. Automatic inference of models for statistical code compression. *ACM SIGPLAN Programming Languages Design and Implementation*, 242–6.

Fraser, C., Myers, E., and Wendt, A. 1984. Analyzing and compressing assembly code. *ACM SIGPLAN Symposium on Compiler Construction 19*, 117–21.

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

Hennessy, J. L. and Patterson, D. A. 1995. *Computer Architecture: A Quantitative Approach*, Second ed. Morgan Kaufman, San Francisco, Ca.

Hoevel, L. W. and Flynn, M. J. 1977. The structure of directly executed languages: a new theory of interpretive system design. Tech. Rep. CSL-TR-77-130, Stanford University.

Hong, I., Kirovski, D., and Potkonjak, M. 1997. Potential-driven statistical ordering of transformations. *Design Automation Conference*, 347–52.

Hopcroft, J. E. and Ullman, J. D. 1979. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA.

Howard, P. 1993. The design and analysis of efficient lossless data compression systems. Ph.D. thesis, Brown University.

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

Huffman, D. 1952. A method for construction of minimum redundancy codes. *Proceedings of the IEEE 40*, 1098–101.

Intel Corp. 1999a. http://www.intel.com/design/pentiumiii.

Intel Corp. 1999b. Intel architecture software developer's manual, volume 2: Instruction set reference manual. *http://developer.intel.com/design/processor/*.

Intel Corp. 2000. http://www.intel.com/design/pentium4.

Kirovski, D., Kin, J., and Mangione-Smith, W. H. 1997. Procedure based program compression. *International Symposium on Microarchitecture*, 204–13.

Kolmogorov, A. N. 1965. Three approaches to the quantitative definition of information. *Problems of Information Transmission 1,* 1, 1–7.

Korolev, L. 1958. Coding and code compression. *Journal of the ACM 5,* 4, 328–33.

Lau, J., Schoenmackers, S., Sherwood, T., and Calder, B. 2003. Reducing code size with echo instructions. *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 84–94.

Lekatsas, H. and Wolf, W. 1999. Random access decompression using binary arithmetic coding. *Data Compression Conference*, 306–15.

Liao, S. 1996. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems 18,* 2, 235–53.

Liao, S., Devadas, S., Keutzer, K., and Tjiang, S. 1995. Instruction selection using binate covering for code size optimization. *ACM IEEE International Conference on Computer-Aided Design*, 393–9.

Lucco, S. 2000. Split-stream dictionary program compression. *ACM SIGPLAN Programming Languages Design and Implementation*, 27–34.

Moffat, A. 1990. Implementing the PPM data compression scheme. *IEEE Transactions on Communications 38,* 11, 1917–21.

Mohney, D. 2003. It's all about the last mile. *Available on-line at http://www.theinquirer.net*.

Murtagh, T. 1991. An improved storage management scheme for block structured languages. *ACM Transactions on Programming Languages and Systems 13,* 3, 327–98.

Proebsting, T. 1995. Optimizing a ANSI C interpreter with superoperators. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 322–32.

Pugh, W. 1999. Compressing Java class files. *Programming Language Design and Implementation*, 247–58.

Rao, A. and Pande, S. 1999. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. *ACM SIGPLAN Programming Languages Design and Implementation*, 128–38.

Rissanen, J. 1978. Modeling by shortest data description. *Automatica 14*, 465–71.

Rissanen, J. and Mohiuddin, K. M. 1989. A multiplication-free multialphabet arithmetic code. *IEEE Transactions on Communications 37,* 3, 129–46.

Romer, T. H., Lee, D., Voelker, G. M., Wolman, A., Wong, W. A., Baer, J.-L., Bershad, B. N., and Levy, H. M. 1996. The structure and performance of interpreters. *ACM Architectural Support for Programming Languages and Operating Systems*, 150–159.

Shannon, C. 1951. Prediction and entropy of printed english. *Bell Systems Technical Journal*, 50–64.

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

Systa, T., Yu, P., and Muller, H. 2001. Shimba - an environment for reverse engineering Java software systems. *Software Practice and Experience 31,* 4, 371–94.

Truman, T., Pering, T., Doering, R., and Brodersen, R. 1998. The InfoPad multimedia terminal: A portable device for wireless information access. *IEEE Transactions on Computers 47,* 10, 1073–87.

Weaver, C. 2000. SPEC2000 binaries. *Available on-line at http://www.simplescalar.org*.

Witten, I., Moffat, A., and Bell, T. 1999. *Managing gigabytes: Compressing and indexing documents and images.* Morgan Kaufmann, San Francisco, CA.

Wolfe, A. and Chanin, A. 1992. Executing compressed programs on an embedded RISC architecture. *International Symposium on Microarchitecture*, 81–91.

Zhang, X. and Gupta, R. 2005. Whole execution traces and their applications. *ACM Transactions on Architecture and Code Optimization (TACO) 2,* 3, 301–34.

Ziv, J. and Lempel, A. 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions Information Theory IT-24*, 530–36.

A. PPM OVERVIEW

In this section, we detail the generic PPM engine as a backbone for our code compression algorithm presented in Section 3. Let's 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. The PPM model consists of sets of frequency counts for each symbol that appears for each previously seen context in the input data stream. The maximal length of recorded contexts is typically set to a constant value. PPM estimates the amount of information for the symbol being processed based on the frequency count of that symbol in the encountered context [Cleary and Witten 1984]. It has been shown that increasing the maximum order beyond five improves the compression ratio only negligibly [Bunton 1997]. 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 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.

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.

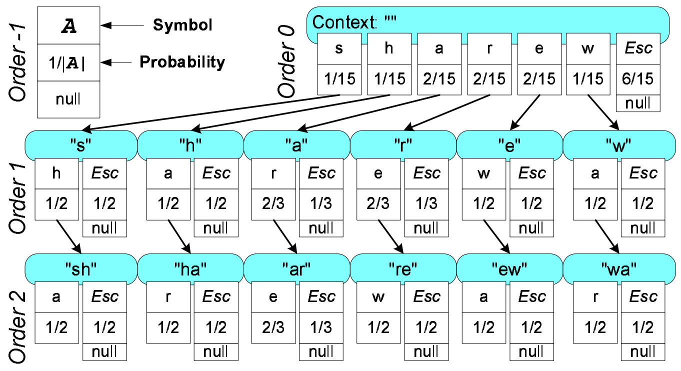


Figure 19. A PPM model with maximum order of two after processing the string "*shareware*".

An example of a PPM model with a maximal order of two after processing the string "*shareware*" is illustrated in Figure 19. The model records the occurrence of each symbol in its order-0 context. For each new symbol, the ε counter of the order-0 model is incremented. Central to a PPM implementation is the digital search tree or trie [Witten et al. 1999]. 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 ε symbol.

A.1 Arithmetic coding

PPM uses arithmetic coding to encode predicted and *escape* symbols according to the probability of their occurrence after a certain context. An arithmetic coder (AC) converts an input stream of arbitrary length into a single rational number within $[0, 1)$. Details of the arithmetic coding codec can be found in [Rissanen 1978]. The principal strength of arithmetic coding is that it can compress arbitrarily close to the entropy [Shannon 1951; Rissanen 1978].

Arithmetic coding is commonly described through examples [Witten et al. 1999]. We show how a new symbol "*s*", concatenated to "*shareware*", is encoded assuming:

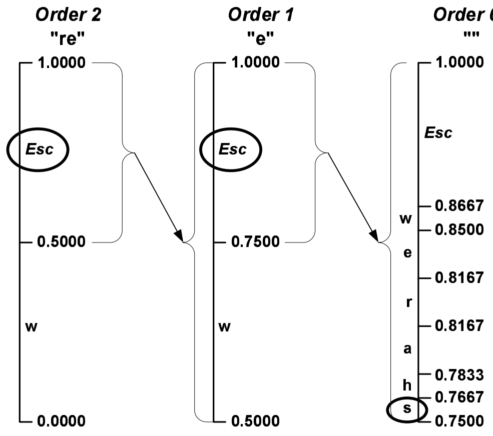


Figure 20. An arithmetic coding example based on the PPM model for the input *shareware* and an upcoming symbol s.

- previous symbols have already been encoded,
- the range of the AC has been reset to $[0, 1)$, and
- the PPM model is in a state as presented in Figure 19.

The example is illustrated in Figure 20. The longest, order-2, context for "s" is "**re**". According to the current PPM model, $P_{\text{“}re\text{”}}(w) = P_{\text{“}re\text{”}}(\varepsilon) = 1/2$. Thus, the AC divides its range into two subranges $[0, 0.5)$ and $[0.5, 1)$, each representing "w" and ε respectively. Since "s" has not been previously recorded after "**re**", current context is switched to order-1 context "**e**" and an ε symbol is emitted by limiting the AC range to $[0.5, 1)$. Since $P_{\text{“}e\text{”}}(w) = P_{\text{“}e\text{”}}(\varepsilon) = 1/2$, the AC subdivides further its range to $[0.5, 0.75)$ and $[0.75, 1)$ for "w" and ε respectively. Symbol "s" has never occurred after "**e**", thus, the current context is switched to order-0 context "" and another ε is emitted by shrinking the AC range to $[0.75, 1)$. The order-0 context model contains 7 symbols, including "s". Thus, the current AC range is further subdivided into 7 subranges for each symbol. The length of each subrange corresponds to the probability of occurrence for the corresponding symbol. To encode "s", the AC reduces its range to $[0.75, 0.7667)$. If "s" were the last symbol to be encoded, the AC would output as a result of the compression any number within $[0.75, 0.7667)$. Given the starting PPM model, any number within $[0.75, 0.7667)$ uniquely identifies the symbol "s" at the decoder. After processing a single symbol, the PPM model is updated.

The AC iteratively reduces its operating range until the leading digits of the high and low bound are equal. Then, the leading digit can be transmitted. In the example above where AC range equals $[0.75, 0.7667)$, digit 7 can be transmitted and the updated AC range becomes $[0.5, 0.6667)$. This process, called *renormalization*, enables compression of files of any length on limited precision arithmetic units. Performance improvements of classic AC focus on: precomputed approximations of arithmetic calculations [Howard and Vitter 1993] and replacing division/multiplication with shifting/addition [Rissanen and Mohiuddin 1989].

A.2 PPM Improvements

A number of PPM variants have been developed as an improvement to the original algorithm [Cleary and Witten 1984]. The main difference between variants of PPM is how the *escape* probabilities are calculated. For example, the original, variant A of PPM, has fixed the count of *escape* symbols to 1; variant D halves the weights assigned to *escape* as oppose to the other, already recorded, symbols [Howard and Vitter 1993]. Other methods include the calculation of *escape* probabilities using heuristics that account for the number of symbols that occurred in a particular context. A standard improvement of PPM, exclusion, assumes that only the context where the symbol is found as well as higher order contexts are updated in the PPM model [Witten et al. 1999].