

# PPMexe: PPM for Compressing Software

Milenko Drinić<sup>†</sup> and Darko Kirovski<sup>‡</sup>

<sup>†</sup> Computer Science Department, UCLA, Los Angeles, USA

<sup>‡</sup> Microsoft Research, One Microsoft Way, Redmond, USA

## Abstract

With the emergence of software delivery platforms such as Microsoft's .NET, code compression has become one of the core enabling technologies strongly affecting system performance. In this paper, we present *PPMexe* - a set of compression mechanisms for executables that explores their syntax and semantics to achieve superior compression rates. The fundament of *PPMexe* is the generic paradigm of prediction by partial matching (PPM). We combine PPM with two pre-processing steps: instruction rescheduling to improve prediction rates and partitioning of a program binary into streams with high auto-correlation. We improve the traditional PPM algorithm by using: an additional alphabet of frequent variable-length super-symbols extracted from the input stream of fixed-length symbols and a low-overhead mechanism that enables decompression starting from an arbitrary instruction of the executable, a feature pivotal for run-time software delivery. *PPMexe* was implemented for x86 binaries and tested on several large Microsoft applications. Binaries compressed using *PPMexe* were 16-23% smaller than files created using PPMD, the best available compressor.

## 1 Introduction

Software delivery platforms such as Microsoft's .NET, depend tremendously on compression of binaries for two reasons. First, client performance is 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 rate (CR)<sup>1</sup> of program binaries.

Code compression is a difficult problem for several reasons. First, the best off-the-shelf compression algorithms succeed in compressing binaries at the 0.50 level, leaving the impression that significant improvements can be achieved. Second, it is difficult to obtain a lower bound on the CR because computing the entropy of a binary is an NP-hard problem. As opposed to text which cannot be modified due to compression, and multimedia which is commonly compressed using lossy algorithms, executables can be transformed in many ways while still preserving the same functionality. Finding the optimal list of transformations that only minimize uncompressed code size is an NP-hard problem [8]. Finally, program binaries have complex intra-correlations that are hard to extract and analyze.

---

<sup>1</sup>CR equals the size of the compressed over the size of the uncompressed data file.

In this paper, we present compression mechanisms for program binaries that explore their syntax and semantics to achieve superior CR. The fundament of our algorithm is prediction by partial matching (PPM) [3]. We combine this powerful compression paradigm with the following techniques for performance improvement:

- (i) **INSTRUCTION RESCHEDULING.** We have developed a heuristic that reschedules instructions while preserving their dependencies with an aim to maximize the correct predictions of a PPM predictor.
- (ii) **SPLIT<sup>2</sup>-STREAM** initially breaks a program into a large number of highly auto-correlated sub-streams and then, heuristically merges certain sub-streams to reduce the CR increase which typically occurs when a PPM-like algorithm compresses small amounts of data.
- (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 it represents a list of most frequent unique instructions in the program.

A software delivery system at the client may decompress the program in two ways: entirely before execution and partially, caching frequently used program pages at run-time. To address the latter, more viable case, we introduce three different variants of PPM for:

- (iv) **RANDOM ACCESS DECOMPRESSION.** The variants are: model copying, model undoing, and model stopping. This modification enables random access decompression, a feature pivotal for run-time software delivery. However, they respond differently to the decompression speed vs. compression rate trade-off.

We implemented PPMexe for x86 binaries and tested its performance on several large Microsoft applications. Note that raw x86 code is already compressed to certain extent because instructions are variable-length, i.e. some form of Huffman coding is already applied. Binaries compressed using PPMexe were 16-23% smaller than files created using off-the-shelf PPMD [9], the best available compressor [1]. In addition, we were able to facilitate random access decompression at a negligible increase in file-size.

## 1.1 Related Work

A good survey of transformations that reduce uncompressed code size is presented in [5]. An example of run-time code decompression is Lucco's split-stream format for JIT interpretation of compressed code that halves program-size at the expense of 27% decrease in program performance [12]. Similar results were obtained by using the "wire-code" compression format that can be interpreted on-the-fly without decompression [6]. Fraser explored the bounds on the CR for the intermediate representation (IR) of code by using machine learning techniques to automatically infer a decision tree that separates IR code into streams that compress better than the undifferentiated code [7]. Random access decompression has been enabled by resetting the statistical model used for compression for each block of data [11].

The PPM compression paradigm introduced by Cleary and Witten [3], has set the bar on CR performance that no other algorithm has been able to reach to date. Moffat's improvement, PPMC [13], set the benchmark for several years, until recently, when Howards' variant,

PPMD [9], with improved computation of escape symbols was recognized as the best overall compression scheme [1]. Latest variants, such as PPMZ [2] and PPM\* [4] improve slightly upon PPMD's performance at the expense of significant increase in codec run-time.

## 2 PPMexe - The Compression Techniques

### 2.1 Split<sup>2</sup>-Stream

A PPM model is aware only of repetitive local correlations between input symbols. Since most PPM compressors operate with 8-bit symbols, a PPM model can observe only the correlations that happen at byte boundaries. However, the x86 code<sup>2</sup> has a significantly richer structure than text. An x86 instruction is variable-length (1-16 bytes) and with the following fields: an optional *instruction prefix*, an *opcode* with register and addressing specifications, an address *displacement* for control-flow instructions (optional), and *immediate data* (optional). Details of the format of x86 instructions can be found in [10].

Program name	Compressed	Exp. A.	Exp. B.
	[bytes]	[%]	[%]
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 1: *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 (A.) the register and register addressing fields and the program remainder and (B.) the displacements of CALL instructions and the program remainder.*

CR can be improved by making PPM aware of correlations that exist across instruction fields. 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 type of a field across all instructions. While building its model, PPM observes and addresses only horizontal correlations among neighboring fields. We demonstrate how vertical and horizontal correlation affect the CR of PPMD using two experiments. We extract the register and register addressing (RAR) fields from all instructions (A.) and all 4 byte displacements of CALL instructions (B.) and compress separately the two resulting files (the extraction and the remainder). Note that no additional information needs to be stored in the compressed file to assemble the two streams during decompression. From the results presented in Table 1, we observe that A results in CR *increase*, clearly indicating that horizontal correlation of the RAR field is stronger than the vertical. Similarly, the improved CR in B demonstrates slightly higher correlation of extracted displacements along their vertical.

To address all of these issues we have developed an algorithm for sub-stream isolation: SPLIT<sup>2</sup>-STREAM. The key characteristic of x86 code is that instruction's prefix and opcode

<sup>2</sup>Without 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.

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.

Function  $split(a, b)$  determines whether vertical correlation of a stream  $a$  is greater than its horizontal correlation with respect to its containing stream  $s, a \subset s$ . It is defined as:

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

where  $s - a$  is a stream that represents an exclusion of  $a$  from  $s$ , and  $\varrho(x)$  returns the size of the PPM-compressed argument stream  $x$ . The input to the SPLIT<sup>2</sup>-STREAM algorithm is a set of streams  $A$  extracted from the initial program stream  $p$  such that:

$$(\forall a_i \in A) split(a_i, p) > 0. \quad (2)$$

Set  $A$  is created by: first, considering the set of all 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. 2. The remaining stream  $c = p - \bigcup_{i=1}^{|A|} a_i$ , contains only core instructions (i.e. core-I), where a *core-I* is an instruction with removed constants and control-flow displacements. An example of a core-I is shown in Figure 1.

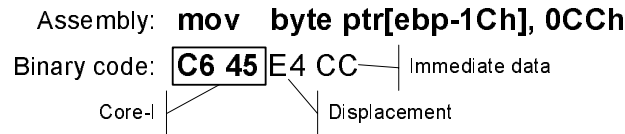


Figure 1: An example of an x86 instruction and its components: its core-I, an 8-bit displacement, and 8-bit immediate data.

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 streams from  $A'$  in  $b_i$ . For commonly considered  $|A| = 25$  and  $5 < M < 10$ , exhaustive search is computationally too expensive, hence, we 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 streams  $a, b \subset B$  with minimal  $merge(a, b)$ , where function:

$$merge(a, b) = \ln \left( \frac{\varrho(a \cup b)}{\varrho(a) + \varrho(b)} \right) \quad (3)$$

evaluates the effect of merging streams  $a$  and  $b$  on the resulting CR. Then, the heuristic merges  $a$  and  $b$  into a single stream  $f$  and replaces streams  $a, b$  with  $f$  in  $B$ .

Most of the resulting sub-streams can be compressed well bellow the CR of the original program. However, the stream containing core-Is is the CR bottleneck as it represents 50% of the total source data and compresses at a CR significantly higher than the remaining streams. Hence, we present two techniques that aim at reducing the CR of this stream.

## 2.2 Instruction Rescheduling

In this subsection, we present an algorithm for instruction rescheduling (IR) that improves 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. IR operates only on the core-I stream. Changes in the order of instructions in the core-I stream must be propagated to other streams for structure consistency. The algorithm that we present first identifies the set of unique core-Is along with their frequencies. This set represents the symbol alphabet  $\mathcal{A}$  for the input stream to PPM. The input to PPM is a vector of  $N$  symbols  $x \in \{\mathcal{A}\}^N$ . A pair of instructions,  $x_i x_j$ , denotes that  $x_i$  precedes  $x_j$  in the input vector. 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  $x_i$  must be scheduled within a basic block before instruction  $x_j$ . The goal of IR is to create a permutation  $\pi(x)$  such that:

- $\forall x_a x_b \in \pi(x) \ d(b, a) \notin D(x)$ , i.e. no dependency is violated, and
- a PPM prediction engine of order  $K$  emits minimal number of *escape* symbols while compressing  $\pi(x)$ .

The algorithm aims at building a solution that improves PPM's context matching both locally and globally by sorting instructions within a basic block. Instructions are sorted by their binary codes of their corresponding core-I. Thus, 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 block, the likelihood of core-Is with a higher binary code increases. Hence, after sorting, a pair of instructions is more likely to appear in the same order throughout the binary.

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

a) Schedule before ordering

b) Schedule after ordering

Figure 2: An example of IR within a basic block. Instructions are sorted by their core-Is (bold) under the constraint of dependencies among instructions.

Consider the example shown in Figure 2. The first two instructions do not have dependencies. Thus, sorting switches their positions in the input stream. Since there could not possibly exist dependency between instructions with such core-Is, in any state of the PPM model, **8B 4C 24** cannot exist in the context of **8B 50**. This means that the instructions that do appear in the context of **8B 50** are predicted with higher probability, and consequently with fewer bits.

Another consequence of sorting instructions under the constraint of their mutual dependencies, is that instruction pairs  $x_i x_j$ , where  $\{x_i, x_j\} \in D(x)$  and  $bc(x_i) > bc(x_j)$ , are likely to be positioned subsequently.<sup>3</sup> For example, if  $bc(x_j)$  is relatively small, the sorting procedure would move  $x_j$  toward the beginning of the basic block. Similarly, the instruction  $x_i$  with relatively large  $bc(x_i)$  is moved toward the bottom of a basic block. However, because of  $\{x_i, x_j\} \in D(x)$ , when  $x_j$  is rescheduled immediately after  $x_i$ , the sorting procedure cannot swap their positions and their adjacency remains permanent. For our benchmark, after IR for sorting, the number of instruction pairs  $x_i x_j$  with  $\{x_i, x_j\} \in D(x)$  and  $bc(x_i) > bc(x_j)$

<sup>3</sup>Function  $bc(x)$  returns the binary code of the instruction  $x$ .

increased between 15 and 20%. An example of such cases are reschedule of **8B F3** next to **89 70 08**, and **3B CA** next to **8B 50 04**.

The sorting procedure used in IR slightly differs from standard sorting procedures. The algorithm performs several passes through a basic block to find its ascending<sup>4</sup> order under the constraint of mutual instruction dependencies. For each pass, the sorting algorithm processes the instructions in their current order, swapping subsequent pairs  $x_i x_{i+1}$  if  $bc(x_{i+1}) < bc(x_i)$ . Multi-passes are required because swapping  $x_i x_{i+1}$  to  $x_{i+1} x_i$  in one pass (creating a sequence  $x_{i-1} x_{i+1} x_i$ ), may enable swapping  $x_{i-1} x_{i+1}$  to  $x_{i+1} x_{i-1}$  in the subsequent pass. The final pass is the one where no instructions satisfy the criteria to be swapped. The maximum number of passes for a single block equals the total number of instructions in the basic block.

### 2.3 Dual Alphabet PPM

Using all unique core-Is as an alphabet of input symbols for PPM may drastically increase the size of the PPM model because there are commonly several thousand unique core-Is per application in our benchmark set. Before we propose our solution to this problem, we make the following observation for x86 core-Is: *regardless of program size<sup>5</sup> and 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 3 illustrates how the coverage (y-axis) of program's core-Is increases as the x-axis most frequent unique core-Is are considered. For a set of functionally different applications such as Microsoft Office and a compiler, the observation holds as true.

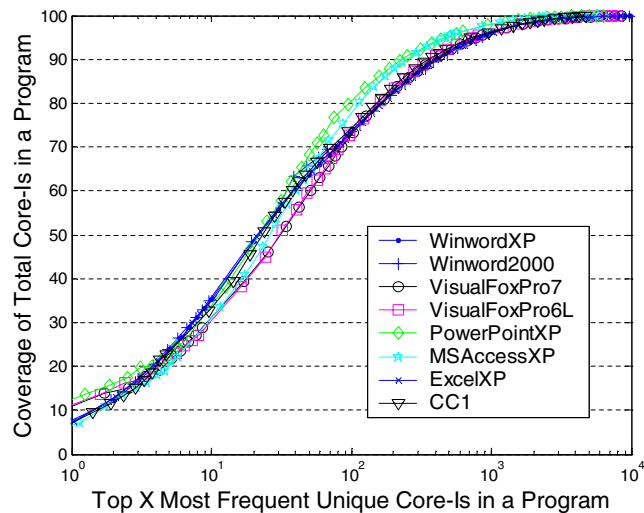


Figure 3: Percentage of all Core-Is of a program (y-axis) covered with a set of x-axis most frequent unique core-Is in the program.

We explore this observation in the following way. We introduce a dual alphabet PPM as a technique to improve the traditional PPM model while processing a program binary. The considered alphabets are:

- a higher priority alphabet  $\mathcal{B}$  - symbol set of  $L_1$  most frequent core-Is (typically  $L_1 = 256$ ), where core-I is a variable-length symbol typically 1 to 4 bytes long, and
- the standard alphabet  $\mathcal{A}$  - symbol set of all  $L_2 = 256$  8-bit values.

<sup>4</sup>Sorting core-Is in the descending order of their binary codes yields comparative CR.

<sup>5</sup>The observation targets only large programs.

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  $\mathcal{B}$ -symbol. Otherwise, the extracted  $m$ -byte core-I is split into  $m$  8-bit  $\mathcal{A}$ -symbols, where each of these symbols is individually fed to the PPM model. It is important to stress that there are core-I's that are 8-bit 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 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 4 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.

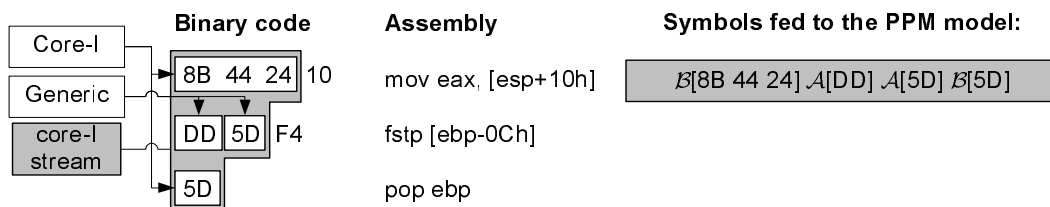


Figure 4: An example of how a disassembler parses the input stream of core-I's.

An additional advantage of the dual alphabet PPM model is that it reduces the number of symbols processed. The improvements in the compression rate with respect to traditional PPM, come at the expense of approximately doubling the size of the model (empirically determined). More importantly, a larger model means that saturating its statistics requires processing more input data. Since our algorithm targets large applications, the described disadvantages are of minor importance.

### 3 Random Access Decompression

The developed compression algorithm targets software delivery (SD) systems. The key feature of a compression algorithm for SD 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.

In a typical SD environment, a client is linked to a SD server through 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 - its *code-base*. Then, as the client invokes functions not yet downloaded, service requests are sent to the SD server. A service request is typically an address offset within the binary. The SD server replies with a block of data containing the target address. The distributed blocks may represent procedures or groups of procedures (i.e. funclets<sup>6</sup>). Upon reception, the client stores the received funclet and executes the invoked procedure. A system like this enables richer software

<sup>6</sup>In the remainder of the paper, we assume that variable-size funclets are distributed through the SD link.

pricing and metering models, greatly facilitates software updates and bug fixes, and improves software integrity and availability.

To address the decompression requirements of an SD platform, we propose three different modifications to the original PPM paradigm. 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 5 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.

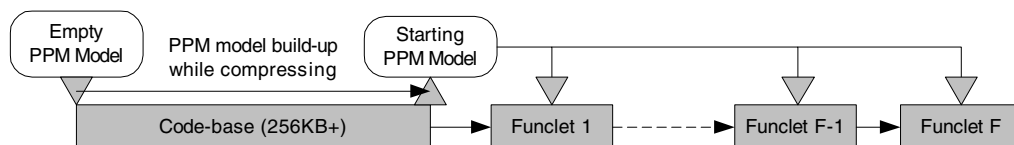


Figure 5: *Funclet compression for random access.*

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 2: *Relative overhead in compression rate of different random access enabling methods (CM - CM/UM and SM) for variable funclet and code-base size. Results obtained using the core-I stream of the CC1 compiler.*

- **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 destroyed afterwards.
- **UNDO MODEL (UM).** This technique greatly improves upon the performance 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 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 performance 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 rate under the assumption that the model has been statistically saturated to a level which enables near-maximum performance.

Experimental data presented in Table 2 quantifies the impact of CM/UM<sup>7</sup> and SM on compression rate as code-base and funclet size vary. The compressed data is the core-I stream of the CC1 compiler. Note, that for the CM/UM method, code-base and funclet size of only

<sup>7</sup>CM and UM have equivalent CRs, but different decompression run-times.

256KB and 32KB respectively, yields a compression rate that is within the variance of the compression rate 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 2 clearly demonstrate that saturation of the PPM model is a factor far more important than funcelet size as compression rate slightly varies with this parameter. Finally, the improvement in decompression throughput is two orders higher for SM with respect to CM/UM at the cost of  $\sim 2\%$  increase in compression rate, practically leading to a conclusion that with little overhead, SM can effortlessly feed a CPU with software even from highest bandwidth Internet connections. UM brings a 5-10% increase in decompression throughput with respect to CM.

Program Name	Uncompressed File Size	PPMD		PPMexe		Improvement [%]
		Size	CR	Size	CR	
Compiler CC1	872588	416218	0.477	347964	0.3988	16.4
Winword2000	6137358	3225381	0.5255	2658102	0.4331	17.6
WinwordXP	7535800	3919572	0.5201	3217001	0.4269	17.9
ExcelXP	7303411	3807300	0.5213	3127621	0.4282	17.85
PowerPointXP	4449093	2140822	0.4812	1658045	0.3727	22.6
MSAccess	3942060	1919666	0.487	1560923	0.3960	18.7
VisualFoxPro6L	3769781	1976977	0.5244	1553961	0.4122	21.4
VisualFoxPro7	3941408	2102326	0.5334	1666827	0.4229	20.7

Table 3: *Compression rate comparison for a benchmark of large applications: PPMD vs. our technology PPMexe. File-size reported in bytes. Our technology used  $M = 7$  different streams, PPM model of order  $K = 4$  with  $L_1 = L_2 = 256$  symbols for each alphabet.*

## 4 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 rates significantly lower than the remainder of a program binary (less than 0.25). The results are presented in Table 3. PPMexe has outperformed PPMD on the average for 19.1%. SPLIT<sup>2</sup>-STREAM, INSTRUCTION RESCHEDULING, and DUAL ALPHABET PPM contributed to the improvement approximately with a 2:1:1 ratio. 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. Needless to say, using the SM variant of PPMexe for random access decompression, the decompression throughput is increased nearly 2 orders of magnitude.

## 5 Conclusion

In this work, we have explored PPM-based compression algorithms that focus exclusively on program binaries. Our algorithm, PPMexe uses several pre-processing steps to PPM as well as a modification to the generic PPM technology to significantly improve upon the compression rates exhibited by the best PPM variants. The key mechanism for enabling dynamic decompression of program binaries is random-access. We have introduced several techniques that

enable this feature with different effects on the decompression speed vs. compression ratio trade-off. We have demonstrated the effectiveness of the developed techniques by building a compression codec for x86 binaries. Binaries compressed using PPMexe were 16-23% smaller than files created using off-the-shelf PPMD.

## References

- [1] J. Gilchrist. The Archive Compression Test. Available on-line at <http://compression.ca>.
- [2] C. Bloom. PPMZ. 1995. <http://www.cco.caltech.edu/~bloom/src/ppmz.zip>.
- [3] J.G. Cleary and I.H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, vol.32, no.4, pp.396–402, 1984.
- [4] J.G. Cleary, W.J. Teahan, and I.H. Witten. Unbounded length contexts for PPM. *Data Compression Conference*, 1995.
- [5] S. Debray, et al. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, vol.22, no.2, pp.378–415, 2000.
- [6] J. Ernst, et al. Code compression. *Programming Languages Design and Implementation*, pp.358–365, June 1997.
- [7] C. W. Fraser. Automatic inference of models for statistical code compression. *Programming Languages Design and Implementation*, pp.242–246, 1999.
- [8] I. Hong, et al. Potential-Driven Statistical Ordering of Transformations. *Design Automation Conference*, pp. 347–52, 1997.
- [9] P.G. Howard and J.S. Vitter. Design and analysis of fast text compression based on quasi-arithmetic coding. *Data Compression Conference*, pp.98–107, 1993.
- [10] Intel Architecture Software Developer’s Manual, Volume 2: Instruction Set Reference Manual, <http://developer.intel.com/design/processor/>
- [11] H. Lekatsas and W. Wolf. Random access decompression using binary arithmetic coding. *Data Compression Conference*, pp.306–315, 1999.
- [12] S. Lucco. Split-stream dictionary program compression. In *Programming Languages Design and Implementation*, pp.27–34, 2000.
- [13] A. Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, vol.38, no.11, pp.1917–21, 1990.