Data Hiding in Compiled Program Binaries for Enhancing Computer System Performance

Ashwin Swaminathan[1], Yinian Mao[1], Min Wu[1] and Krishnan Kailas[2]

[1] Department of ECE, University of Maryland, College Park, MD 20742, USA
{ashwins, ymao, minwu}@eng.umd.edu *
[2] IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA
krish@watson.ibm.com

Abstract. Information hiding has been studied in many security applications such as authentication, copyright management and digital forensics. In this work, we introduce a new application where successful information hiding in compiled program binaries could bring system-wide performance improvements. Our goal is to enhance computer system performance by providing additional information to the processor, *without* changing the instruction set architecture. We first analyze the statistics of typical programs to demonstrate the feasibility of hiding data in them. We then propose several techniques to hide a large amount of data in the operand fields with very low computation and storage requirements during the extraction process. The data embedding is made reversible to recover the original instructions and to ensure the correct execution of the computer program. Our experiments on the SPEC CPU2000 benchmark programs show that up to 110K bits of information can be embedded in large programs with as little as 3K bits of additional run-time memory in the form of a simple look-up table.

1 Introduction

The machine instructions in a compiled computer program, as specified by the Instruction Set Architecture (ISA) of a processor [1], are the primary means for exchanging information between the programmer and the computer hardware. An instruction set consisting of fixed width instructions is one of the key aspects of the reduced instruction set computing (RISC) architecture principles employed by several modern processors [1, 2]. This is because fixed width instructions are easy to fetch, decode and execute; thus greatly help simplify the fetch and decode stages of processor pipeline. However, the fixed width RISC instruction sets make it difficult to expand the instruction encoding space in the future for adding more information to the existing instructions, or for adding more instructions to an existing ISA.

It has been shown by several microarchitecture studies in the past that if a small amount of side information could be added to instructions, it would

* This work was supported in part by the U.S. National Science Foundation under CAREER Award CCR-0133704.

help improve the performance of processors. For example, the accuracy of data value prediction techniques can be greatly improved if one could embed an extra bit of information to each instruction of interest. This would help to classify them based on the predictability criteria, resulting in improved instruction-level parallelism opportunity as well as better utilization of prediction tables [3]. In the multiprocessor systems, it has been shown that more relaxed memory consistency models can be easily supported for obtaining better performance on multiprocessor workloads if one can classify the load/store instructions with an extra bit of information [4]. In general, even though adding 1-2 extra bits of information to existing instructions could improve performance or simplify the hardware structures, it is not practical to re-design the instruction set architecture (ISA). This is mainly because changing the ISA is a major effort, and it cannot easily support backward compatibility – making it difficult for older versions of binary programs to run on the new versions of the processor. It is clear from the above discussion that embedding additional information to the instructions of a program binary without modifying the ISA has the potential for providing system-wide performance improvements.

In this work, we investigate the feasibility and techniques to store and extract additional information for computer programs in an ISA-independent way and without inserting extra instructions. We also study the cost for such data embedding and extraction, which can be quantified in several aspects including the amount of computation and storage needed during embedding and extraction. Our contribution in this paper is three fold. First, we show that such invertible embedding is possible for most programs. Second, we propose algorithms to find embeddable program segments and introduce schemes that can achieve data embedding and extraction transparent to the program execution. Third, we optimize the embedding/extraction algorithms under stringent practical constraints in terms of computation complexity and storage limitations.

This paper is organized as follows. In Section 2, we discuss the proposed data hiding framework. The details of the proposed algorithms are explained in Section 3 along with their simulation results. An improvement to the proposed schemes to reduce the memory overhead is examined in Section 4 and the related works are discussed in Section 5. The final conclusions are drawn in Section 6.

2 Challenges, Feasibility, and the Proposed Framework

In this section, we examine the constraints and challenges in hiding data in program binaries, in order for the processor to take advantage of the hidden data to enhance the execution performance. We then discuss the feasibility of hiding data in program binaries and present the proposed data hiding framework to meet these stringent constraints.

2.1 Challenges and Constraints

Embedding data into program binaries is a challenging task because of the numerous stringent constraints at the decoding side in the Central Processing Unit

(CPU). Given the high sophistication and performance requirement in modern processor design, the extra on-chip logic and memory have to be kept minimum for extracting the hidden data in parallel with the program execution. It is also not desirable to introduce extra instructions in the program for data hiding, as it consumes more cycles and slows down the program execution. Compared with data extraction, the data embedding process can be performed off-line and only once for a program. Thus, we can afford relatively more computation and memory resource during data embedding. Another important constraint in hiding data in program binaries is that the data hiding scheme should not interfere with the normal program execution. This requires the data embedding be reversible (or lossless), so that both the original program (host data) and the embedded data can be recovered at the decoder side without any errors.

Reversible data hiding has been studied in the context of multimedia data [5, 6]. Typically, the host data is first losslessly compressed and the extra information is then appended to the compressed host data. At the decoder side, the appended extra information is read out and the host data is recovered using a lossless decoding method. The main challenge in compressing a program binary without interfering normal program execution arises from the fact that the order of execution of all instructions in a program can be unpredictable. Due to the data dependent control flows and branches in a program, instructions are not always executed in the same sequence as they appear in the program binary. The instruction execution order is often dynamically determined at the program execution time and some instructions might not be executed at all. Additionally, all modern processors use speculative execution of instructions [1]. The most common form of control flow speculation involves executing instructions after a branch (along a predicted path) before the branch instruction completes its execution. When the branch condition is resolved and a mis-prediction is indicated, the processor will have to roll-back to the branch. As such, the data embedding scheme cannot presume a certain execution order to use the previous instruction(s) to hide information for subsequent instruction(s). Hence it is very difficult to employ common lossless data compression schemes such as the Lempel-Ziv and the arithmetic coding [7], where the decoding result depends on the preceding codes. In other variable-length coding schemes, such as the Huffman coding [7], the length of some codewords that appear infrequently may well exceed the length of an instruction (32 bits) and would not suit our purpose. These common compression schemes also require a non-trivial amount of static or run-time memory, which is expensive to accommodate in the CPU design.

To satisfy the stringent computation and memory constraints on the processor side and the considerations discussed above, we choose to use individual instruction as the basic data embedding unit as opposed to using a block of code segments. We also choose to use fixed-length compression techniques through simple table lookup operations as opposed to variable-length ones. Next, we discuss the feasibility of hiding data in program binaries and present the proposed data hiding framework.

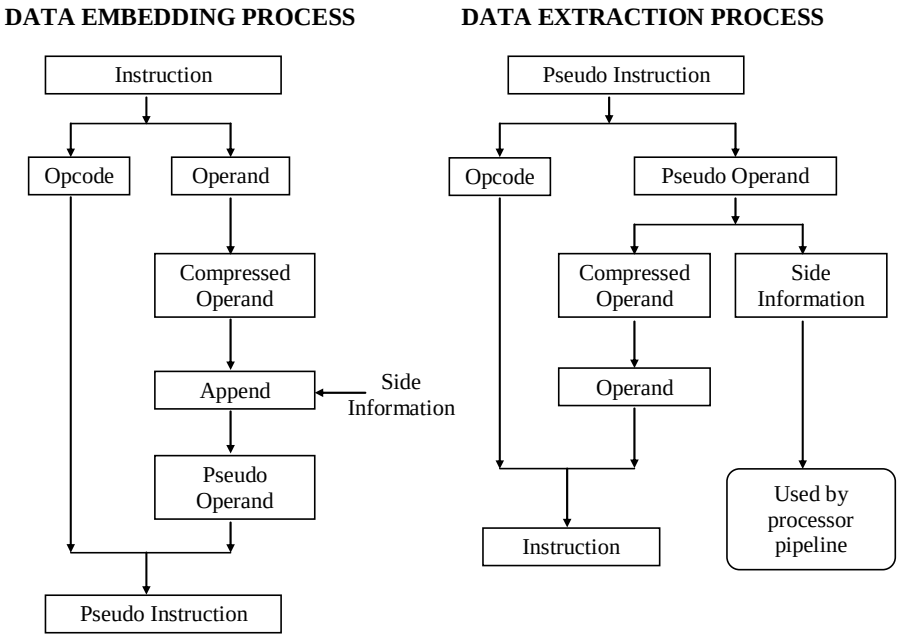

Fig. 1. A lossless data hiding framework for program binaries

2.2 A Lossless Data Hiding Framework for Program Binaries

We use IBM PowerPC ISA as an example to demonstrate the feasibility of hiding data in program binaries. The principle of our scheme can be extended to other ISAs. The IBM PowerPC ISA is a typical RISC ISA with fixed-width instruction encoding [8]. A typical PowerPC instruction contains 4 bytes of information, including 6 bits for the opcode (which indicates the type of operation) and the remaining 26 bits for the operand (which indicates the parameters to use). The 26-bit operand suggests a total of 2^{26} possible combinations. While using 26 bits to represent these combinations makes it easy for the computer hardware to process, information theory suggests that it is possible to use fewer than 26 bits if the 2^{26} combinations are not equally likely to appear [7].

To understand the distribution of the operand combinations in computer programs, we examine the SPEC CPU2000 benchmark programs [9]. SPEC CPU2000 are a collection of representative programs widely used in computer system research. A brief description of the SPEC programs can be found in the Appendix of this paper. By analyzing the operands of these representative programs, we observe that only a very small portion of the 2^{26} combinations in the operand field actually appear in a program. The number of unique combinations of the operands that occur at least once in each of the SPEC programs are shown in Table 1. The table shows that less than one percent of the 2^{26} possible combinations are used in these programs. This important observation suggests that

the operand information can be represented in fewer than 26 bits, and there is a considerable amount of room to compress the operands and hide data.

As a general framework for reversible data hiding in program binaries, we first losslessly compress the compiled program binaries and embed the side information into them. The details of the embedding and the extraction processes are shown in Fig. 1. Since each instruction in the program binary usually contains a short opcode and a longer operand field, we focus on the operand field and compress them in a lossless manner to represent them in fewer bits. We then use the resulting space to store the desired extra information. The data extraction is performed during program execution. In this stage, the operand field is decompressed and side information is extracted. The opcode along with the decompressed operand field is executed the normal way. The extracted side information may be used by the processor pipeline for a variety of applications.

In principle, if X distinct combinations appear in the operand field, we can represent the operands uniquely in $y = \lceil \log_2(X) \rceil$ bits with a one-to-one reversible mapping. The mapping table can be stored in program header and used to establish a look-up table (LUT) in the data extractor, as long as its size is very small compared to the number of bits gained. The $(26 - y)$ bits per instruction that are obtained in this process can be used to store the side information. At the data extraction side, the y-bit compressed operand is replaced by the matching 26-bit operand found in the LUT. Once the hidden data is extracted and the original instruction is recovered, the extracted side information can be utilized without affecting the normal program execution.

For large programs such as the CC1 (GNU C++ compiler), we have $X = 86355$ and $y = 17$ from Table 1, suggesting that we would save $26 - y = 9$ bits per instruction to store the side information. However, if we build a mapping table for these 86355 instructions, its size would be at least $X \times 26 = 2245230$ bits. Clearly, this is a large overhead for gaining 9-bit side information per instruction. Moreover, it is difficult to store such a large table in the on-chip memory for fast access. Therefore, instead of trying to embed many bits into each instruction, we consider a more practical objective, namely, to embed only one or a few bits into each instruction using as small LUT as possible. In the next section, we will discuss how to design data hiding schemes that can achieve fast and resource-efficient data extraction.

3 The Proposed Data Hiding Algorithms

Computer architecture research has suggested that even one-bit side information per instruction would be very advantageous to enhance system performance. As such, we focus on designing efficient techniques to embed one bit of extra information per instruction. These techniques can easily be extended to embed more than one bit per instruction at an added cost of a larger LUT.

Considering the challenges and constrains posed in Section 2.1, we propose the following practical method for embedding information into a program. We search for the smallest n so that there exists a subset of n operand bits (out of

Table 1. Statistics of SPEC programs and the data hiding results by applying the Exhaustive Search algorithm to all instructions.

Program Name	Total # of instructions in the program (N)	# of distinct combinations appearing in operand field (X)	n	Memory for LUT (S bits)	Relative overhead (η) Memory req. / Data Hidden
SWIM	2937	1800	9	2330	79.33 %
ART	5985	2564	9	2330	38.93 %
WUPWISE	8218	3807	9	2330	28.35 %
EQUAKE	9589	4500	10	5146	53.66 %
LUCAS	12449	6430	10	5146	41.34 %
APPLU	15936	8992	10	5146	32.29 %
MCF	18351	6738	10	5146	28.04 %
FACEREC	19044	9015	10	5146	27.02 %
GZIP	28727	10443	11	11290	39.30 %
BZIP2	28632	9838	11	11290	39.43 %
APSI	43244	18538	12	24602	56.89 %
AMMP	46346	15682	12	24602	53.08 %
GALGEL	57971	19990	12	24602	42.44 %
PARSER	62807	15194	11	11290	17.98 %
CRAFTY	72486	23876	11	11290	15.58 %
TWOLF	85981	23876	12	24602	28.61 %
EON	121012	23787	11	11290	9.33 %
MGRID	151202	31492	13	53274	35.23 %
VORTEX	167056	29951	11	11290	6.76 %
VPR	182039	36942	13	53274	29.27 %
PERLBMK	192898	35929	12	24602	12.75 %
MESA	209986	39538	13	53274	25.37 %
GAP	220308	39085	13	53274	24.18 %
FMA3D	235383	80849	13	53274	22.63 %
SIXTRACK	360292	84514	14	114714	31.84 %
CC1	571820	86355	14	114714	20.06 %

the 26 bits), for which no more than 2^{n-1} distinct operand combinations have appeared in the program. We can then losslessly represent this subset of n bits using $(n-1)$ bits and thus provide one bit per instruction for data hiding. To facilitate the discussion, we say that the operands set has a *Negative Redundancy* of n in this case. A lower value of negative redundancy implies more redundancy in the program, and in turn a smaller LUT for data embedding and extraction.

As far as data extraction is concerned, if we have M combinations of n-bit pattern appearing, we would require a LUT of size $M \times n$ for the inverse mapping. Additionally, we would require up to 26 bits to specify the subset of bits involved in the mapping. Thus, the total cost in terms of memory usage is upper bounded by $(Mn+26)$ bits. As mentioned earlier, this LUT can be stored in the program header, and will be loaded into run-time memory to initialize the data extractor at the beginning of program execution. To reduce the overhead in program storage and in run-time memory, we would prefer this LUT to be as

small as possible. The data embedding can be performed by searching through the LUT, and the computation and memory requirement for embedding are much less stringent than for extraction. These considerations lead us to propose the following encoding and decoding algorithms.

3.1 Exhaustive Search Algorithm

As indicated before, the size of the look-up table (S) is upper bounded by ($Mn+26$) bits where $M \leq 2^{n-1}$. To minimize the memory and computation overhead at the decoding end, we would like to find the smallest subset of bits for which we start to see at most 2^{n-1} distinct combinations appear. We denote this optimal value of n by n_{opt} and the optimal subset that gives this mapping by Ω_{opt}. One way to find n_{opt} and Ω_{opt} is by exhaustively searching over every possible subset. The worst case complexity of the search over the 26-bit operand is $\binom{26}{1} + \binom{26}{2} + \cdots + \binom{26}{26} = 2^{26} - 1$ on the embedder side. The *Exhaustive Search* algorithm is described in Algorithm 1.

Algorithm 1 Exhaustive Search Algorithm

```
Input: Compiled Static Instruction file
Output: Possible Mapping
for n = 1, 2,..., 26 do
  Initialize: Γ - set of all possible n-bit combinations
  for k = 1, 2, ..., (26 choose n) do
    Choose as P, the k^th n-tuple in Γ.
    Count the number of distinct combinations that appear in P. Call it X
    if X ≤ 2^(n-1) then
      {One possibility found}
      return Obtained Position (P)
    end if
  end for
end for
```

The exhaustive search algorithm for finding the bit positions was tested on the standard SPEC CPU2000 benchmark program suite [9]. In this experiment, we consider all the instructions in a program and try to find a bit-position combination that can be used for embedding. The experiment results are summarized in Table 1. We list the number of instructions (N) in the program (excluding all the relocatable instructions that will be modified by the OS loader); the number of unique combinations of the bit positions, out of the possible 2^{26}, in the operand fields (X); the negative redundancy of the operand bits (n); the memory required to store the inverse mapping table (S); and its ratio (η) with respect to the amount of data hidden (D), where D is equal to the number of instructions with one hidden bit per instruction. From the results, we observe that the negative redundancy n is usually in the range of 9 to 15 and therefore the size of the

LUT, S, is not small. Moreover, the ratio η is above 20% in most cases and this indicates a relatively high overhead in obtaining data hiding payload. Another disadvantage for the exhaustive search is the high computational complexity in finding the bit positions. However, it provides a basis for comparison with other search algorithms.

3.2 Consecutive Search Algorithm

The exhaustive search algorithm has two problems - large memory requirement in data extraction and high computational complexity in data embedding. To address these problems, we introduce the consecutive search algorithm. We also observe through experiments that n can be greatly reduced if we choose to embed data in only a subset of instructions (e.g. load/store instructions) that appear frequently in static programs.

To speed up the search for the bit positions, we propose a modified approach to find the sub-optimal value of n by considering only consecutive bit positions in the search. This would speed up the encoding process exponentially. The number of iterations required to find a mapping reduces from $\binom{26}{n}$ to $(26 - n) + 1$ for a particular n. We denote this sub-optimal value of n by $n_{sopt}^{(c)}$ to reflect the positions being consecutive. While we have $n_{sopt}^{(c)} \geq n_{opt}$, our experiments show that it is close to the optimal solution in most cases. The *Consecutive Search* algorithm is described in detail in Algorithm 2.

This method of choosing consecutive bit positions was tested with the SPEC CPU2000 benchmark suite [9]. In this experiment, we selected the Load/Store instructions for data hiding, as the memory access are often the performance bottleneck in program execution [1]. For comparison, the results for the exhaustive and consecutive search are shown in Table 2. We note that in most programs, about one third of the instructions are load/store instructions. Therefore, there is still a substantial amount of space for hiding data. From the table we can see that n_{opt} is usually around 5–9, while $n_{sopt}^{(c)}$ is greater than n_{opt} by 1 or 2 bits; and the total memory required for data extraction is usually no more than 2500 bits. Furthermore, the computation complexity is greatly reduced when consecutive search is employed. We also observe that by restricting our data hiding scheme to only load/store instructions, the ratio $\eta = S/D$ is only around 5%, which indicates that we can hide more data for a fixed amount of memory usage.

3.3 Iterative Search Algorithm

In this part, we introduce the Iterative Search algorithm to mitigate the disadvantages of both the exhaustive search and the consecutive search algorithms discussed before. The Iterative Search algorithm is based on the observation shown in Table 3, that the exhaustive and consecutive search algorithms often produce bit position subsets that have a large overlap. So we first run the Consecutive Search algorithm to find an initial guess for the solution with a negative

Algorithm 2 Consecutive Search Algorithm

Input: Compiled Static Instruction file
Output: Possible Mapping
for $n = 1, 2, \ldots, 26$ **do**
  Initialize:
  Γ - set of all possible n-bit combinations, where $|\Gamma| = 27 - n$
  $\Gamma = \{(1, 2, \ldots, n), (2, 3, \ldots, n+1), \ldots, (26 - n + 1, \ldots, 26)\}$
  for k = 1, 2, ..., $26 - n + 1$ **do**
    Choose as P, the k^{th} n-tuple in Γ
    Count the number of distinct combinations that appear in P. Call it X
    if $X \leq 2^{n-1}$ **then**
      {*One possibility found*}
      return *Obtained Position* (P)
    end if
  end for
end for

redundancy of $n_{sopt}^{(c)}$, and then proceed with an iterative algorithm to find the optimal solution.

Suppose the solution in the i-th iteration has a negative redundancy of r and the set of bit positions is P_i. In the $(i+1)$-th iteration, we use P_i to find a solution of negative redundancy $r-1$ by a systematic search. In this search, we form a new set of bit positions by choosing j ($j = r-1, r-2, \ldots, 1, 0$) positions out of the solution P_i and remaining $(r-1-j)$ positions from the set $\{1, 2, \ldots, 26\} - P_i$. The basic idea behind this ordering is the resemblance of the final optimal solution (obtained using exhaustive search) and the sub-optimal solution (obtained using the consecutive search), as indicated by Table 3. We then check if this set of bit positions is a possible solution. We initially start our iteration from $j = r-1$ and proceed to lower values of j. If we are able to find a mapping of negative redundancy $(r-1)$, we update P_{i+1} to the set of new bit positions and proceed on to the next iteration. It is to be noted that if P_{i+1} is a solution, then adding any extra bit to P_{i+1} set still remains a solution. Therefore, if we are not able to find any mapping of negative redundancy $(r-1)$, we conclude that there is no mapping with a negative redundancy less than r and declare P_i to be the optimal solution. The details are presented in Algorithm 3.

We note that the Iterative Search algorithm in the worst case corresponds to the Exhaustive Search algorithm and in the best case would correspond to the Consecutive Search algorithm in terms of computational requirements. By this ordered search, we can expect to reach the optimal solution n_{opt} in fewer iterations than the exhaustive search.

4 Improved Data Hiding through Program Partitioning

In this section, we reduce the storage overhead of the basic data hiding algorithm introduced in the previous section by program partitioning. We divide the main

Table 2. Data hiding results of the Consecutive and Exhaustive Search algorithms on Load/Store instructions. One bit is embedded in each load/store instruction.

Program Name	Total # instr. in the prog. (N)	# of embeddable Load/Store instructions (N_{ls})	# of unique combinations of Ld/St instr. appearing in operands(X_{ls})	n_{opt}	$n_{sopt}^{(c)}$	LUT size (bits)	Overhead *(sub-opt)* $\frac{mem.req.}{datahidden} \times 100\%$	Overhead *(opt)* $\frac{mem.req.}{datahidden} \times 100\%$
SWIM	2937	854	515	4	5	58	12.41 %	6.79 %
ART	5985	1611	544	5	6	106	13.53 %	6.57 %
WUPWISE	8218	2682	1255	5	5	106	3.95 %	3.95 %
EQUAKE	9589	3581	1526	5	6	106	6.09 %	2.96 %
LUCAS	12449	3755	2059	5	5	106	2.82 %	2.82 %
APPLU	15936	5846	3563	7	7	474	8.11 %	8.11 %
MCF	18351	5272	1976	5	7	106	8.99 %	2.01 %
FACEREC	19044	6215	3506	4	5	58	1.71 %	0.93 %
GZIP	28727	7546	2627	5	8	106	13.91 %	1.40 %
BZIP2	28632	7604	2163	6	8	218	13.80 %	2.87 %
APSI	43244	16380	6429	7	7	474	2.89 %	2.89 %
AMMP	46346	15338	4830	6	8	474	6.84 %	3.09 %
GALGEL	57971	20062	7405	8	8	1050	5.23 %	5.23 %
PARSER	62807	17182	3482	5	8	106	6.11 %	0.61 %
CRAFTY	72486	18213	5710	5	8	106	5.76 %	0.58 %
TWOLF	85981	26965	6204	6	8	218	3.89 %	0.81 %
EON	121012	43185	7646	7	8	474	2.43 %	1.09 %
MGRID	151202	39662	7299	9	10	2330	12.98 %	5.87 %
VORTEX	167065	43696	5346	5	8	106	2.40 %	0.24 %
VPR	182039	42880	7238	9	10	2330	12.00 %	5.43 %
PERLBMK	192898	57535	5733	7	8	474	1.82 %	0.82 %
MESA	209986	67305	14037	9	9	2330	3.46 %	3.46 %
GAP	220308	58038	6855	7	9	474	4.01 %	0.81 %
FMA3D	235383	98215	41437	8	8	1050	1.07 %	1.07 %
SIXTRACK	360292	101848	26804	11	11	11290	11.08 %	11.08 %
CC1	571820	138792	13373	9	10	2330	3.70 %	1.67 %

program into several parts and find a mapping table for each part. Each table would have smaller size than without the partitioning. These tables can be stored together in the program header and loaded to the on-chip memory sequentially during program execution. The storage overhead in the static program is the total size of all LUTs, but the run-time memory overhead is determined only by the size of the largest LUT.

We use the program *SIXTRACK* to illustrate this principle. From Table 2, we see that the size of the LUT for *SIXTRACK* is more than 11K bits without program partitioning. When we split the program into several segments and embed data into each segment, we can reduce the mapping table size by a factor of two to four. These results are shown in Table 4. Similar experiments were conducted on some other SPEC program files to find out the minimum number

Table 3. Operand positions obtained for data hiding using the Exhaustive Search and the Consecutive Search algorithms

	Exhaustive Search		Consecutive Search	
Program Name	n_{opt}	operand bits selected	$n_{sopt}^{(c)}$	operand bits selected
MCF	5	(1,2,11,12,13)	7	(10,11,12,13,14,15,16)
PARSER	5	(2,3,11,12,13)	8	(6,7,8,9,10,11,12,13)
TWOLF	6	(1,2,3,11,12,13)	8	(6,7,8,9,10,11,12,13)
EON	7	(2,3,7,8,11,12,13)	8	(7,8,9,10,11,12,13,14)
VORTEX	5	(1,2,11,12,13)	8	(7,8,9,10,11,12,13,14)

Algorithm 3 Iterative Search Algorithm

Input: Compiled Static Instruction file (F)
Output: Possible Mapping
Initialize: P_1 = *Consecutive_Search(F)*
$n_{sopt}^{(c)} = |P_1|$
for i = 1, 2, ..., 26 **do**
  {*Use P_i to find P_{i+1} of a lower negative redundancy*}
  Γ_i = *Generate_Ordered_Positions (P_i)*
  for k = 1, 2, ..., $|\Gamma_i|$ **do**
    Choose as p_k, the k^{th} (n-1)-tuple in Γ_i
    Count the number of distinct combinations that appear in p_k. Call it X
    if $X \leq 2^{n-2}$ **then**
      {*One possibility found*}
      $P_{i+1} \leftarrow p_k$
      break **for loop** and *goto* **flag:**
    end if
  end for
  if $X > 2^{n-2}$ for all positions in Γ_i **then**
    {P_i *is the best solution*}
    return P_i
  end if
  flag:
end for

Generate_Ordered_Positions
Input: Initial Guess (p)
Output: Ordered set (Γ) to run search
Initialize: $n = |p| - 1$
for $j = n, n-1, \ldots, 0$ **do**
  Obtain a subset p_1 by choosing j components from the vector p
  Obtain a subset p_2 by choosing $n-j$ components from the vector $\{1, 2, \ldots, 26\} - p$
  Concatenate p_1 and p_2 to form a search vector g
  Add the search vector g to the set Γ
end for
return Γ

Table 4. Data hiding results using program partitions for the *SIXTRACK* program

	Negative Redundancy	Total # bits hidden	Memory overhead (LUT size in bits)
(a) FULL PROCESSING	11	101848	11290
(b) BLOCK PROCESSING - 3 BLOCKS			
First Set (100K instructions)	6	27005	218
Second Set (100K instructions)	9	27155	2330
Third set (160K instructions)	9	47688	2330
Total		101848	4878
(c) BLOCK PROCESSING - 4 BLOCKS			
First Set (100K instructions)	6	27005	218
Second Set (90K instructions)	8	24722	1050
Third set (90K instructions)	7	26677	474
Fourth set (80K instructions)	8	23444	1050
Total		101848	2792

Table 5. Results on program partition for selected SPEC programs: showing here are the number of partitions required to limit the size of each LUT to be less than a given value S.

Program Name	$S = 1$ Kbits	$S = 12$ Kbits	$S = 24$ Kbits
SWIM	2	1	1
APPLU	5	1	1
APSI	11	4	1
GALGEL	14	6	3
TWOLF	20	5	2
VORTEX	37	8	1
PERLBMK	42	10	2
MESA	45	12	2
GAP	47	14	2

of partitions required to limit the size of each LUT to be less than a given value S. Table 5 presents the results for three different values of S. We can see that program partitioning can help reduce the size of each LUT to a manageable extent. This is achieved at the cost of reloading the corresponding LUT prior to the execution of each partition. Such cost can be reduced by carefully designing the partitions based on program flow models.

5 Related Work

In this paper, we investigate the possibility of data hiding in complied program binaries for enhancing system performance in RISC ISAs. The related prior art mostly falls in four main categories:

Steganography for program binaries has been studied in [18], where side information is inserted into a selected set of binary instructions by choosing one out of two (or more) different forms of the instruction that are functionally

identical. Such an embedding scheme requires an equal amount of computation both at the embedding side and at the decoding side. To achieve reversibility, the effective embedding payload will be substantially reduced.

In the computer architecture field, there are works on instruction abbreviation techniques for embedded DSPs. In [10], the authors present a technique for entropy bounded encoding of the ISA, where the primary concern is on variable size instructions which frequently occur in DSP architecture. In [11] and [12], the authors present an instruction set synthesis technique for configurable ASIPs and variable instruction set architectures. As these techniques require changes to the ISA, they cannot be applied in fixed-width RISC instruction sets.

Software watermarking techniques have been proposed for intellectual property protection. Early software watermarking schemes re-organize basic blocks in complied codes to embed a hidden mark [13]. Later, it was proposed to incorporate graph theoretical approaches in software watermarking [14]. In this case, the mark is embedded by inserting extra instructions and re-structuring existing instructions in a given program; and the watermark is formed by the control flow of the program. Dynamic path-based software watermarking was proposed in [15]. It uses the run-time trace of a program and a particular program input (the secret key) to carry hidden information. An analogous approach was proposed to watermark HDL code for ASIC and FPGA design [16]. All these schemes aim at preventing software piracy, where hostile adversaries have strong incentives to remove the embedded watermark. In our application of enhancing computer system performance, such adversarial environment does not exist and our focus is to provide side information to the processor at the lowest cost. In most software watermarking schemes, usually after watermark embedding, the number of instructions will be increased and the execution of the program will be slowed down. In contrast, our scheme aims at speeding up the program execution while maintaining the number of instructions.

In the field of multimedia signal processing, various techniques for reversible data hiding and lossless compression have been proposed for multimedia data [5, 6]. Some algorithms use additive spread spectrum techniques [17] and some others hide data by modifying selected features (such as the LSB) of the host signal [5]. These techniques cannot be directly extended for hiding data in program binaries because of the inherent differences in the host data. As discussed in Section 2.1, compression techniques such as the Lempel-Ziv and the arithmetic coding require the knowledge of the execution order of the instructions and are not suitable for our purpose. To our best knowledge, the current paper presents the first work that applies information hiding techniques to program binaries of fixed-width instruction set processors, whereby extra information is transparently embedded and can be extracted with very low cost by the processor to enhance computer system performance.

6 Conclusions

In this work, we have investigated data hiding in computer programs for transparently embedding additional information that may be used by the processor

for a variety of applications. We have shown that it is feasible to achieve efficient data hiding and fast data extraction using minimal additional logic and memory resources. We present a framework to achieve ISA-independent data hiding that is transparent to program execution. Under this framework, we introduce three algorithms to find bit positions in the operands of instructions that can be losslessly compressed to embed data. In addition, we propose improvement techniques through program partition to reduce the cost in data embedding and extraction. The effectiveness of our approaches are demonstrated through experimental results on the SPEC benchmark programs. Our experiments show that in most cases the proposed schemes can achieve linear time complexity in data embedding and require less than 3K bits of run-time memory overhead.

References

1. D. A. Patterson and J. L. Hennessy, *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1996.
2. G. Radin, "The 801 minicomputer," in *Symposium on Architectural Support for Programming Languages and Operating Systems (1st ASPLOS'82), Computer Architecture News*, (Palo Alto, CA), pp. 39–47, 1982.
3. F. Gabbay and A. Mendelson, "Can program profiling support value prediction?," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 270–280, IEEE Computer Society TC-MICRO and ACM SIGMICRO, 1997.
4. K. Gharachorloo, D. Lenoski, L. Laudon, P. Gibbons, A. Gupta, and H. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *Proceedings of the 17th Annual International Symposium on Computer Architecture, published in ACM SIGARCH*, vol. 18, pp. 15–26, May 1990.
5. J. Fridrich, M. Goljan, and R. Du, "Lossless data embedding - new paradigm in digital watermarking," *EURASIP Journal on Applied Signal Processing*, vol. 2002, no. 2, pp. 195–196, 2002.
6. M. U. Celik, G. Sharma, A. M. Tekalp, and E. Saber, "Reversible data hiding," in *Proc. of IEEE Intl. Conference on Image Processing*, vol. 2, pp. 157–160, 2002.
7. T. M. Cover and J. A. Thomas, *Elements of Information Theory*. John Wiley & Sons, 1991.
8. IBM Corporation, *Book I: PowerPC User Instruction Set Architecture, Version 2.01*, 2003.
9. Standard Performance Evaluation Corporation (SPEC). `http://www.spec.org`.
10. G. G. Pechanek, S. Lorin, and T. Conte, "Any-size Instruction Abbreviation Technique for Embedded DSPs," *15th IEEE Intl. ASIC/SOC Conf.*, pp. 8–12, 2002.
11. J. Lee, K. Choi, and N. Dutt, "Efficient Instruction Encoding for Automatic Instruction Set Design of Configurable ASIPs," *Proceedings of the IEEE/ACM international conference on Computer-aided design*, pp. 649–654, 2002.
12. J. Liu, T. Kong, and F. C. Chow, "Effective compilation support for variable instruction set architecture," in *IEEE International Conference on Parallel Architectures and Compilation Techniques*, pp. 56–67, 2002.
13. R. L. Davidson and N. Myhrvold, "Method and system for generating and auditing a signature for a computer program," *US Patent 5,559,884*, 1996.
14. R. Venkatesan, V. Vazirani, and S. Sinha, "A graph theoretic approach to software watermarking," in *Proc. of 4th Intl. Workshop on Info. Hiding*, pp. 157–168, 2001.

15. C. Collberg, E. Carter, S. Debray, H. Huntwork, J. Kececioglu, C. Linn, and M. Stepp, "Dynamic path-based software watermarking," *ACM SIGPLAN Notices*, vol. 39, pp. 107–118, May 2004.
16. L. Yuan, P. R. Pari, and G. Qu, "Soft IP protection: Watermarking HDL codes," in *Proc. of 6th International Workshop on Information Hiding*, pp. 224–238, 2004.
17. C. W. Honsineger, P. W. Jones, M. Rabbani, and J. C. Stoffel, "Lossless recovery of an original image containing embedded data," in *US Patent 6,278,791*, 2001.
18. R. El-Khalil, and A. Keromytis, "Hydan: Hiding information in program binaries," in *International Conf. on Information and Communications Security*, ICICS 2004.

Appendix - Description of the SPEC benchmarks [9]

Program Name	Program Type	Language	Description
SWIM	float	Fortran 77	Shallow water modelling software.
ART	float	C	Adaptive Resonance Theory (ART) neural network - used to recognize objects in a thermal image
WUPWISE	float	Fortran 77	Wuppertal Wilson Fermion Solver - a program in quantum chromodynamics
EQUAKE	float	C	Simulates seismic wave propagation
LUCAS	float	C	Lucas-Lehmer test for primality check
APPLU	float	Fortran 77	Computational fluid dynamics and physics
MCF	integer	C	Combinatorial optimization
FACEREC	float	Fortran 90	Implementation of a face recognition system
GZIP	integer	C	GNU zip for data compression
BZIP2	integer	C	Compression program
APSI	float	Fortran 77	Program used in weather prediction
AMMP	float	C	Program used in computational chemistry to model large systems of molecules
GALGEL	float	Fortran 90	Program used in computational fluid dynamics
PARSER	integer	C	Program used for word processing
CRAFTY	integer	C	A high-performance computer chess program
TWOLF	integer	C	Used in computer aided design
EON	integer	C++	A probabilistic ray tracer based computer visualization program
MGRID	float	Fortran 77	A simple multi-grid solver in computing three dimensional potential field
VORTEX	integer	C	A single-user object-oriented database transaction benchmark
VPR	integer	C	Versatile Place and Route (VPR) is a FPGA circuit placement and routing program
PERLBMK	integer	C	A cut-down version of Perl v5.005_03 program
MESA	float	C	A free OpenGL work-alike 3D graphics library
GAP	integer	C	Implements a language and library designed mostly for computing in groups
FMA3D	float	Fortran 90	A finite element method computer program designed for Mechanical Response Simulation
SIXTRACK	float	Fortran 77	High energy nuclear physics accelerator design
CC1	integer	C	C++ language compiler