An Interpretation-like Technique for Monitoring x86 Code on an x86 Host Machine*

Fabiano Ramos, Valmir C Barbosa, and Edil S T Fernandes
PESC – COPPE
Federal University of Rio de Janeiro
ramosf, valmir, edil@cos.ufrj.br
Relatório Técnico ES–659/04

Abstract

We present an interpretation-like technique which reproduces the execution environment of binary code on x86 machines. This technique looks like an interpretation process because it accompanies the execution of application programs, in an interpretative fashion, investigating either general or specific peculiarities of the binary code being monitored, and it leaves to the underlying hardware the actual execution of the binary code. Our technique is reliable and can be used for several purposes, as for instance, to detect program bugs and self-modifying code. Besides describing the interpretation technique, the paper presents some algorithms to help the programmers to validate, assess, and debug their binary code. A study evaluating the efficiency of the technique during the characterization of the integer programs of the SPEC2000 suite is presented.

1. Introduction

Binary Interpretation is a technique that has been used in the implementation of recent processors. For instance, Intel and AMD are companies that use "Binary Translation and Interpretation" to implement the x86 repertoire. Transmeta on the other hand, uses a software layer to achieve the compatibility between its VLIW architecture and x86 processors [2, 4].

The advent of superscalar processors and their scheme of out-of-order and speculative execution, made the identification and elimination of hardware/software malfunctions a very hard task.

From the software viewpoint on the other hand, despite all the innovations introduced in the design of recent processors, there is a lack of the corresponding tools to debug the related application programs, and for this reason, "the debugging scandal" described in [6] still holds today: as mentioned by P. Zhou et al. in a recent work [3], the cost of software bugs is very high.

In addition to the implementation of processors, Binary Interpretation has also been used in debugging. The Valgrind [9] is a remarkable example: it is an emulator that reproduces the activities of an x86 processor. It helps to find memory-management bugs, provides a report of the cache accesses, and so on. The author of this tool, Julian Seward, was one of the winners of the "Open Source Awards 2004." According to the author, a shortcoming refers to the report of the cache provided by the tool. It assumes that other processes and the kernel functions are not sharing that resource. In other words, the cache misses figures exhibited in the report are approximated.

In this work we are concerned with the interaction between the Computer Architecture and the application program. There are several architecture topics to be investigated in this area, but industrial/commercial interests prevent the makers to disclose important details of how their $\mu-$architecture works. For example, it would be useful to know precisely how many speculative instructions were executed and discarded by current superscalar processor because they belong to the wrong path. In other words, what is the percentage of useless instructions which were executed? Is it worthwhile to pay that price? Or would be better to have a more selective mechanism to decide whether the speculation should be activated?

In order to surmount the lack of information, a team at University of Illinois developed the rePLay framework [7] which reproduces the behavior of Intel and AMD $\mu-$architectures (like the Pentium and Athlon). Using this framework, the team conducted experiments with optimization of micro-operations and found that a significant number of redundant micro-operations are executed by the $\mu-$machine because the x86 decoder ignores the current contents of the ISA x86 registers [10].

* Work Sponsored by CNPq, grant 552091/02–2

The interpretation-like technique presented here is an effort to provide a more suitable tool to assess the effects provoked by code optimizers and hardware features on the performance of computing systems. Our technique is non-intrusive and it accompanies the execution of a statically linked x86 binary, being unnecessary to recompile or relink the executable program. By deciding in favor of a non-intrusive approach, we prevent the insertion of instrumenting instructions into the binary code to be monitored, and for this reason, we are sure that its original behavior on the host machine will be the same on our interpretative environment.

This interpretative environment handles x86-Linux executables programs and is based on the `ptrace` function of the Linux. We were benefited from the help provided by the Linux-kernel people that introduced a patch fixing a problem we had with the `ptrace` function.

The ptrace syscall allows a parent process to control the execution of another process, watching and changing its core image and registers. In our implementation, the parent is the interpreter process and the child is the x86 code to be monitored. Figure 1 shows a diagram of our binary interpretation strategy.

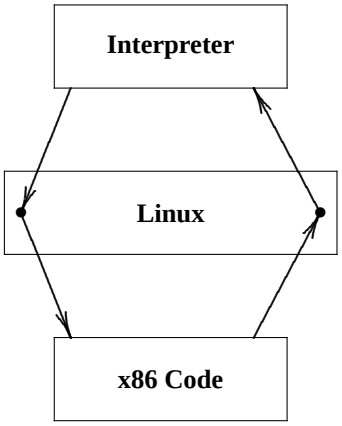

Figure 1: Interpretation-like Diagram

As illustrated in Figure 1, our interpretation process uses the Linux kernel (via the `ptrace` function) to control the progress of the x86 code. In this way, we explore the ptrace facility to switch the machine control between the interpreter and the x86 code being monitored.

The paper is organized as follows. Section 2 provides an overview of the techniques used in the work. Section 3 presents our interpretation-like method, and the following section discusses some variations of the method. Section 5 describes how the integer programs of the SPEC2000 were interpreted and characterized. Section 6 concludes.

2. Overview

Before transferring the control to the code being monitored, the interpreter must specify how long the child process should take the processor control each time it is activated. The code portion to be executed by the monitored process can be: a single instruction; one basic block; the whole binary program; until the next control transfer instruction; after the execution of a particular type of instruction; and so on. It is up to the user to decide where the execution should be interrupted. Consequently, our interpretation-like technique can also be used to detect software bugs.

In order to select the point in the original code where the control should be transferred back to the interpreter, the user must replace the op-code in that point by a trap. Whenever the control flow reaches a trap, occurs a context switch to the interpreter.

We will start by presenting the topics that are essential for our approach. Basically, they are related with the data structure that guides the interpretation; the disassembly of the binary code (see [1, 8]), including the main obstacles present in the x86 repertoire; and the more updated `ptrace` function of the Linux used in our implementation.

2.1. Code Description Structure

This structure can be considered as an interpretation guider. The interpreter examines it to find where the next breakpoint will be injected in the code. The structure describes the instructions of the x86 program, and it contains the following fields: for each byte of the `text` section there is an entry on the array. Each entry has a copy of the corresponding byte of the instruction and other fields specifying if that byte of the executable is: the first, the last or an intermediate byte of the instruction; idem for a basic block; and bookkeeping fields (e.g., how many times the corresponding instruction was executed so far; how many bytes the instruction have; how many instructions form the basic block, and so on).

We decided to keep in the structure a copy of the `text` section because in some cases we need to check the presence of self-modifying code. In this way, whenever a new interpretation cycle starts, the instruction pointed by the program counter (the “eip” register) is compared with the corresponding bytes in the structure.

An x86 instruction starts in any byte of the `text` and it can be formed from one up to sixteen bytes. For this reason, the structure has fields telling what is the position of the byte within the instruction (i.e., fields indicating start, intermediate, and last byte of the instruction). The same applies for the fields related to a basic block of the program.

2.2. x86 Code Disassembly

Before the interpretation, the fields within the description structure are fulfilled by a disassembler. Most of the instructions and basic blocks can have their bounds found by this static disassembler. However, there are instructions and basic blocks that are impossible to disassemble statically. In these cases we leave to the interpreter this task. It is worthwhile to mention that disassembling a binary for CISC machines is a very complex task: jump tables and other data may be mixed-up with the instructions, confusing the disassembler and "silent errors" can be generated along the process [8].

2.3. Dynamic Disassembly

Static disassemblers cannot detect the target address of all branch instructions because some of them depend on the results produced at run time. It is essential to find these target addresses because new basic blocks may be hidden, and disclosing their existence will make the guider structure much more precise. The interpretation-like technique allows to fulfill some empty fields of those instructions (and basic blocks) that have been executed at least once by the x86 code. In other words, our interpretation-like technique can play the role of a dynamic disassembler as well.

3. The Interpretation-like Technique

As mentioned before, Linux provides a system call that allows a process to trace the execution of another (i.e., the child process) leaving to the kernel the task of activating/deactivating the child. We have specified two interpretation modes: **single-step** and **leap** interpretation modes.

3.1. Interpretation Modes

Whenever the interpreter calls `ptrace()`, the Linux sets the trap bit of `eflags` register of the child and makes it runnable. After the child executes its next instruction, a debug trap is generated and the Linux, as specified by the debug handler, stops the child, clears the debug bit and notify the interpreter. The child will only be made runnable again when its parent makes a new call to `ptrace()`. These actions characterize our single-step interpretation mode.

For example, if our technique is acting as a dynamic disassembler, the single-step mode is used. In this case, at the end of each instruction of the child, the interpreter examines the structure and checks the new contents of the program counter. Eventually, a new target address of a branch can be detected and some fields of the interpretation structure updated. Also, the interpreter can verify whether the contents of the structure fields, which were fulfilled before by the static disassembler, must be modified or not (e.g., to increment the instruction counter of the binary code being monitored).

Detection of self-modifying code, can be done in a similar way: the interpreter receives the control after the execution of each instruction of the child, compares the instruction to be executed (pointed to by the program counter of the child) with the instruction in the structure.

In the leap interpretation mode, we have a variable number of child instructions being obeyed before the interpreter takes the control again. This interpretation mode reduces the overhead provoked by the huge number of context switches which occurs with the single-step mode, and for this reason, the interpretation process is faster.

3.2. Single-step Algorithm

Next, we outline the single-step algorithm.

```
1:  InstrCount = 0
2:  while (program does not finish) do
3:     single-step through next instruction
4:     InstrCount = InstrCount + 1
5:     if (PC is within the text segment) then
6:        I = instruction at PC
7:        if (PC is inside text section) then
8:           LAST = last instruction executed
9:           if ((PC is not the address that follows LAST)
             and (PC is not a target of LAST) ) then
10:             PC starts a new basic block
11:          end if
12:          if (I is a branch instruction) then
13:             Mark both targets as new basic blocks
14:          end if
15:       end if
16:    end if
17: end while
```

3.3. Cost of Single-step Mode

Before the interpretation of an instruction in single-step mode, some context switches are required. First, the interpreter executes a syscall to ptrace(), causing a switch to kernel mode. The Linux sets a trap bit in the x86 code and resumes its execution, causing another switch. After the interpretation of the x86 instruction, there is another context switch (from user to kernel mode), and finally another one to the interpreter.

During the interpretation in leap mode, this overhead has less impact because we only need to switch the context at those instructions specified as the end of a leap and the instructions outside the text segment.

3.4. Leap Mode Interpretation

In order to specify the end of a leap, the interpreter injects a trap instruction (opcode 0xCC) into the program, causing the interpreter to be invoked when the child reaches this breakpoint. For example, if we are interested to execute a sequence of instructions between two control transfer instructions, then at the start of the code portion, we search for the next control transfer instruction in the structure. When such instruction is detected, a trap instruction is injected into the original code and the child program execution is resumed.

The interpreter will receive the control again after the execution of the trap instruction. The original code is then restored (and the program counter of the child as well), the original branch is executed in single-step mode, and the procedure is repeated.

We must be careful with instructions preceded by repetition prefixes, such as `repz`. These prefixes forces the corresponding instruction to be executed several times. Since these instructions are not branches, then at the end of the interpretation, the instruction count can have a wrong value. To solve this inconsistency, we can consider any instruction preceded by these prefixes as a conditional branch instruction, whose target address points to itself.

A simplified description of the leap-mode algorithm follows.

```
1:  while (program does not finish) do
2:      single-step until next breakpoint
3:      if (PC is within the text segment) then
4:          I = instruction at PC
5:          if (PC is within the text section) then
6:              LAST = last instruction executed
7:              if ((PC is not the address that follows LAST)
                and (PC is not a target of LAST) ) then
8:                  PC starts a new basic block
9:              end if
10:             if (I is a branch instruction) then
11:                 Mark its targets as new basic blocks
12:             else
13:                 TEMP = instruction that follows I
14:                 while (TEMP is not a branch instruction) do
15:                     TEMP = instruction that follows TEMP
16:                 end while
17:                 replace TEMP with 0xCC
18:                 restart child and waits for notification
19:                 restore TEMP
20:             end if
21:         end if
22:     end if
23: end while
```

3.5. Interpretation Hazards

A traditional tool used to find program errors is the GDB. In our case, there are two disadvantages with that debugger. First, it is only helpful if the source code exists. Second, the injection of extra commands into the object code is not acceptable because it alters the actual behavior of the program being investigated. For example, the addition of a new instruction can interfere in the instruction cache performance, perhaps hiding cache collisions because some instructions were moved to another cache line, and so on.

Other interpretation hazards that we found were the repeat prefixes and that minor problem with the `ptrace` function which was fixed by the Linux kernel staff.

4. Interpreting x86 code

4.1. Platform

The results presented in this paper refer to the Intel x86 architecture, running the Linux Operating System. Our experiments were carried out on a 3.0 GHz Pentium IV-HT, with Slackware Linux 9.1, kernel version 2.6.6 (with the ptrace patch). All test programs were compiled with GCC 3.2.3 and binutils 2.14.90 library.

The integer programs from the SPEC2000 suite were used in the experiments with the reduced input (large set) [5]. Vortex program was executed with the original test input set (provided by the SPEC) because there isn't the little endian format in the reduced input set for this benchmark.

5. Experiments

Currently, we are collecting many characteristics of the SPEC2000 integer programs with our interpretation-like framework. For example, the percentages of Figure 2 were obtained through the single-step mode (the useful instruction values) and the leap mode (the useful basic blocks).

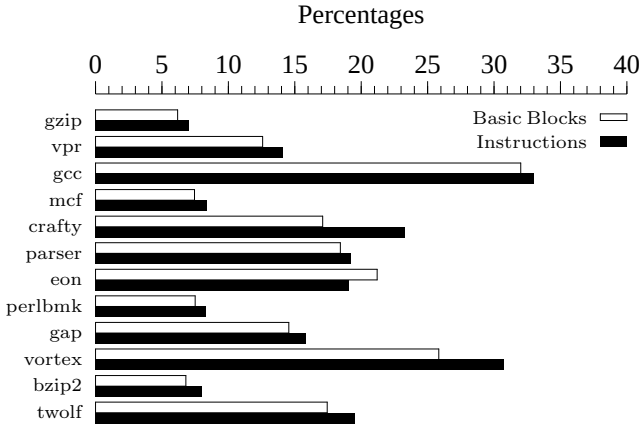

Figure 2: Useful Basic Blocks and Instructions

The empty bars in Figure 2 represent the percentages of basic blocks which were executed at least once, and the dark bars refer to the instructions. Examining these percentages, one can see that less than 40% of instructions (and basic blocks) were useful for the execution of the programs. The remaining, and more significant, portion of each program never has been executed. Actually, we have observed the same behavior for RISC machines (e.g., SimpleScalar). In general, the percentages of useful instructions were larger than the corresponding basic blocks which were executed once. The `eon` was the unique exception.

In order to investigate the reasons leading to the predominance of useful instructions over the basic blocks, we conducted several experiments. In the first, we found the number of basic blocks and instructions of each program. The static and dynamic disassemblers (together with the interpretation-like modes) were used in these experiments. Table 1 gives the static characteristics of the programs.

Program	Instructions	Basic Blocks
gzip	104,225	26,894
vpr	125,803	30,964
gcc	449,320	120,492
mcf	95,036	24,673
crafty	137,325	33,009
parser	127,046	33,162
eon	359,719	75,907
perlbmk	234,273	62,347
gap	220,449	56,058
vortex	226,662	53,220
bzip2	102,823	26,519
twolf	141,953	34,520

Table 1. Static Characteristics

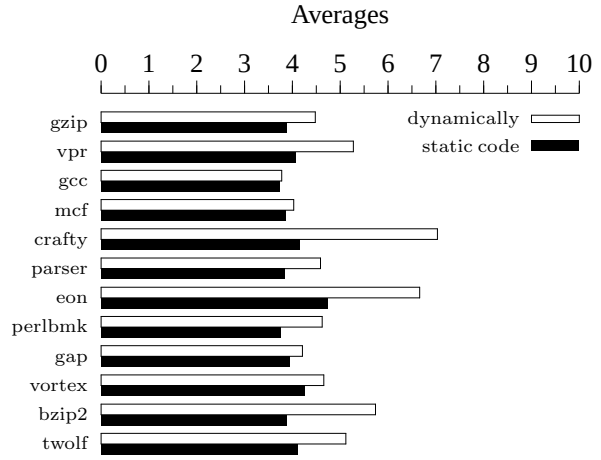

Figure 3: Average Numbers of Instructions per Basic Block

Next, the experiments were performed again to find the average number of instructions within each basic block. Figure 3 gives the numbers of instructions per basic block. The dark bars in Figure 3 represent the average numbers of instructions within the basic blocks, and they were obtained from the static code. The empty bars on the other hand, are the averages found at execution time.

According to the values in Figure 3, the averages numbers of instructions which were evaluated dynamically overcome the corresponding ones in the static code. This happens because there are many blocks in the static code formed by one, two and three instructions. However, the execution frequencies of larger blocks contribute for the higher average numbers. Also, it should be mentioned that the average numbers of instructions per basic block range from 3.77 up to 7.03 for the dynamic evaluation.

An x86 instruction has from one up to sixteen bytes. Consequently, the averages numbers provided in Figure 3 are not enough to have an idea of the space in the various memory levels occupied by a basic block. To provide an insight of this space, we performed experiments to evaluate the average sizes (in bytes) of basic blocks, both in the static code and dynamically. Figure 4 shows these averages for each benchmark obtained from the static code and at execution time.

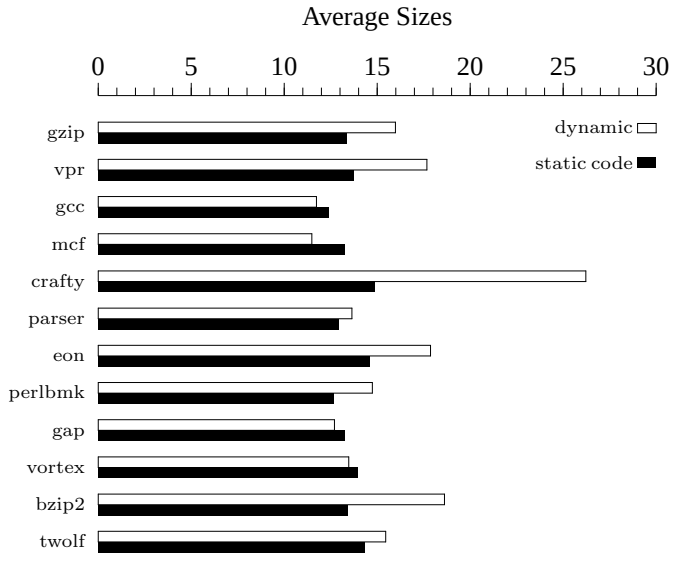

Figure 4: Average Sizes of Basic Blocks in Bytes

In Figure 4, the average sizes evaluated dynamically overcomes the corresponding ones for eight benchmarks. Like as in the previous experiment (average number of instructions per block), the `crafty` program presented the higher average number of bytes (26.21 bytes).

Tables 3, 4 and 5 in the Appendix provide the whole values obtained by the experiments and partially presented in this section.

6. Conclusions and Future Work

In this work we have shown a Binary Interpretation-like technique for x86 machines with the Linux Operating System. Our technique is oriented for binary code and for this reason the source program is unnecessary. This Interpretation process is reliable because we use the underlying hardware to perform the last stage of the interpretation (i.e., the execution) of each x86 instruction. Consequently, the interpreter is exempted from those errors that may be found within a software layer responsible for the interpretation process.

The overhead imposed by the great number of context switches makes such interpretation process very slow. However, the technique became viable with the advent of recent processors which operates at very high frequency rates. We are using hyper-thread processors (the Pentium-4 HT model from Intel) and techniques to distribute the interpreter and the x86 code being monitored in separate threads are already being investigated.

Our interpretation mode using a variable number of instructions as the standard unit of switching is a very efficient alternative for reducing the overhead of the technique when the single-step mode is used.

Many research topics in Computer Architecture can be carried out with the Interpretation technique presented here. Identifying the execution frequencies of instructions and basic blocks, the percentages of untouched instructions of a program, and how may time a particular memory load instruction transfers the same value, are examples of features provided by our interpretation-like framework.

References

[1] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software: Practice & Experiece*, 25(7):811–829, 1995.

[2] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing^{TM} software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the international symposium on Code generation and optimization*, pages 15–24. IEEE Computer Society, 2003.

[3] P. Z. et al. iWatcher: Efficient architectural support for software debugging. In *Proceedings of the 31st annual international symposium on Computer architecture*, page 224. IEEE Computer Society, 2004.

[4] A. Klaiber. The technology behind the crusoe processors. In *URL http://www.transmeta.com/pdf/white_papers /paper_aklaiber_19jan00.pdf*. Transmeta Corporation, 2000.

[5] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, 2002.

[6] H. Lieberman. The debugging scandal and what to do about it. *CACM*, 40(4):26–29, 1997.

[7] S. J. Patel and S. S. Lumetta. rePLay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):590–608, 2001.

[8] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 45. IEEE Computer Society, 2002.

[9] J. Seward. Valgrind: an open-source memory debugger for x86-GNU/Linux. In *URL http://www.ukuug.org/events/ linux2002/papers/html/valgrind/*, 2002.

[10] B. Slechta, D. Crowe, B. Fahs, M. Fertig, G. Muthler, J. Quek, F. Spadini, S. J. Patel, and S. S. Lumetta. Dynamic optimization of micro-operations. In *Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture (HPCA'03)*, page 165. IEEE Computer Society, 2003.

Appendix

Program	Instr. Count	Instructions	B.Bls
gzip	1,916,484,097	104,225	26,894
vpr	1,530,725,506	125,803	30,964
gcc	3,236,426,097	449,320	120,492
mcf	596,748,718	95,036	24,673
crafty	949,969,122	137,325	33,009
parser	3,051,631,424	127,046	33,162
eon	5,255,974,938	359,719	75,907
perlbmk	1,243,359,184	234,273	62,347
gap	563,477,156	220,449	56,058
vortex	8,189,657,958	226,662	53,220
bzip2	1,751,445,745	102,823	26,519
twolf	797,767,520	141,953	34,520

Table 2. Program Characteristics

Program	Instructions	Exec	Exec (%)
gzip	104,225	7,352	07.05
vpr	125,803	17,766	14.12
gcc	449,320	148,428	33.03
mcf	95,036	7,968	08.38
crafty	137,325	32,007	23.31
parser	127,046	24,399	19.20
eon	359,719	68,668	19.09
perlbmk	234,273	19,424	08.29
gap	220,449	34,924	15.84
vortex	226,662	69,656	30.73
bzip2	102,823	8,245	08.02
twolf	141,953	27,740	19.54

Table 3. Instructions

Program	B.Bls	Exec	Exec (%)
gzip	26,894	1,661	06.18
vpr	30,964	3,896	12.58
gcc	120,492	38,566	32.01
mcf	24,673	1,837	07.45
crafty	33,009	5,643	17.10
parser	33,162	6,111	18.43
eon	75,907	16,093	21.20
perlbmk	62,347	4,674	07.50
gap	56,058	8,156	14.55
vortex	53,220	13,753	25.84
bzip2	26,519	1,802	06.80
twolf	34,520	6,019	17.44

Table 4. Basic Block Execution

Program	Static	Dynamic
gzip	13.374	15.980
vpr	13.758	17.672
gcc	12.414	11.733
mcf	13.246	11.485
crafty	14.855	26.218
parser	12.920	13.640
eon	14.596	17.867
perlbmk	12.680	14.739
gap	13.278	12.701
vortex	13.958	13.470
bzip2	13.422	18.617
twolf	14.344	15.457

Table 5. Average Sizes of Basic Blocks in Bytes