Verifying Program Profiles [*]

Patrick Moseley, Saumya Debray, Gregory Andrews

Department of Computer Science
University of Arizona
Tucson, AZ 85721.
{moseley, debray, greg}@cs.arizona.edu

ABSTRACT

Execution profiles have become increasingly important for guiding code optimization. However, little has been done to develop ways to check automatically that a profile does, in fact, reflect the actual execution behavior of a program. This paper describes a framework that uses program monitoring techniques in a way that allows the automatic checking of a wide variety of profile data. We also describe our experiences with using an instance of this framework to check edge profiles. The profile verifier uncovered profiling anomalies that we had been unaware of and that would have been very difficult to identify using existing techniques.

1. Introduction

Profiling is commonly used to measure aspects of performance. Programmers use high-level tools, such as *gprof* [12], that sample the program counter in order to estimate how much time is spent in each function. Compilers now routinely use profile information such as basic block and edge counts to direct a wide variety of code optimizations [6, 7, 9, 10, 11, 20]. Finally, many processors contain hardware performance counters that measure low-level events such as cache misses and branch mispredictions.

This paper addresses an issue that occurs in profiles that are generated and used by compilers and related tools such as link-time optimizers. These kinds of profiles typically associate runtime execution counts with components of a program, e.g., basic blocks, control flow edges, or paths in a control flow graph. The specific focus of this paper is the following question: How can one tell whether the counts generated by a profiler accurately reflect the behavior of the program being profiled?

Interestingly, while there has been a great deal of research on testing programs (e.g., see [8, 15, 17, 21]), the question of testing program profilers has received relatively little attention. From our own experience and what we have learned from others, the prevailing practice in testing profilers is very *ad hoc*: “eyeball” the profiling code, make sure the profiling and original version of a program produce the same “regular” output, generate expected profile counts manually for a few sample programs, and check that these match the profiler-generated counts for the same programs. This is hardly a satisfactory situation. First, it is time consuming and error prone. Second, it is reasonable to compute execution counts manually only for a small number of small programs—especially if one is looking at complex profiles, such as for type feedback in object-oriented languages [14], edge-pair profiling [18], or path profiling [4]. This means that profiling bugs that show up only on large inputs will not be detected. Finally, erroneous results can result from the fact that compiler-generated profiling is intrusive: It inserts profiling code in the program being profiled, which alters the memory layout of the original program and hence can alter its behavior.[1]

A significant reason why the testing of program profiles is more difficult than conventional program testing is that it deals with a fundamentally different view of program execution. Conventional program testing takes the denotational view that what is significant about a computation is *what* is computed, i.e., the output produced for a given input. Therefore conventional testing focuses on ascertaining that these outputs are being computed correctly. Profiles, by contrast, capture an operational view of a computation, where the property of interest is *how* the computation is carried out. Because of this, in order to check a program’s execution profile, it is necessary to observe the actual computation of the program, not just the end result.

An execution profile is in essence a set of assertions about the execution of a program on some specific set of inputs, e.g., “basic block x is executed m times” or “program path y is taken n times.” We would like to be able to check, automatically and for arbitrary programs, that these assertions are correct—i.e., that when executed on the same set of inputs, basic block x is, in fact, executed m times, or that program path y is taken n times. The issue here is not that we must have “exact” execution counts for code optimization. In fact, several researchers have observed that approximate profiles can be useful and effective for optimization purposes [2, 5, 13]; these profiles are generated and used deliberately, with knowledge that precision is being sacrificed, an understanding of where and how much precision is lost, and some sense of how much loss can be tolerated. Rather, the issue is how to guard against imprecision that is *inadvertently* introduced as a consequence of bugs in the profiler. This kind of imprecision is unsuspected and unpredictable. Thus, it can cause, for example, heavily executed portions of code to appear to have low execution counts, which in turn can result in misoptimized programs, performance anomalies, and, for researchers, faulty experimental data.

This paper presents a technique for automatically verifying many different kinds of profile data. Our approach has two components: a general framework and a specific checker. The general framework uses a trap-based scheme, similar to that used in a debugger, to monitor the behavior of a program at runtime. A checker for a specific type of profile data is then built on top of the framework. A significant advantage of our approach is that because the monitoring code runs in a separate address space, the program whose profile is being checked does not have to be modified. This helps avoid profiling bugs that arise from changes to code or data addresses in the program being profiled as a result of inserting instrumentation code.

[*]This work was supported in part by the National Science Foundation under grants CCR-0073394, EIA-0080123, and CCR-0113633.

[1]This in fact happens in the *gcc* benchmark that is part of the SPEC-95 benchmark suite. We discovered this using our profile verifier; see Section 5 for details.

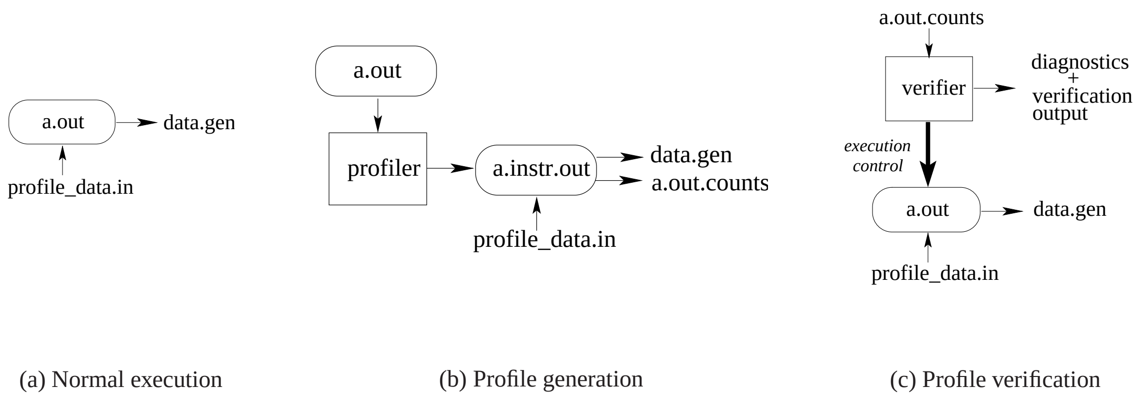

Figure 1: Normal execution, profiling, and profile verification

We have implemented the general framework and a checker for edge profiles in the context of the PLTO binary rewriting system [23]. Section 2 gives more detailed background information on compiler-level profiling and on our approach to profile verification. Section 3 describes our implementation of the general framework and a checker for edge profiles generated by PLTO. Section 4 describes subtle issues that arose during the implementation and testing of our framework. Section 5 describes what we learned when we used our checker to verify profiles of the SPEC-95 benchmarks, and it gives information on execution times. Finally, Section 6 gives concluding remarks.

2. Profiling and Profile Verification

Intuitively, profiling involves counting specific events of interest—e.g., entering a basic block, or taking a particular control flow edge. Events are counted each time they happen during the execution of a program, then this data is written out at the end of the program's execution. In order to profile a program, it is first *instrumented*, i.e., it is rewritten to insert additional *instrumentation code* that records the runtime events of interest as they happen. Figure 1 illustrates this. Figure 1(a) shows the normal execution of a program *a.out*, which takes the input *profile_data.in* and generates the output *data.gen*. Figure 1(b) shows how the program is profiled. The original binary *a.out* is passed through a profiler, which produces an instrumented version. The instrumented version *a.instr.out* is then executed with the same input; it writes *data.gen* and the profile data *a.out.counts*.

As an example, in order to obtain execution counts for each basic block in a program (a basic block profile), we might insert code into each basic block to increment a block-specific counter each time that block is entered. Note that the insertion of this instrumentation code can cause code addresses to change. More complex profiles, e.g., value profiles [5, 19] or path profiles [4], require correspondingly more complex processing, but the general idea remains the same. Profilers may also employ more sophisticated profiling logic to reduce runtime overheads [3, 26], but again the essential idea does not change.

In addition to inserting instrumentation code for counting and recording profiling events, a profiler needs to allocate memory space for the profile counters themselves. For simple kinds of profiles, such as basic block and edge profiles, the number of counters that are required can be determined statically by examining the program; therefore, the counters can be statically allocated in the data segment of the instrumented program. For more complex profiles, it may not be possible to determine statically how many counters are required. In this case, the counters may have to be dynamically allocated during execution. In either case, data addresses in the original program might change.

While it may seem a simple proposition to insert instrumentation code into a program, the issues that have to be addressed are not always straightforward. For example, the GNU C compiler *gcc*[2] has facilities for basic block profiling, but if the program being instrumented is large—e.g., the SPEC-2000 benchmark *176.gcc*—the instrumented executable can silently generate incorrect profiles when executed, or crash with a segmentation fault. We mention this only to observe that even experienced software developers can sometimes have trouble with profilers, even for something as supposedly straightforward as basic block profiling.

Profilers are often implemented by rewriting binaries (e.g., see [16, 25]). In this case it is imperative that changes to code and data addresses resulting from instrumenting the program be reflected in all parts of the program. Otherwise, the instrumented program may behave differently than the original program. Obviously, if the behavioral differences between the two versions result in different outputs being generated, the problem will be noticed, but we cannot always guarantee that this will be the case. Another problem may arise from the fact that, in general, it is impossible to rule out disassembly errors, especially if there may be data embedded in the text segment [24]. In such cases, e.g., if executable code is mistakenly identified as data (or vice versa), the insertion of instrumentation code can cause the program to crash or to produce an incorrect profile. A more subtle problem can arise if the event counters are too small, e.g., if 32-bit counters are inadvertently used for long running programs where execution counts are too large to fit into 32 bits. This can happen, for example, if code is reused across architectures; e.g., a C variable of type `long` occupies 64 bits on a Compaq Alpha but only 32 bits on an Intel Pentium. We have in fact encountered all the above problems in our own work.

Some other profiling schemes avoid the problem of disassembly errors—e.g., by instrumenting an interpreter or by dynamic instrumentation, in which the program is instrumented as it executes. However, the other problems still remain. Moreover, like binary rewriting, both these schemes perturb the executable: In an instrumented interpreter, the code now executes out of the data segment of the interpreter rather than its own text segment; dynamic instrumentation inserts instrumentation code in the executable.

Even when all the above problems have been addressed, the act of inserting instrumentation code into a program can affect its behavior, possibly in subtle and hard-to-detect ways (e.g., see Section

[2]This refers to *gcc* version 2.96 on Intel Pentium 3 and Pentium 4 workstations running RedHat Linux 7.3. Instrumentation for basic block profiling is generated when the compiler is invoked as *gcc –a*.

5). In implementing a profile verifier, therefore, one of our goals is to avoid perturbing the program being profiled as much as possible. One way to realize this is to have the instrumentation code reside in a different address space than the program being profiled, with control being transferred from program's address space to the instrumentation address space, and back, each time a profile event occurs during the program's execution.

This idea is strongly reminiscent of the way debuggers work [22]. A debugger monitors and controls the program being debugged. It allows the program to be single-stepped an instruction at a time, allows breakpoints to be set, and allows the program to be executed until a breakpoint is reached. In addition, the debugger has access to the memory space of the program being debugged, so it can examine and change the core image of the program. (On Linux, this functionality is supported by the *ptrace* system call).

Our approach to profile verification is based on this idea. The verifier starts the program whose profile we want to check on the appropriate profiling inputs, and then uses the appropriate system call (*ptrace* on Linux) to control its execution. The verifier observes the runtime behavior of a program in the same way a human uses a debugger to debug a program: by single stepping, setting breakpoints, running until breakpoints are reached, and examining the memory space of the program. In the process, the verifier can count profiling events, e.g., entry to a basic block or control flow along an edge, as they occur and check whether the resulting counts match those generated by the profiler. This is illustrated in Figure 1(c). The verifier controls execution of the original program *a.out* on the original input file *profile_data.in*. Additionally, the verifier takes the event counts data *a.out.counts* as an input and produces diagnostics and verification output as appropriate.

In summary, the goal of our work is to observe and independently count events in the original program—without altering that program—in order to determine whether the counts agree with those of the profiled version of the program. It is also worth emphasizing what our work is not about. It is not about choosing suitable profiling inputs; that is a task that is best left to the application programmer. Nor is it about choosing suitable test programs, e.g., randomly generated programs, to observe and check profilers; that issue is largely orthogonal to the topic of this paper. Finally, it is not about checking the consistency of profiles, i.e., answering the question of whether a profile "makes sense;" all we are checking is whether the profiler computes what it purports to compute.

3. An Example: Verification of PLTO Edge Profiles

We have built the trap-based profile verification framework sketched above and have built a verifier for edge count profiles generated by the PLTO binary rewriting system for Intel Pentium processors [23]. The trap-based framework was implemented on Linux using the *libDebug* library from the *ald* debugger [1]. This section first describes the basic operation of our verifier then discusses optimizations we used to improve its performance.

3.1 Basic Operation

A PLTO edge count profile represents an edge as a pair of basic blocks and a counter; each basic block is denoted by its starting address in the program that was profiled. The verifier first reads a program's edge count profile, creates a control flow graph for the program using the edge information, and initializes edge counts in this control flow graph with values read from the edge count profile. The verifier then starts the program and monitors its execution. As the program executes, the verifier has two main tasks:

1. Check the structure of the control flow graph and report any anomalies that are detected, such as missing edges or basic blocks. The existence of anomalies point to disassembly errors in the profiler (see Figure 1(b) and [24]).

2. Decrement edge counters as control flow edges are traversed.

When the program being verified terminates, the verifier outputs all of the edges in the control flow graph that do not have counts equal to zero. These nonzero edge counts indicate where the profile data did not match the execution behavior of the program as seen by the verifier.

The technique used to follow and verify the control flow graph is straightforward. The verifier starts executing the program at the beginning of the basic block corresponding to the entry point of the program. The verifier regains control at the end of that basic block, finds the start of the next basic block, decrements the appropriate edge counter, and then repeats these actions. This iterative process is implemented as follows:

1. The verifier records the address of the current basic block B, then finds the last instruction in B, as described below.

2. The verifier then sets a breakpoint at the last instruction in B.

3. The verifier returns control to the program, which executes at native speed until it encounters the breakpoint at the end of block B. This returns control back to the verifier.

4. The profile verifier then executes the last instruction in basic block B. This is accomplished using the *ptrace* single-step interface, which executes a single instruction then returns control to the verifier.

5. After single stepping, the program is at the start of some basic block B'. At this point, there are three possibilities:

   (a) Block B' exists and there is an edge in the control flow graph from B to B', so the count associated with the edge $B \rightarrow B'$ is decremented. This is the expected case.

   (b) Block B' does not exist in the control flow graph—i.e., there is no block with the same starting address as B'—so it is reported as missing and added to the control flow graph. This may cause an existing basic block to be split into two blocks. (In this case, a new edge is also created; its count is initialized to the sum of the edge counts into the block that was split.)

   (c) Block B' exists but the edge $B \rightarrow B'$ does not exist, so that edge is reported as missing and added to the control flow graph.

To find the last instruction in a basic block B, the verifier starts at the first instruction in B and repeatedly disassembles each instruction I until one of the following two conditions holds:

1. If I is a branch instruction, then I is the last instruction in B.

2. If the starting address of I is in the list of basic block starting addresses obtained from the control flow graph being verified, then I is the first instruction in the block after B. (This situation occurs when there is a fall-through edge from one block to another in the control flow graph.) In this case, the instruction immediately preceding I is the last instruction in block B.

3.2 Optimizations

The basic trap-based verifier described above turns out to be quite slow. This is due primarily to the overhead associated with using the *ptrace* system call. To speed up the trap-based profile verifier, we have therefore focused on reducing the number of times the verifier has to use *ptrace* to access the profiled program's memory. We are able to reduce the number of calls by caching key values, as described below.

An especially significant source of overheads results from the verifier's need to disassemble the profiled program's instructions every time it is looking for the end of a basic block. In order to examine memory in the program being verified, *ptrace* must be called for each word of data. In the basic verifier described above, each instruction in a basic block is disassembled each time that basic block is executed, so the total number of calls to *ptrace* is roughly comparable to the total number of instructions executed by the original program.

We can reduce this overhead dramatically by having the verifier cache the end address of each basic block in the profiled program, i.e., the address of the last instruction in the block. This means that the verifier only has to disassemble a basic block once to find its last instruction. (There is one exception to this, which occurs infrequently: If a basic block is split as a result of a missing basic block in the control flow graph, the verifier has to disassemble the block that was split one more time to identify the new last instruction.)

We can reduce the number of *ptrace* calls still further by having the verifier cache some additional information about the type of the last instruction in a block. It turns out that PLTO does not profile certain types of control flow edges, e.g., function return edges. Hence the verifier has to determine the type of the control flow edge out of a block in order to avoid giving spurious diagnostic messages. This, in turn, is governed by the type of the last instruction in the block. Thus, even if we know the address of the last instruction in a block, we still have to determine the type of that instruction. This requires disassembling the instruction each time the block is executed, which requires *ptrace* calls. We can avoid these calls by caching the type of the last instruction in a block the first time we disassemble the block.

These optimizations turn out to be *very* effective in reducing the overheads associated with *ptrace* calls. As three data points:

1. The verification time for the SPECfp-95 benchmark program *fpppp* dropped from 1 hour 51 minutes 43 seconds to 1 minute 56 seconds, an improvement of about 58 times.

2. The time for the SPECint-95 benchmark *compress* dropped from 17 minutes 40 seconds to 2 minutes 56 seconds, an improvement of about 6 times.

3. The time for the SPECint-95 benchmark *gcc* dropped from 8 hours 56 minutes to 1 hour 53 minutes, an improvement of about 5 times.

The different amounts of improvement for these programs can be explained by differences in their execution characteristics. The *fpppp* program has a well-defined hot spot consisting of a loop containing a very large basic block; hence, caching the address of the last instruction avoids the very large overhead of repeatedly disassembling this block. The *compress* program also has a well-defined hot spot, but it contains much smaller basic blocks than in *fpppp*; hence, the disassembly overhead is less than for *fpppp*, so there is less room for speedup due to caching. The *gcc* program has a much more diffuse profile than *fpppp* or *compress*, with less well-defined hot spots; consequently, cached information is used less frequently than in the other two programs.

Even with the above caching optimizations, we still have to use *ptrace* to disassemble the instructions in a block the first time the block is executed. We attempted to eliminate these *ptrace* calls by having the verifier use the *mmap* system call to access the text segment of the executable file, which is available on disk. However, for reasons we do not yet understand, this gives mixed results, with significant speedups on some processors and slight slowdowns on others. We are currently examining this puzzle.

4. Implementation and Environmental Issues

We encountered two kinds of issues when implementing and testing our profile verification framework. The first results from peculiarities of the Intel IA-32 (Pentium) architecture and the Linux operating system; other processors and operating systems might have similar peculiarities. The second kind of issue arises from subtle or nondeterministic aspects of the environment in which the profiler and verifier execute; these issues are inherent to generating and checking profiles.

4.1 Host Processor and Operating System

The trap-based profile checker follows edges in the control flow graph by finding the last instruction in a basic block and single-stepping that instruction to find the first instruction in the next basic block. Thus we assume that after single-stepping an instruction, the program counter contains the address of the next instruction to be executed. We found two situations that we have to treat specially because this assumption does not hold. One results from the Pentium architecture's repeating string instructions; the other results from how the Linux implementation of *ptrace* handles the Pentium's interrupt instruction.

A repeating string instruction on the Pentium is an instruction that repeatedly executes one of the Pentium's basic string instructions until a specified terminating condition is met; then the program counter advances to the next instruction in the text segment. Thus, a repeating string instruction is essentially a one instruction loop, but it is encoded as a single instruction with no explicit control transfer. Single-stepping such an instruction results in the program counter remaining on the instruction until the terminating condition is met. Thus, when such an instruction is the last instruction in a basic block—a situation we have seen several times—we cannot single step the original program to get to the first instruction in the next basic block. On the other hand, we know that the first instruction of the next basic block will be the instruction following the repeating string instruction. Thus, when we see this special case, we set a break point at the next instruction, return to the original program, and let the processor execute the repeating string instruction. When the breakpoint trap returns control to the verifier, we realize that we are already at the first instruction in the next basic block and then proceed as in the normal case.

The second problematic situation results from the Pentium's `int` (interrupt) instruction. Under Linux, the effect of using *ptrace* to single-step an interrupt instruction is that the instruction following the interrupt is skipped and the program counter ends up pointing to the second instruction after the interrupt. Thus, when `int` is the last instruction of a basic block, the trap-based profile checker single-steps the `int` to determine the address of the second instruction of the next basic block, then looks at the previous instruction and determines its address. This situation is rare, but of course it still has to be detected and handled.

4.2 Execution Environment

If the execution environment of the profiler and verifier differ, the verifier might see errors in the profile data and produce diagnostic messages even if the profile is correct. Here, “environment” refers to the settings of a user’s shell variables and anything else external to the program that can affect its execution. This can even include such minutiae as the number of characters in the command line used to invoke the program, because this affects the code that reads in and parses the command line options. For example, we found small differences in the profile of the string library function *strrchr* when programs being verified were invoked with a slightly different path to the output file compared to when their profiles were originally generated. The lesson here is that one has to be careful when interpreting the output from the verifier to distinguish genuine errors from explainable differences due to different execution environments.

A situation that is harder to deal with occurs when the execution of the program being profiled or checked depends on some non-deterministic aspect of its environment, such as the time of day. For example, the following program fragment uses the results of the standard library function *time()* to initialize the random number generator, and then uses the random number generator to generate hash codes:

```
srand( time(NULL) );
...
hash_code = (random()*index) % hash_table_sz;
```

If this kind of code occurs in the original program, no two executions of the program would ever have exactly the same profiles. Hence, a profile checker will always discover differences. However, this is not a limitation of our technique for profile verification. Moreover, by using a profile verification framework such as that described in this paper, such differences become explicit and hence get noticed; Section 5 describes a real example that we discovered.

One concludes from this that the output from the profile verifier must be reviewed carefully to determine which differences are caused by real errors in the profile data and which are environmental differences that were not, or could not be, duplicated. In our use of the profile checker for PLTO edge count profiles, the major sources of environmental differences are the standard library initialization code, the program startup code, and the lower-level implementations of standard library functions. These differences are seen because PLTO operates on statically linked programs; hence, the standard library code is also instrumented. In practice, it is not clear how significant an effect environmental differences may have on different types of profiles, but it is something that needs to be kept in mind.

5. Results

The way we originally tested our implementation of edge profiling using PLTO is, we believe, fairly representative of the prevailing practice with regards to testing profilers. We examined the instrumentation code by “eyeballing” it and checked the generated profiles manually on modest sized test programs where it was practical to manually compute the various edge counts. The generated profiles were then used to guide the implementation and evaluation of optimizations, where the profiles were themselves subjected to scrutiny. There was no indication, during any of this, that the profiles were inaccurate in any way. We were surprised, therefore, when our trap-based verifier exposed situations in which the generated profiles did not reflect the actual runtime edge counts of programs.

The inaccuracies we discovered fall into two broad categories. The first category involves differences in execution counts for standard library functions, such as *open* and *write*, that are used both by the application and by the profiler to write out the profile data at the end of the program’s execution. The effect is that the counts obtained for control flow edges in these functions are slightly higher than their actual values. We knew that there would be differences for these functions; the verifier gave us quantitative output telling us how far the counts were off.

The second type of inaccuracy results from inserting instrumentation code into the program being profiled, which causes addresses to change, thereby changing the execution behavior of the program as well. It manifests itself in the SPECint-95 program *gcc*, where several functions have edge counts that differ between the instrumented and uninstrumented versions of the program. While the differences are small—around 1.5–3%—they are still significant. Moreover, the presence of these inaccuracies was a total surprise. It turns out that this program contains code (file `cse.c`, function `canon_hash()`) that has the form

```
switch (..) {
  ...
  case SYMBOL_REF:
    return (hash
    + ((HOST_WIDE_INT)SYMBOL_REF << 7)
       + ((HOST_WIDE_INT) XEXP(x, 0)
                   & ((1 << HASHBITS) - 1)));
}
```

where `XEXP(···)` yields a pointer value. Thus, the code uses addresses to calculate hash values. Since the instrumented executable has a different size than the original executable, heap addresses start at a different location in the instrumented executable. Because of this, the hash table is actually constructed slightly differently in the instrumented code compared to the original code. This, in turn causes the execution of code that traverses this data structure to behave slightly differently in the two versions.

While the inaccuracies turn out to be relatively small in each of these cases—and benign from the standpoint of code optimization—the important point is that the profile verifier correctly identified a situation where the instrumented executable behaves differently than the original executable. The system implementors can then examine the reasons for the discrepancies in execution counts and address them as appropriate.

This example illustrates that “real” programs can and do use code whose behavior can change due to the presence of instrumentation code. Indeed, it is easy to extend the idea to write similar programs in which the instrumented version behaves very differently, with a completely different hot spot, than the uninstrumented original. This is shown in Figure 2. The original version of this program incurs no collisions while inserting into the hash table, but the instrumented version incurs a collision at every iteration of the loop, i.e., *all* the values hash into the same bucket!

These discrepancies in profiling results also highlight the fact that manual checking is not adequate for testing profilers for correctness. For example, it is completely impractical to check manually execution profiles for a program comparable in size to *gcc*. The problem is only magnified if we consider more complex profiles, such as value or path profiles. A means for automatically checking profile data is necessary in order to test profilers thoroughly.

As a side note, our profile checker also helped to track down a non-profile-related problem in PLTO. The SPECfp-95 benchmark program *applu* was known to crash when optimized with PLTO, but tracking down the problem was proving difficult. It turned out that the process of verifying the profile data for this program uncovered some discrepancies that suggested a periodic corruption of memory during execution. This turned out to be a useful insight that helped focus our subsequent investigations into the problem and eventually led us to identify the actual cause.

Table 1 shows the execution time overhead incurred by profile verification relative to the time taken to profile the programs in the SPECint-95 benchmark suite using conventional instrumentation techniques. The “Normal” column gives the execution times for

```
#include <stdlib.h>
#include <stdio.h>
#define HASH_SIZE 1024
#define LOOP_COUNT 1000
#define MOD 2
struct hash_entry {
    int data;
    struct hash_entry *next;
};
static struct hash_entry *HashTable[ HASH_SIZE ];
static int collisions = 0;
int hash_value( int data ) {
    int x = (int) HashTable;
    return ((data * ((x / 0x100 ) % MOD)) ) % HASH_SIZE;
}
void hash_insert( int data ) {
    int hash = hash_value( data );
    struct hash_entry *entry = malloc( sizeof(struct hash_entry) );
    entry->data = data;
    entry->next = NULL;
    if ( HashTable[hash] != NULL ) {
        struct hash_entry *temp = HashTable[hash];

        collisions++;

        while ( temp->next != NULL ) {
            temp = temp->next;
        }
        temp->next = entry;
    } else {
        HashTable[hash] = entry;
    }
}

int main( int argc, char *argv[] ) {
    int i;

    for ( i = 0; i < LOOP_COUNT; i++ ) {
        hash_insert( i );
    }
    printf( "HashTable = %p\n", HashTable );
    printf( "collisions = %d\n", collisions );
    return 0;
}
```

Figure 2: A program whose behavior changes due to instrumentation

the benchmarks; the "Profile" column gives the execution times for instrumented versions of the programs that generate profiles; the "Verify" column gives the execution times to check the generated profiles. All programs in the same row had the same input data (the SPECint-2000 training input for that benchmark). Our experiments were run on an unloaded 2 GHz Pentium IV workstation with 1 GB of main memory, running Redhat Linux 7.3. The programs were compiled with *gcc* version 2.96 at optimization level `-O3` and statically linked, with additional flags to instruct the linker to retain relocation information.[3] It can be seen that verification incurs an overhead of roughly two orders of magnitude (about $600\times$ to $700\times$) relative to conventional profiling. While this slowdown is large, the verifier has to be run only once to check a profile. More importantly, our verification framework makes it possible to check large profiles generated by large programs—which is not possible using existing techniques—and to do so automatically.

6. Conclusions

While execution profiles play an increasingly important role in guiding compiler optimizations, the question of verifying that the profiles are correct—i.e., reflect the actual runtime behaviors of programs—has not received much attention. In practice, implementors use *ad hoc* techniques to test profilers, typically by inspecting the profiling code and checking its behavior manually on a few small programs where execution counts can be manually computed. This paper describes a framework for automatic verification of profiles. We illustrate our approach using a verifier for an edge profiler we have implemented in the context of the PLTO link-time optimizer. Using this verifier, we were able to identify several discrepancies in generated profiles that had gone undetected using conventional testing techniques.

[3] The requirement for statically linked executables is a result of the fact that *PLTO* relies on the presence of relocation information to distinguish addresses from data. The Unix linker `ld` refuses to retain relocation information for executables that are not statically linked.

7. REFERENCES

[1] P. Alken. `ald`: an assembly language debugger. `ftp://ftp.netbsd.org/pub/NetBSD/packages/pkgsrc/devel/ald/`.

[2] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *Proc. ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, pages 168–179, June 2001.

Program	*Execution Time (secs)*			*Slowdown*
	Normal (T_N)	Profile (T_P)	Verify (T_V)	T_V/T_P
compress	0.03	0.29	176	606.9
gcc	0.97	11.99	6774	565.0
go	0.47	3.64	2128	584.6
ijpeg	13.74	82.41	51310	622.6
li	0.11	1.67	1135	679.6
m88ksim	12.87	192.48	125839	653.8
perl	2.68	16.31	11427	700.6
vortex	2.23	16.61	11543	694.9
GEOMETRIC MEAN:				636.7

Table 1: Runtime overheads of profiling and verification

[3] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *Proc. 29th Annual International Symposium on Microarchitecture*, pages 46–57, 1996.

[5] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, vol. 1, March 1999.

[6] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software - Practice and Experience*, 21(12):1301–1321, 1991.

[7] W. Y. Chen, S. A. Mahlke, N. J. Warter, S. Anik, and W. W. Hwu. Profile-assisted instruction scheduling. *International Journal of Parallel Programming*, 22(2):151–181, April 1994.

[8] J. C. Cherniavsky. Validation through Testing. In *Proc. 11th International Conference on Software Engineering*, page 354, May 1989.

[9] R. Cohn and P. G. Lowney. Hot cold optimization of large Windows/NT applications. In *Proc. 29th Annual International Symposium on Microarchitecture*, pages 80–89, December 1996.

[10] R. Cohn and P. G. Lowney. Design and analysis of profile-based optimization in Compaq's compilation tools for Alpha. *Journal of Instruction Level Parallelism*, vol. 2, May 2000.

[11] S. K. Debray and W. Evans. Profile-guided code compression. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI-02)*, pages 95–105, June 2002.

[12] S. L. Graham, P. B. Kessler, and M. K. McKusick. `gprof`: A call graph execution profiler. In *Proc. ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, June 1982.

[13] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Proc. Fourth Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[14] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proc. ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994.

[15] W. Howden. *Software Engineering and Technology: Functional Program Testing*. McGraw-Hill, 1987.

[16] J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software—Practice and Experience*, 24(2):197–218, February 1994.

[17] J. Laski. Testing in the Program Development Cycle. *Software Engineering Journal*, 4(2):95–106, March 1989.

[18] E. Mehofer and B. Scholz. A novel probabilistic data flow framework. *Lecture Notes in Computer Science*, 2027, 2001.

[19] R. Muth, S. Watterson, and S. K. Debray. Code specialization based on value profiles. In *Proc. 7th. International Static Analysis Symposium (SAS 2000)*, pages 340–359, June 2000.

[20] K. Pettis and R. C. Hansen. Profile-guided code positioning. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[21] A. Podgurski, C. Yang, and W. Masri. Partition Testing, Stratified Sampling, and Cluster Analysis. In *Proc. ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, pages 169–181, December 1993.

[22] J. B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, 1996.

[23] B. Schwarz, S. K. Debray, and G. R. Andrews. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

[24] B. Schwarz, S. K. Debray, and G. R. Andrews. Disassembly of executable code revisited. In *Proc. IEEE 2002 Working Conference on Reverse Engineering (WCRE)*, October 2002. (To appear).

[25] A. Srivastava and A. Eustace. ATOM—A system for building customized program analysis tools. In *Proc. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, June 1994.

[26] S. Watterson and S. K. Debray. Goal-directed value profiling. In *Proc. Tenth International Conference on Compiler Construction (CC 2001)*, April 2001.