2018

Evaluation of Dynamic Binary Instrumentation Approaches: Dynamic Binary Translation vs. Dynamic Probe Injection

Valerie Zhao
vzhao@wellesley.edu

Evaluation of Dynamic Binary Instrumentation Approaches: Dynamic Binary Translation vs. Dynamic Probe Injection

Valerie Zhao

Submitted in Partial Fulfillment
of the
Prerequisite for Honors
in Computer Science
under the advisement of Benjamin P. Wood

May 2018

Abstract

From web browsing to bank transactions, to data analysis and robot automation, just about any task necessitates or benefits from the use of software. Ensuring a piece of software to be effective requires profiling the program's behavior to evaluate its performance, debugging the program to fix incorrect behaviors, and examining the program to detect security flaws. These tasks are made possible by instrumentation—the method of inserting code into a program to collect data about its behavior. *Dynamic binary instrumentation* (DBI) enables programmers to understand and reason about program behavior by inserting code into a binary during run time to collect relevant data, and is more flexible than static or source-code instrumentation, but incurs run-time overhead. This thesis attempts to extend the preexisting characterization of the tradeoffs between *dynamic binary translation* (DBT) and *dynamic probe injection* (DPI), two popular DBI approaches, using Pin and LiteInst as sample frameworks. It also describes extensions to the LiteInst framework that enable it to instrument function exits correctly. This evaluation involved using the two frameworks to instrument a set of SPEC CPU 2006 benchmarks for counting function entries alone, counting both function entries and exits, or dynamically generating a call graph. On these instrumentation tasks, Pin performed close to native binary time while LiteInst performed significantly slower. Exceptions to this observation, and analysis of the probe types used by LiteInst, suggest that LiteInst incurs significant overhead when executing a large number of probes.

Acknowledgments

First and foremost, I would like to express my sincerest gratitude to my advisor, Professor Benjamin Wood. Despite being on sabbatical for the year, he had graciously agreed to advise me on this thesis. I could not have arrived at this point without his guidance and support for this project, academia, and everything else in life (as well as gems like "'work' is also unitless and continuous (not discrete) like 'happiness…' ;)").

Next, I would like to thank my thesis committee, Professors Lyn Turbak, Ashley DeFlumere, & Robbie Berg, for taking an interest in this work and putting in the time to provide helpful feedback. I would also like to thank the authors of LiteInst, Buddhika Chamith and Professor Ryan Newton, for providing the source code to LiteInst and answering my questions.

I am very grateful for all my friends—from Wushu and NEUR 100 and other places—for the fun times we have shared, the support when I encountered problems, and the understanding when I isolated myself and crawled into the hole that is my dorm room to work on this thesis for many moons. I am also thankful for everyone at the Computer Science department who has taught me so much and supported my endeavors.

Lastly, I am thankful for my family, for supporting me in my academic pursuits and always having my best interest at heart.

Contents

1 Introduction

In software development, programmers often want to examine program behavior for information such as potential bugs, security vulnerabilities, and performance bottlenecks. This can be achieved by adding more code into the program to measure aspects of its run-time behavior – a process known as *instrumentation*. Depending on usage and preference, instrumentation can be done on the source code or binary executable level, and statically or during run time.

Source code instrumentation can be useful for manipulating programs directly to evaluate expected program behaviors, but is unportable across source languages, inefficient when massive amounts of code are present to examine, and impossible when the source code is not available. Binary instrumentation addresses these issues, but is less accessible when reasoning about high-level semantics, and thus requires a robust framework to abstract the reasoning away in this type of situations.

Static binary instrumentation allows programmers to determine their instrumentation tools, identify locations for instrumentation, and insert instrumentations before running the program, and is therefore relatively easy and robust. It is, however, not as flexible as *dynamic binary instrumentation* (DBI), which is the method of dynamically adding code to a binary to understand its run time behavior. DBI enables programmers to examine the run-time behavior of binaries, and to which they can adapt instrumentations, but DBI's dynamic nature can result in poor performance. Therefore, optimizing DBI performance can benefit the programming experience.

There are two common approaches to DBI. *Dynamic binary translation* (DBT), used by frameworks like Intel Pin [23], involves copying the target binary in segments and instrumenting the copies during runtime. *Dynamic probe injection* (DPI), used by frameworks like LiteInst [14], manipulates the target binary in memory to insert probes that lead execution to the instrumentation. Although prior literature has evaluated the performance of DBT frameworks and, at times, compared the two approaches for specific purposes (such as instrumenting kernels [26]), more work is needed to characterize their performance with a broader scope in mind.

1.1 Problem Statement

DBI incurs performance overhead from carrying out much of its work during run time. Insight for optimizing this technique can be gained from comparing its two main approaches, DBT and DPI. DBT enables one to instrument any part of a binary, but can incur extra performance overhead from copying the binary. DPI bypasses copying overhead by instrumenting the binary directly, but can incur extra performance overhead from preserving target program behavior while inserting probes, as well as costs from context-switching and redirections of execution by the probes.

Prior work has evaluated several DBI frameworks. The most direct evaluation of DBI approaches to date evaluated only DBT frameworks (specifically Pin, DynamoRIO, and Valgrind) [27]. Application-specific evaluations are scattered across papers targeting specific use cases; for example, the idea that DBT offers better performance than the traditional DPI usage for instrumenting operating systems served to introduce the JIFL ("JIT Instrumentation Framework for Linux") prototype [26]. Part of the reason for this gap in the literature may be that these frameworks were designed with different purposes in mind. DBT frameworks tend to be used to create heavyweight tools, which carry out complicated analyses that requires dense instrumentation at a fine granularity of code.[1] DPI frameworks support lightweight tools such as profilers, which carry out simpler analyses and simpler data management [25]. Since these two types of DBI frameworks were designed with different workloads and goals in mind, it is likely that they are optimized for their respective goals while still supporting (but perhaps not optimized for) other use cases.

This thesis seeks to evaluate Pin ([23], a DBT framework) and LiteInst ([14], a DPI framework) on a specific set of analysis tasks and benchmarks. While Pin is able to support a variety of instrumentation granularities, such as at beginnings and ends of procedures (procedure boundary) and basic blocks (basic block boundary), LiteInst is able to instrument only procedure boundaries at the moment, with partial support for other levels of granularities. This limits the analysis tasks to those that consider only procedure boundaries; future work can focus on instrumentation of other granularities for analysis tasks that are more complex. Nonetheless, results from this evaluation can help to guide further exploration of the kind of usage that would best benefit from the two DBI approaches, and hint at areas where further optimization can be made. In addition, they may offer insights into future development of a hybrid approach that could combine the fast startup of DPI with the flexibility of DBT.

1.2 Contributions

In this thesis, I make the following contributions.

[1]Valgrind [25] is explicitly designed for this usage.

- I extended the LiteInst framework to effectively support procedure boundary instrumentation (instrumenting at the beginning and end of a procedure).

- I built tools with LiteInst and Pin to evaluate the two frameworks. For each framework, I built a procedure counter to count the number of entries for each procedure during an execution; an entry-exit counter, which performs the same task as the procedure counter while also counting number of exits for each procedure; and a call graph generator, which generates a call graph dynamically.

- I analyzed the performance of the tools on two timing metrics: total elapsed time (from when the instrumentation is started up and begins, to the tear-down phase when memory is freed after the instrumented binary has finished running) and isolated run time of the instrumented binary (in-between the start-up and tear-down). My experimental results showed that for most benchmarks, Pin added little overhead to the target binary, while LiteInst added little overhead to the target binary when simply instrumenting their procedure entries, but contributed significantly more overhead when instrumenting procedure exits in addition to entries. For the two slowest benchmarks, however, both Pin and LiteInst performed almost just as well as the native binary for all three pairs of tools.

- I characterized sources of performance overheads in LiteInst by analyzing the probes it executed during run time. An analysis on the probe types used by LiteInst showed that LiteInst's significant overhead on the other benchmarks may be due to the large number of executed probes.

1.3 Outline

This work is presented as follows.

- Chapter 2 gives an overview of the background behind instrumentation, with a focus on DBI, Pin, and LiteInst.

- Chapter 3 examines the complexity behind instrumenting procedure exits with DPI, and describes my extension to the LiteInst framework for accurate procedure exit instrumentation.

- Chapter 4 describes the analysis tools implemented as subjects of the evaluation.

- Chapter 5 describes the evaluation and results.

- Chapter 6 proposes directions for future work.
- Chapter 7 concludes this thesis.

2 Background

2.1 Instrumentation

Suppose that, for debugging purposes, a programmer would like to determine where a function call was made in a program execution. Having a call graph, in which vertices are functions and direct edges point from one function to a function they call, would be helpful in this situation. To construct such a graph, the programmer could manually simulate the execution with expected input values—but this is tedious, fallible, and difficult to generalize to all programs. They could go through every single function in the source code and write a print statement at the beginning of each one, to print out the name of each function as they are being called—but manual and mundane labor is still required. Is there a method that would enable them to understand the program behavior in an accurate and efficient manner?

This is a common question to encounter in the software engineering process. Even after a piece of software has been written, ensuring it to be effective, secure, and correct requires profiling the program's behavior to evaluate its performance, debugging the program to fix incorrect behaviors, and examining the program to detect security flaws. These tasks, and others that demand understanding and reasoning about how the program will run, can be made possible by instrumentation—the method of inserting code into a program to collect data about its behavior. Multiple approaches to instrumentation exist to help programmers avoid menial tasks such as those presented above. For the purpose of minimizing confusion, this thesis will refer to any functions appearing in the target program (which is to be instrumented) as "procedures" from now on, and continue to refer to functions in the instrumentation frameworks and tools as "functions," even though the two terms are typically used interchangeably.

2.2 Some Approaches to Instrumentation

2.2.1 Source Code Instrumentation

The programmer can choose to instrument the source code, the binary, or both. Source code instrumentation (static or dynamic) is used for many languages and for various purposes, such as security verifications for C [22], query-based instrumentation for C++ [19], and concurrent program behavior analyses for Java [8]. There are also frameworks that aim to handle multiple languages, such as GEMS [16], which has language-specific parsers to parse the program into a common model, instrument it, and then pretty-print it back to the original language.

Source code instrumentation is particularly useful in cases such as expectation testing [1], in which the tester instruments the software to compare its executed behavior with expected behavior under specific scenarios. For other use cases, however, it has several disadvantages. Its language-specificity and source requirement make library instrumentation particularly difficult, as libraries contain a large amount of code that may also be written in multiple languages. For programs whose sources are unavailable, such as untrusted software, source code instrumentation is entirely impossible. In these cases, a more appropriate approach is binary instrumentation.

2.2.2 Static Binary Instrumentation

In binary instrumentation, instrumentation is done to the binary executable rather than the source code. Although binary instrumentation has the complication of being architecture-specific, it bypasses the source code requirement, and handles complex libraries with relative ease. One can choose to instrument the program statically or dynamically. Prior work related to *static binary instrumentation* (SBI) includes ATOM [28], an early implementation based solely on the now defunct Alpha platform; MIL [15], a language that enables the collection of program behavior information for SBI; PEBIL [21], or "PMaC's Efficient Binary Instrumentation Toolkit for Linux"; and P𝕊I [30], a platform that specifically targets security instrumentation.

The static aspect of SBI offers several advantages over dynamic techniques. Instrumenting the program is easier and more robust due to SBI's static nature; and because the instrumentation is completed before run time, this same instrumentation can be used for multiple program executions without having to instrument during every execution, thus eliminating some run-time overhead from the instrumentation process. For counting basic block in executions, PEBIL outperformed several dynamic binary instrumentation frameworks in terms of execution time [21], namely Valgrind [25], DynamoRIO [9, 10], Pin [23], and Dyninst [11] (in decreasing order of run-time overhead). For instrumentation that aids program analyses for security purposes, P𝕊I is able to achieve less overhead than dynamic binary instrumentation frameworks for real-world applications [30]. In addition, P𝕊I retains qualities that are necessary for secure instrumentation but difficult to find in traditional SBI frameworks, namely completeness (all executed code can be instrumented) and non-bypassability (the instrumented code cannot bypass or subvert instrumentation)—qualities that had historically made dynamic binary instrumentation frameworks the favorable choice for secure instrumentation [30].

Despite its advantages in some aspects, SBI lacks the the run-time flexibility that dynamic methods have. The binary must be compiled after static instrumentation, and so it must be compiled multiple times if different instrumentations are needed, which is inconvenient. Therefore, *dynamic binary instrumentation* (DBI) can be desirable in certain use cases, as described in the next section.

2.3 Dynamic Binary Instrumentation

DBI instruments programs during run-time, and thus allows changes to be made dynamically, including toggling between instrumenting or ignoring certain parts of the code. It has been used for purposes such as malware defense [2], classical virtualization of the x86 ISA [4], and demand-driven structural testing [24]. There are currently two main approaches to DBI: *dynamic binary translation* (DBT, also known as the JIT-based approach to DBI) and *dynamic probe injection* (DPI, or the probe-based approach).

2.4 Dynamic Binary Translation

In general, DBT incrementally recompiles and executes code, by copying the compiled code, instrumenting the copies, and then executing the instrumented copy. A JIT compiler incrementally finds code to copy, and the instrumented copies are linked together to form the execution path. Frameworks such as Pin [23], DynamoRIO [9], and Valgrind [25] are among those that utilize this approach. DBT enables one to instrument any part of a program with relative ease, since the JIT compiler will ultimately recompile (in increments) a copy of the entire program, during which it will ensure that the instrumentation functions fit into the copies. The compiler can also optimize the instrumentation functions during the recompilation process, such as inlining them into the copies. However, DBT may incur overhead in the process of creating and managing the copies. Pin is particularly popular and has been the target of extensions such as the timing-sensitive Dime* [5].

2.4.1 Intel Pin

Intel Pin, or simply "Pin," is a DBT framework that supports run-time instrumentation for Linux [23]. Since its academic publication in 2005, Pin has been cited 1,027 times (according to the ACM Digital Library [3]), sometimes as a reference point for performance evaluation of new DBT frameworks. It is also under consistent development by the Intel Corporation. For these reasons, Pin seemed to be a great choice for evaluation.

To use Pin, the programmer writes instrumentation tools, or “Pintools,” that specify where and how the program will be instrumented. For example, if the programmer wants to count the number of times procedure `foo()` is called, they should to define a function `forFoo()` in the tool. In this `forFoo()`, the programmer should utilize Pin’s API to specify that the instrumentation function, `incrementFooCounter()`, will be called at the beginning of `foo()`. Then, the programmer should register `forFoo()` with Pin in the `main()` function of the Pintool, so that Pin can instrument the program.

Pin and Pintools work together to instrument the target program. While Pin carries out the JIT-compilation and instrumentation, it loads the Pintool for directions on the instrumentation. The program is instrumented one trace at a time, where a “trace” is a series of instructions that would be executed sequentially, until termination by one of the following:

- An unconditional control transfer: branch (direct and indirect jump instructions), call, or return.
- A pre-defined number of conditional control transfers, such as the `JE` instruction (“jump if equal to”) in x86.
- A pre-defined number of instructions, which have been included in the trace without encountering the above two conditions.

The instrumentation process is as follows.

- Before executing the target program’s first instruction, Pin generates a copy of the program’s starting trace that, when executed, will return control to Pin after branching.
- Pin transfers control to the copy it created, which is now executing.
- After the copy has exited execution and returned control to Pin, Pin generates a copy for the next trace, instruments it, and executes the copy.
- The cycle continues: as more code is discovered, Pin makes a copy of the trace and instruments the copy while also making sure that the copy will return control to Pin after the trace ends.

- The instrumented copies are stored in a code cache so that they can be reused for similar traces in the future.

With the help of the instrumentation paradigm above (summarized in a high-level manner in Figure 1), Pin offers some advantages over other DBT frameworks, including optimizations to the JIT compilation process (through register reallocation, code inlining, liveness analysis, and instruction scheduling).

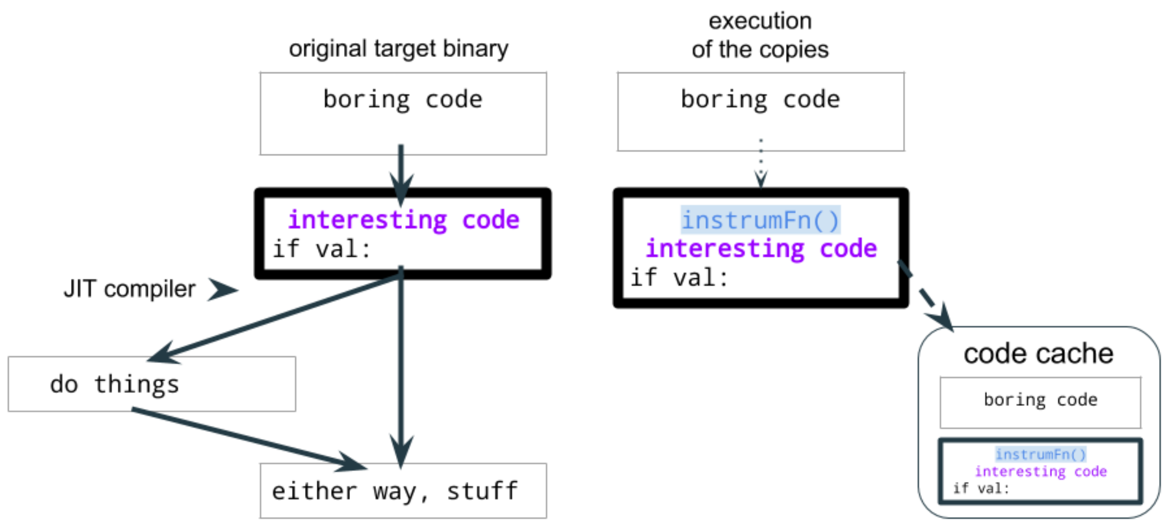


Figure 1: *High-level diagram of Intel Pin, a framework for dynamic binary translation (DBT). The original binary is represented by a control-flow graph. The interesting code (in purple) is the target code for instrumentation. The JIT compiler arrow points past the block it has just discovered in the original target binary. Blocks with dark outlines are associated with the block that is currently being copied (left), instrumented (middle), and stored in the code cache (bottom right).*

Nonetheless, because Pin still copies the original binary as DBT frameworks do, it may retain some overhead in managing these copies and instrumenting them rather than the original code. Although a side-by-side comparison of pre- and post-instrumented binary with Pin could be helpful for further analyzing the instrumentation process and potential overhead of the copying process, it is not presented in this thesis because post-instrumented copies of a binary are not accessible through Pin's API (to the best of my knowledge).

2.5 Dynamic Probe Injection

On the other hand, the DPI approach edits the code directly in memory while it is running. It replaces instructions that one would like to instrument with probes, usually jump or trap instructions, that lead program execution to the instrumentation code. “Probe sites” refer to the locations in the code where target instructions for instrumentation reside. Once the instrumentation code (such as an increment routine for counting number of procedures) for a particular probe has been executed, control flow returns to the instruction after the probe.[2] Frameworks that use DPI include LiteInst [14]; Dyninst [11] and Vulcan [29], which are two frameworks that support both SBI and DBI; and DTrace [12], which relies on trap instructions to emulate the behavior of the instrumented targets.

The choice to insert code directly into the binary, rather than a copy, has its trade-offs. It may shorten the setup time by bypassing the copying step, but can suffer from a limited set of possible probe sites. Since probes may be longer than the instructions they aim to replace, their insertion can overwrite neighboring instructions in addition to the target instructions. In a variable-length ISA, such as x86, each probe site would require a customized probe to take into account the target instruction’s size. Some DPI paradigms choose to use probes constructed from trap instructions rather than other control flow instructions, like jump instructions. These can incur performance overhead, as the kernel must respond to the trap, and then save context before directing control back to the trap signal handler in user space. Transfers between kernel and user space have high cost.

2.5.1 LiteInst

LiteInst [14], which is currently available only for x86-64, is a framework that follows the DPI approach by implementing the instruction punning algorithm (see Figure 2). It replaces target instructions with probes, which lead the execution to segments of code called trampolines. Trampolines save the context before calling instrumentation functions and executing replaced instructions.

LiteInst is an appropriate DPI framework for evaluation because it is one of the newest advances for this DBI approach (having been published in June 2017). It enables probes to be inserted virtually anywhere (through the instruction punning algorithm and various contingency plans, as described in the next section), and—at the time of writing—had not

[2]It should be noted that since most modern operating systems forbid changes to code-containing pages of the memory by setting their permissions as no-modify by default, DPI frameworks must change virtual memory mapping permissions in order to manipulate binaries in memory.

yet been evaluated against Pin through timing measurements.

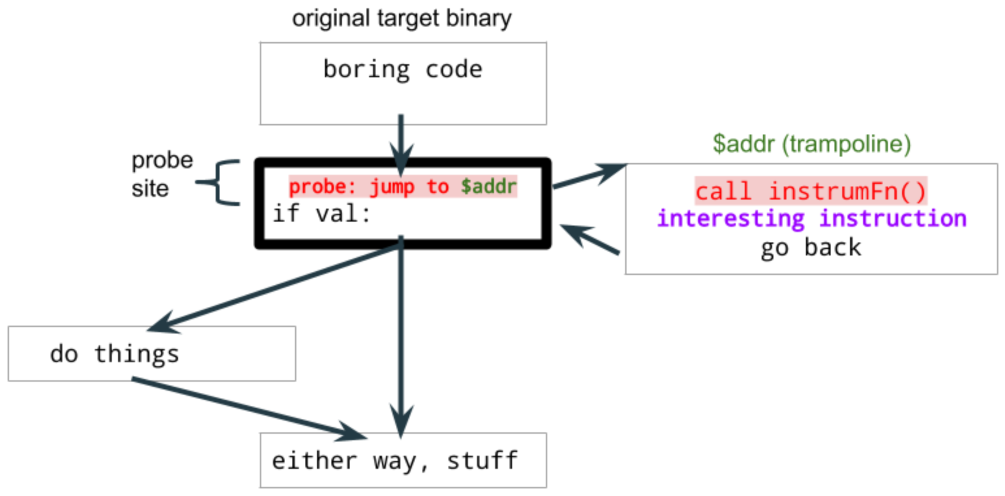


Figure 2: *High-level diagram of LiteInst, a framework for dynamic probe injection (DPI). The original binary is represented by a control-flow graph on the left, and the block with dark outline is currently being executed. The interesting instruction (in purple) is the target instruction, displaced from the original binary.*

For example, consider a target procedure `check_legal()` from the `sjeng` benchmark of SPEC CPU2006 benchmark suite [20], which is shown in Listing 1. Entry and exit instrumentation (instrumentation for the beginning and end of a procedure) is inserted into this procedure, and the results are shown in Listing 2.[3] The beginning two instructions of `check_legal()` in the original binary (Listing 1) have been replaced by a `jmpq` instruction in the instrumented version (Listing 2). This `jmpq` instruction jumps to the trampoline at address 0x62465cb8 for the entry instrumentation function. The `retq` instruction at the end of the original procedure is replaced with an illegal instruction at line <+5510>, which raises a SIGILL. LiteInst has a special handler that receives this SIGILL and then directs execution to the trampoline. The next section describes LiteInst's usage and illegal instruction probes in more detail.

[3]`sjeng` is a program that plays chess, and its variants, against a human player or input file using artificial intelligence techniques.

```
check_legal():
0x405cad <+0>:    push %rbp
0x405cae <+1>:    mov  %rsp,%rbp
0x405cb1 <+4>:    sub  $0x20,%rsp
// ...
0x407233 <+5510>: retq
```

Listing 1: *Beginning and end of `check_legal()` in the original `sjeng` binary.*

```
check_legal():
0x405cad <+0>:    jmpq 0x62465cb8

0x405cb2 <+5>:    sub  $0x20,%esp
// ...
0x407233 <+5510>: <illegal instruction>
```

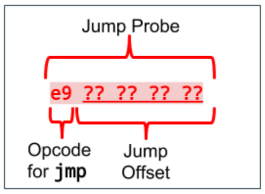
Listing 2: *Beginning and end of `check_legal()` after instrumentation.*

2.5.1.1 Instruction Punning & Alternatives

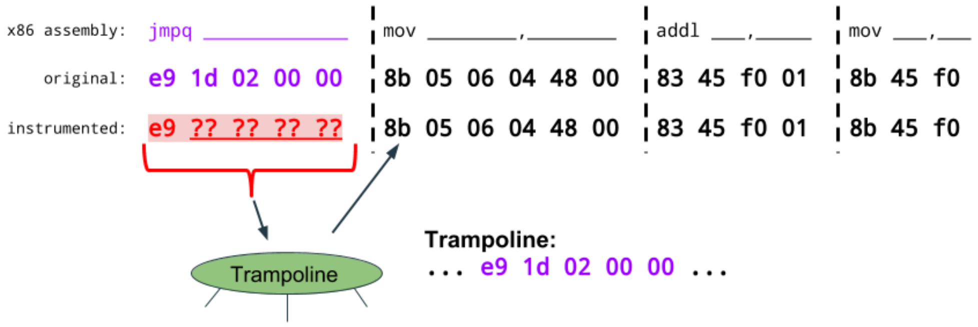
LiteInst attempts to address the probe suitability issue with the instruction punning algorithm [14]. This algorithm uses jump instructions as probes, and the probes lead execution to corresponding trampolines. Each jump probe has 5 bytes: 1 byte for the opcode of (relative) `jmp`, and 4 bytes to encode the jump offset that leads to a trampoline. Each probe is inserted at the head of an instruction in the original binary.

Probes should not affect the target binary other than the instructions being instrumented, which are the target instructions replaced by the probes. This is indeed the case if the replaced instructions are longer than the probes (by having 5 or more bytes), since execution can return from the instrumentation function to anywhere following the probes (Figure 3). In this ideal scenario, the probe jump target can be any mappable address for the trampoline. However, if the replaced bytes belong to more than just the target instruction (in other words, the probe cannot avoid replacing more than just the target instruction), the probe may affect neighbor instructions, resulting in unexpected behavior (Figure 4a). The instruction punning algorithm addresses this issue by tailoring the probe jump offset to resemble the bytes of the neighbor instructions as closely as reasonable (like a "pun"), thereby minimizing unnecessary changes to the original program (Figure 4b).

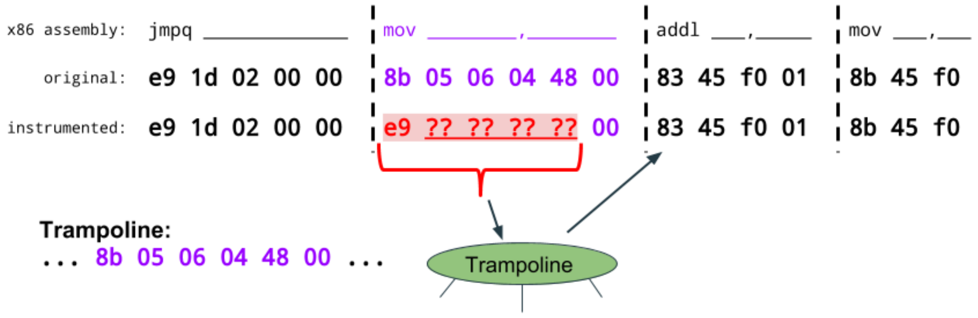
Instruction punning is not always enough. In the worst case scenario, all 5 bytes replaced by a probe may belong to different 1-byte instructions (Figure 5a). Attempting to form a probe from these 5 bytes will be problematic if the constructed jump target is not a mappable address. LiteInst accounts for this scenario by inserting the 1-byte opcode of `int3` trap or some illegal instruction into any of the bytes where the jump target would be. A special signal handler is used to continue execution after receiving SIGTRAP or SIGILL from the 1-byte backup probe. This helps expand the possible set of valid trampoline addresses. Figure 6a is a pictorial depiction of the execution path triggered by the 1-byte probe.



(a) *The composition of a jump probe in LiteInst.*

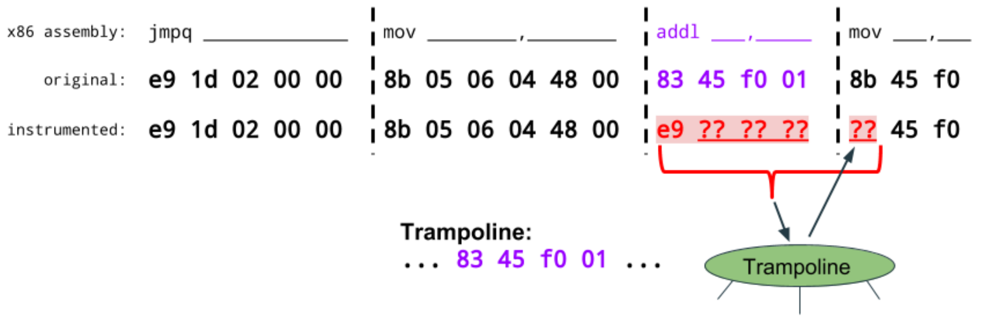


(b) *The target instruction (in purple) is successfully replaced by the jump probe (in red), and execution will be directed to the trampoline and then return to the next instruction in the original binary (the first* `mov` *instruction), as indicated by the arrows.*

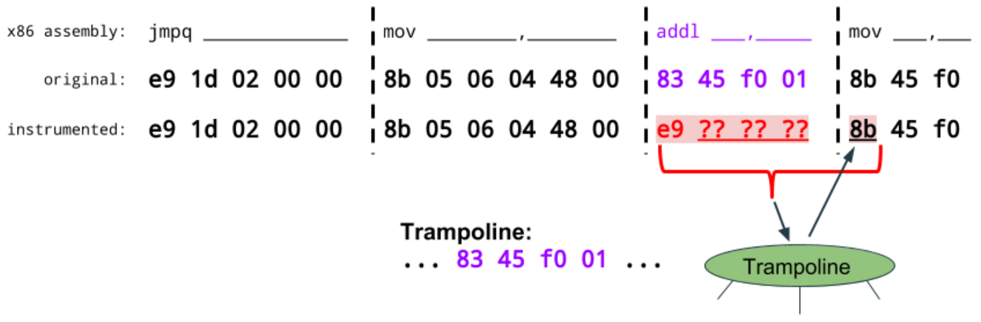


(c) *Similar to (b), the 5-byte jump probe (red) replaces part of the 6-byte target instruction (purple). Arrows depict the rest of the control flow.*

Figure 3: *An example of successful instrumentation by jump probes.*

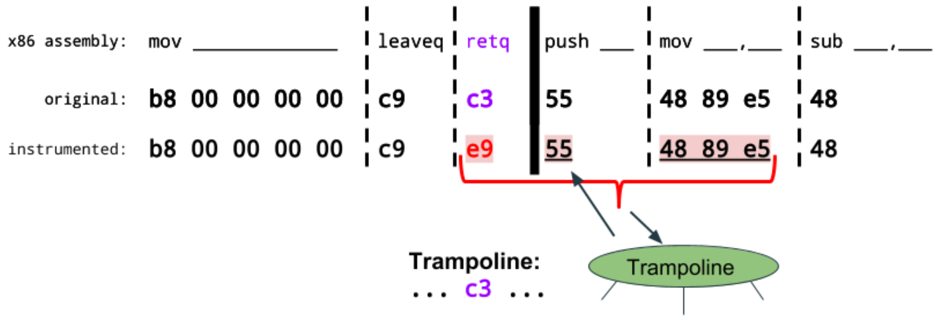


(a) *The target instruction (in purple) and the head of the next instruction are replaced by the jump probe (in red).*

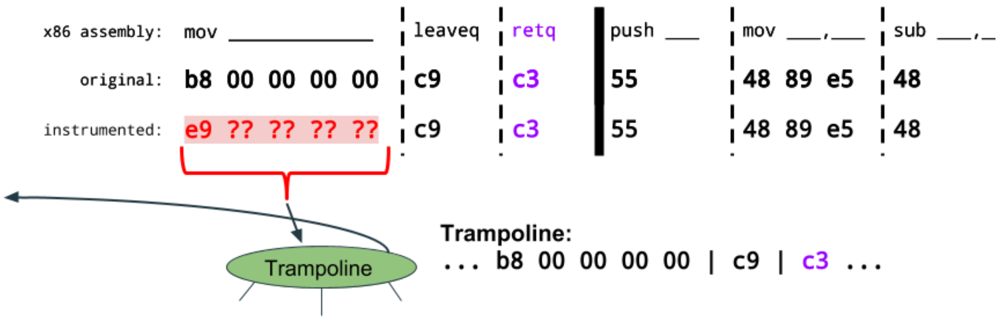


(b) *A probe that contains the same byte at its end as the head of the next instruction (*`0x8b`* in this case, in black text and highlighted in red) is used.*

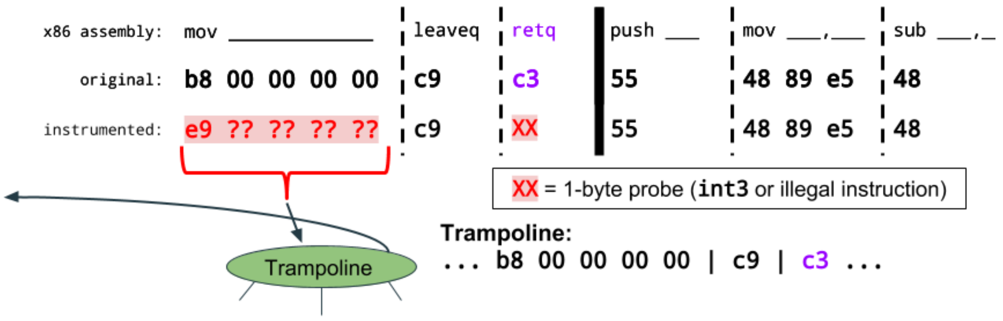
Figure 4: *An example of Instruction Punning.*



(a) *A scenario in which only one pun (black text highlighted in red) is available for the jump probe (in red).*



(b) *LiteInst addresses the problem in (a) by backtracking, in which it finds a location upstream to place the probe instead.*



(c) *In addition to the upstream probe, a 1-byte trap or illegal probe (red "XX") must be placed at the head of the actual target instruction.*

Figure 5: *An example of backtracking.*

Another issue for instruction punning arises from instrumenting instructions at the end of a basic block. For example, procedures tend to exit with a `retq` at the end, which is only 1 byte and ends a basic block (Figure 5a). To instrument this `retq` using the standard paradigm in LiteInst would require a 5-byte probe to lie across the basic block boundary, starting at where `retq` resides. This can have unexpected complications. Single-byte `int3` or illegal instruction probes can solve this, but the signals they trigger are costly. In this case, LiteInst “backtracks” along the instruction stream, to find a more suitable probe site somewhere upstream in the same basic block, relying on the assumption that all instructions of a basic block are executed if any instruction in it is (Figure 5b). This is referred to as “backtracking.” The ideal new probe site would be located at a 5-byte instruction, or on a set of bytes that can be punned for valid trampoline addresses. In either case, there may be branches in the program that reaches the `retq` by skipping over the new probe. Thus, the new probe site cannot overlap with this `retq`, and `retq` will be replaced by a trap leading to the same trampoline as the new probe (Figure 5c). This arrangement is more efficient than not backtracking and only replacing `retq` with the signal-inducing probe, because it redirects some of the branched traffic to the new jump probe, and jumps cost less than signals. A code example can be found in Listing 5 of Section 3, where `mov $0x0, %eax` at <+5504> (which is 5 bytes) and `retq` at <+5510> have been replaced by a jump and an illegal instruction, respectively. In the event of a failed backtracking, LiteInst resorts to trap-based probes (`int3` or one of the 1-byte illegal instructions) that pass signals to LiteInst’s special signal handler, which directs execution to the trampoline elsewhere.

Because the jump offsets in the probes must lead execution to valid memory locations (which may also require additional memory allocation and consequently must be mappable), probe insertion incurs the most performance cost from searching for puns that lead to valid jump addresses. Several scenarios may alleviate or contribute to this overhead:

- For a probe that is relatively unconstrained by the punning aspect (because it replaces very few instructions, perhaps just one), the search space for an optimal offset can be large.
- For a probe that has few pun choices (such as one that replaces 5 1-byte instructions, which yields only one possible standard pun offset), LiteInst may have to resort to an alternative, potentially more expensive probe, such as a trap or illegal instruction.
- To optimize for space, the searching algorithm attempts to group trampolines into previously allocated trampoline pages.

- Special precautions, including probe site relocation (through backtracking) and trap instructions, are needed for probe sites that may require their probes to lie across basic block boundaries, as this can corrupt instruction decoding during the rewrite process otherwise.

Interactions between these scenarios can have interesting effects on LiteInst's performance. While LiteInst has been evaluated on some benchmarks [14], more exploration of its performance can be done by comparing it to DBT frameworks such as Pin.

3 Accurate Procedure Exit Instrumentation for LiteInst

In DPI, the direct manipulation of binary executables poses an important challenge in customizing and handling the probes to ensure the correct changes to control flow. Even though the instrumentation is dynamic, the process of identifying probe sites is static, and thus complexities exist.

In the simple scenario of instrumenting procedure entries and exits (when procedure invocations begin and end, respectively, during run time), most entry instrumentations are relatively simple, since most procedures have some sort of prologue before the body. The framework just has to replace the first instruction of each procedure with a jump instruction to the trampoline. Even instrumenting cases where the beginning basic block is part of a loop is relatively straightforward.

On the other hand, exit instrumentation is more complex, because there are several ways for a procedure to end. This chapter will describe the complexity in designing accurate procedure exit instrumentation, and discuss LiteInst's partial implementation in this domain, extended by this thesis. This chapter first discusses procedure returns in Section 3.1, then jump calls in Section 3.2, and lastly halts and `exit()` in Section 3.3.

3.1 Returns (`retq`)

Procedures most commonly exit by returning to where they were called (specifically, the instruction right after the call instruction that called them) with `retq` instructions. The simplest case studies are procedures that contain only one `retq`, at its end. For example, LiteInst can easily identify the `retq` byte at `increment`<+20> (Listing 3) as a probe site for exit instrumentation.

One problem that is universal to all instrumentation of `retq` instructions arises when the standard probe occupies more than 1 byte, since `retq` is only 1 byte long. To avoid replacing or otherwise affecting instructions next to this `retq`, special care is needed to instrument such probe sites. As described in Section 2.5.1.1, LiteInst addresses this issue by either backtracking to find a more suitable probe site and replacing `retq` with a `int3` or an illegal instruction, or simply perform the latter without the backtracking.

```
increment():
0x400636 <+0>:  push   %rbp
0x400637 <+1>:  mov    %rsp,%rbp
0x40063a <+4>:  mov    %edi,-0x14(%rbp)
0x40063d <+7>:  mov    -0x14(%rbp),%eax
0x400640 <+10>: add    $0x1,%eax
0x400643 <+13>: mov    %eax,-0x4(%rbp)
0x400646 <+16>: mov    -0x4(%rbp),%eax
0x400649 <+19>: pop    %rbp
0x40064a <+20>: retq
```

Listing 3: *The procedure* `increment()`*, which increments its argument and returns the new value.*

3.1.1 Returns, Branching, and Shared Trampolines

Some procedures have multiple return instructions, by splitting off into branches and returning at the end of one branch. Their exits are also easy to identify (by looking for the `retq` opcode), and a DPI framework should replace all `retq` instructions with exit instrumentation probes.

For procedures whose branches split and then join together before returning, exit instrumentation can be more difficult. If the control-flow join point is too close to the `retq`, such that the basic block in-between is too small, there may not be enough space between the join point and `retq` to fit in the exit probe. In these cases, the instrumentation framework must take care to ensure that both the join point and the return are eventually executed the right number of times.

3.1.2 LiteInst Mechanism and Extension

In LiteInst, one example of a small return basic block, as a result of a close branch join point, is in `check_legal()` (Listings 4 and 5). These two listings show a more comprehensive prologue and epilogue of the procedure than Listing 2 does. Recall that due to the size of `retq`, LiteInst backtracks to instrument this exit, which results in the `jmp` probe at <+5504> as well as the illegal instruction at <+5510> (as described in Section 2.5.1.1).

Based on these two listings, there are two paths to reach the exit of the procedure. Path A goes through the instruction at <+5504>, and path B skips that instruction by taking the jump from <+5502> to <+5509>. The probe for path A is the `jmpq` instruction at <+5504>, while the probe for path B is the illegal instruction at the end. It is straightforward for the framework to assign a unique trampoline for each possible path of exit—in this case, one for path A and one for path B.

When taking path A, execution jumps to the trampoline at address 0x8000073e (Listing 6), which should execute the instruction replaced by the probe (`mov $0x0,%eax`), execute the `leaveq` that had been skipped over in the original binary, switch context, call the exit instrumentation function, restore context, and then return on behalf of `check_legal()`. When taking path B, the illegal instruction will be executed, thus passing a SIGILL to LiteInst's signal handler, which then dispatches to the appropriate trampoline. This trampoline should switch context, call the exit instrumentation function, restore context, and then return on behalf of `check_legal()`.

Note that there is overlap between the two trampolines described in the prior paragraph, with the only difference being that path A's trampoline must execute `mov $0x0,%eax` and the skipped `leaveq`. Therefore, this is an opportunity for optimization by having the exit probes of both paths share the same trampoline (Listing 6), where path A's jump probe would enter the trampoline at the `mov $0x0,%eax` instruction, and path B's illegal instruction probe would enter at the instruction right after `leaveq` (since `leaveq` would have been executed already before the trampoline is reached in path B). Figures 6b and 6c are pictorial depictions of path A and path B, respectively.

The LiteInst implementation had largely employed this optimization, but did not account for all instances of procedure exits. Listing 6 shows the trampoline for when `check_legal()` exits. Instead of jumping to the instruction right after `leaveq` (address 0x80000744), path B's illegal instruction probe would jump to the `retq` at the bottom (address 0x800007b0) instead. This prevented the exit instrumentation function from being called (in code starting at 0x80000782). Consequently, for every time `check_legal` exits in `sjeng` through path B, LiteInst skips one count of exit. For a program made up of many procedures, or procedures that are called often, this problem can result in many missed procedure exits. Vetting the logic and isolating this issue was a lengthy and complicated process.

I extended the framework to ensure the correct behavior in this scenario by having the signal handler dispatch to the instruction right after the `leaveq` in the trampoline, at address 0x80000744. This ensures that everything after `leaveq` will be executed, including the exit instrumentation function. Figure 6c depicts the original issue and my fix.

3.2 Jumps (jmp)

In addition to returns, procedures can exit through jump instructions (`jmp`) due to tail-call optimizations. A procedure can jump to another procedure as its last step, and thus simul-

```
check_legal():
0x405cad <+0>:    push   %rbp
0x405cae <+1>:    mov    %rsp,%rbp
0x405cb1 <+4>:    sub    $0x20,%rsp
// ...
// splits into two paths:
0x405e7f <+466>:  jne    0x40722d <check_legal+5504>
// ...
0x40722b <+5502>: jmp    0x407232 <check_legal+5509>
0x40722d <+5504>: mov    $0x0,%eax
0x407232 <+5509>: leaveq
0x407233 <+5510>: retq
```

Listing 4: *Beginning and end of* `check_legal()` *in the original* `sjeng` *binary, with a close join point from two control flow paths shown.*

```
check_legal():
0x405cad <+0>:    jmpq   0x62465cb8

0x405cb2 <+5>:    sub    $0x20,%esp
// ...
// splits into two paths:
0x405e7f <+466>:  jne    0x40722d <check_legal+5504>
// ...
0x40722b <+5502>: jmp    0x407232 <check_legal+5509>
0x40722d <+5504>: jmpq   0x8000073e
0x407232 <+5509>: leaveq
0x407233 <+5510>: <illegal instruction>
```

Listing 5: *Beginning and end of* `check_legal()` *after instrumentation, with a close join point from two control flow paths shown.*

taneously exits while calling the next procedure (hence the term "tail call"—the procedure makes a call at the tail of its execution). This thesis will refer to such tail calls as "jump calls." Jump calls more complex to instrument jump instructions can vary across several dimensions, and serve various purposes that do not all terminate the procedure. Namely, whether a jump is conditional or unconditional, and whether it is absolute or relative, can be ignored when other dimensions of the jump instructions are considered.

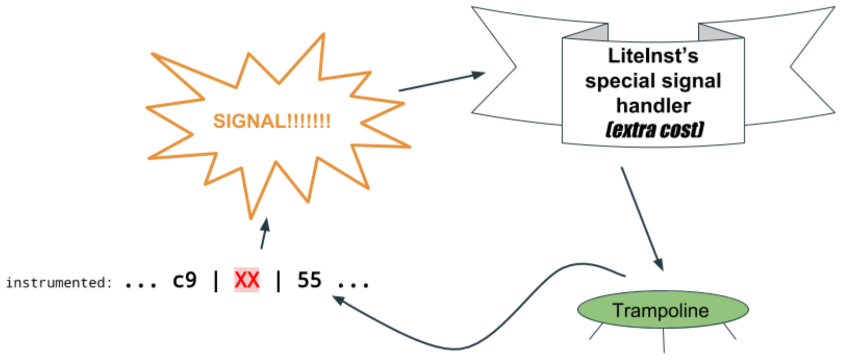
Within the category of relative jumps, there are short, near, and far jumps, based on the number of bytes in the relative offset, which determines the range of possible jump distances. Short jumps can reach a few bytes away, near jumps can reach anywhere within the same segment, and far jumps can reach across segments. These distinctions can hint at whether the jump target is inside or outside of the current procedure, and are thus useful for identifying jump calls.

Another notable distinction is between direct and indirect jumps. Direct jumps have jump targets encoded inside the instruction, whereas indirect jumps find their jump targets at run time in registers or parts of the memory. It is easy to discern the jump target of direct jumps, but not so for indirect jumps.

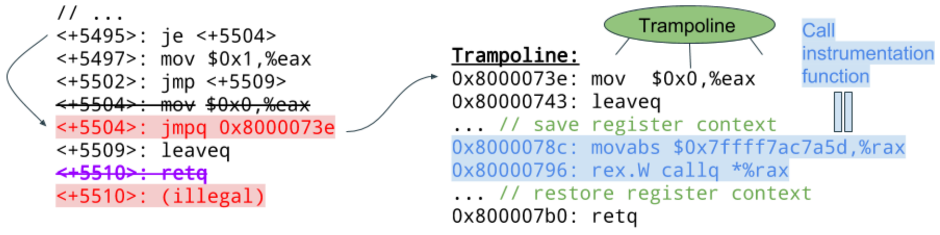
For example, register-indirect jumps (such as `jmpq *%rax` at <+3167> of Listing 7), whose jump targets are stored in registers before the instruction is executed, are tricky to work with, because their jump targets are not always within or outside of the procedure. In other words, register-indirect jumps (and indirect jumps in general) are not always tail

```
trampoline of exiting retq in check_legal():
// from the original binary:
0x8000073e: mov    $0x0,%eax // replaced by jump probe
0x80000743: leaveq // the instruction preceding the original ‘‘retq’’

// save register context
0x80000744: xchg   %ax,%ax
0x80000746: pushfq
0x80000747: push   %rsi
... // push %rdi, %rax, %rbx, %rcx, %rdx, %r8, %r9, %r10, %r11, %r12, %r13, %r14, %r15

// set up arguments to the instrumentation function
0x8000075d: movabs $0x7b,%rdi
0x80000767: movabs $0x4d,%rsi
0x80000771: mov    $0x0,%dx
0x80000775: mov    $0x1,%r10b
0x80000778: movabs $0x0,%r8

// call the exit instrumentation function defined by the analysis tool
0x80000782: movabs $0x407233,%r9 // address of the original ‘‘retq’’
0x8000078c: movabs $0x7ffff7ac7a5d,%rax // address of exit instrumentation function
0x80000796: rex.W callq *%rax // call exit instrumentation function

// restore register context
... // pop %r15, %r14, %r13, %r12, %r11, %r10, %r9, %r8, %rdx, %rcx, %rbx, %rax, %rdi
0x800007ae: pop    %rsi
0x800007af: popfq

// from the original binary: when check_legal() returns
0x800007b0: retq
```

Listing 6: *The trampoline code that leads to the exit instrumentation function after the signal handler handles SIGILL (in <+5510> of Listing 5).*


SIGNAL!!!!!!!
LiteInst's special signal handler [extra cost]
Trampoline
instrumented: ... c9 | XX | 55 ...

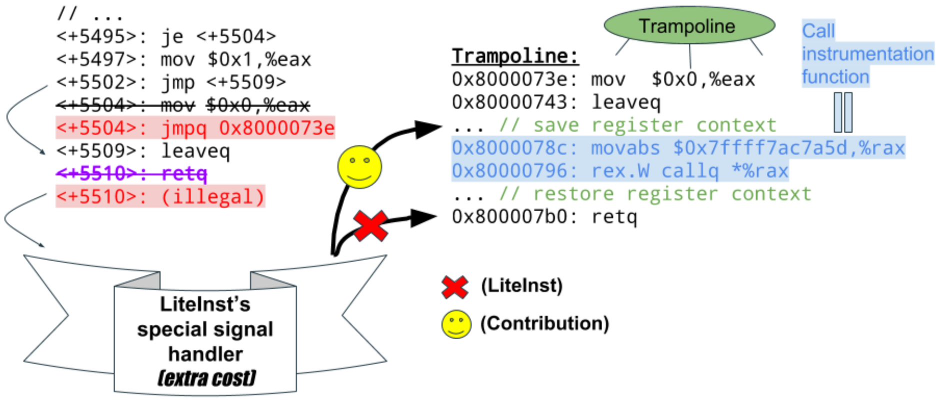
(a) *Control flow of signal handling in LiteInst.*



```
// ...
<+5495>: je <+5504>
<+5497>: mov $0x1,%eax
<+5502>: jmp <+5509>
~~<+5504>: mov $0x0,%eax~~
<+5504>: jmpq 0x8000073e
<+5509>: leaveq
~~<+5510>: retq~~
<+5510>: (illegal)
```

```
Trampoline:
0x8000073e: mov   $0x0,%eax
0x80000743: leaveq
... // save register context
0x8000078c: movabs $0x7ffff7ac7a5d,%rax
0x80000796: rex.W callq *%rax
... // restore register context
0x800007b0: retq
```

(b) *Control flow of path A (which starts from <+5495> and jumps to <+5504>). Target instruction is in purple. Probes are in red, and the in-*


Trampoline
Call instrumentation function
LiteInst's special signal handler [extra cost]
✖ (LiteInst)
☺ (Contribution)

```
// ...
<+5495>: je <+5504>
<+5497>: mov $0x1,%eax
<+5502>: jmp <+5509>
~~<+5504>: mov $0x0,%eax~~
<+5504>: jmpq 0x8000073e
<+5509>: leaveq
~~<+5510>: retq~~
<+5510>: (illegal)
```

```
Trampoline:
0x8000073e: mov   $0x0,%eax
0x80000743: leaveq
... // save register context
0x8000078c: movabs $0x7ffff7ac7a5d,%rax
0x80000796: rex.W callq *%rax
... // restore register context
0x800007b0: retq
```

(c) *Control flow of path B (which starts at <+5495> and continue through <+5497>). Red "X" marks the path that LiteInst originally implemented, and smiley face marks the path that I implemented.*

Figure 6: *LiteInst and signal handling.*

calls, and vice versa. Jump targets stored in registers are determined at run time, thus it is not always possible to decide statically whether a jump instruction of this type will jump to somewhere outside the procedure. The jump target should be checked dynamically to determine whether it is a tail call.

3.2.1 LiteInst Mechanism and Extension

To identify tail calls, LiteInst adheres to the following steps on jump instructions. If the instruction is a far jump, then it is automatically treated as a tail call. If it is relative, then LiteInst examines the jump target to determine whether its jump target resides outside the procedure boundary; if so, then it is treated as a tail call. (The framework treats all conditional jumps as relative jumps.)

LiteInst treated all register-indirect jumps as tail calls, by assuming that register-indirect jump targets always land outside the procedure. This led to `jmpq *%rax` at <+3159> of Listing 7 being instrumented with an exit probe, even though the original jump target could have been elsewhere *within* the function. Therefore, the exit instrumentation function was being called multiple times for a single call to the function, resulting in multiple exits counted for each count of entry.

Consequently, the LiteInst entry-exit counter implemented in this thesis, which counts the number of entries and exits for every procedure (see Section 4.2), produced surprising results. It is expected that procedures exit if and only if they entered (in other words, had been invoked); while some procedures did follow this pattern, some had noticeable mismatches, with significantly fewer or more entries than exits in the produced results.

The correct mechanism for a register-indirect jump instruction should be to replace it with a probe, as the original LiteInst had done, but check in the trampoline (or a special handler) whether the jump target is inside or outside the function. I temporarily mitigated this issue by essentially treating instructions like `jmpq *%rax` as a relative jump, in which the jump target decides whether the instruction simply signifies the end of a basic block (by jumping to elsewhere in the function), or is a jump call. This extension is not valid for all possible programs; however, to the best of my knowledge, tail call optimizations generate direct (not indirect) jumps, and all register-indirect jumps have targets inside the procedure in practice. This is confirmed for SPEC CPU2006 benchmarks with the LiteInst entry-exit counter when it ceased to produce mismatches with the mitigation, but a fully valid extension should implement a run-time check.

```
check_legal():
// ...
0x4068f4 <+3143>: mov     $0x1,%eax
0x4068f9 <+3148>: cmp     $0xb,%eax
0x4068fc <+3151>: ja      0x406e9b <check_legal+4590>
0x406902 <+3157>: mov    %eax,%eax
0x406904 <+3159>: mov     0x423568(,%rax,8),%rax
0x40690c <+3167>: jmpq   *%rax

0x40690e <+3169>: mov     -0x1c(%rbp),%eax
0x406911 <+3172>: movslq %eax,%rdx
// ...
```

Listing 7: *The original binary of* `check_legal()`*. The* `jmpq *%rax` *at <+3167> jumps to somewhere else inside the function.*

```
check_legal():
// ...
0x4068f4 <+3143>: mov     $0x1,%eax
0x4068f9 <+3148>: cmp     $0xb,%eax
0x4068fc <+3151>: ja      0x406e9b <check_legal+4590>
0x406902 <+3157>: mov    %eax,%eax
0x406904 <+3159>: jmpq    0x80000913

0x406909 <+3164>: xor     $0xe0620042,%eax
0x40690e <+3169>: mov     -0x1c(%rbp),%eax
0x406911 <+3172>: movslq %eax,%rdx
// ...
```

Listing 8: `check_legal()` *after instrumentation. An erroneous exit probe at <+3159> replaces original instructions at <+3159> and <+3167>.*

3.3 Halts (`hlt`) and `exit()`

Lastly, a procedure may terminate with the entire program execution, through the halt (`hlt`) instruction or a call to the C library function `exit()`.

Considerations for inserting probes for halts should have some similarities with those for returns, since they are both 1-byte instructions (see Section 3.1). However, unlike returns, *every* invoked procedure exits when a halt instruction is executed. Thus, logically, exit instrumentations should take care to treat returns and halts with similar but separate logic. LiteInst currently treats both equally as returns.

For a framework that does not instrument shared library functions, such as LiteInst, `exit()` will not be treated as a procedure exit in itself, and will not be instrumented. In addition, since `exit()` does not return, its caller will not be considered as having exited (thus LiteInst will not count it as an exit).

4 Tools for Analysis

To evaluate the impact of Pin and LiteInst on programmer experience, namely in the domain of performance as defined by elapsed time, I created three tools per framework that perform tasks a typical programmer may desire for evaluation.

4.1 Dynamic Procedure Entry Counter

I edited a sample Pintool and a sample LiteInst profiler tool from the source distributions to count (dynamically) the number of total procedures invoked and the number of invocations (entries) per procedure in a given program execution. Henceforth, they will be referred to as "procedure counters."

4.1.1 Implementation

In both Pin and LiteInst, procedure counters instrument only procedure entries, never exits. The tools insert a probe for the entry instrumentation function at the beginning of every procedure, and this instrumentation function increments a counter corresponding to the procedure upon the procedure's invocation. Note that there is a difference between "not instrumenting procedure exits" and "instrumenting procedure exits with empty instrumentation functions": the former is less costly since it does not need to insert code at the end of procedures, which also avoids the cost of directing execution to trampolines and coming back. (It can also be especially costly to instrument procedure exits, since they might require the more expensive trap or illegal instruction probes.)

In LiteInst, the procedure counter uses one array to keep track of all procedure entry tallies. The integer ID of a procedure is its index into the array, since this is the most intuitive procedure identifier that is available to instrumentation functions. (String names of procedures are not readily available to instrumentation functions.) The procedure counter defines an entry instrumentation function, and specifies `ProbePlacement::ENTRY` for all procedures. It then associates probes with the entry instrumentation function, to instruct LiteInst to replace the first instructions of each procedure in the symbol table with probes leading to the entry instrumentation function. No exit instrumentation probes are placed. The number of procedures called is equal to the number of nonzero tallies in the array.

Pin's procedure counter uses a linked list, in which each node is a struct representing a single procedure. A linked list is used rather than an array because this list does not have access to the number of procedures for allocating an array, but can access specific fields of

```
// keeps track of all entry tallies
int counts_array[num_procedures]; // initialized to 0

uponEntryInLiteInst(Procedure invoked_procedure) {
    // invoked_procedure is a struct, ID is an integer
    counts_array[invoked_procedure.ID]++;
}
```

Listing 9: *Pseudocode for entry instrumentation functions in the procedure counters, following LiteInst's implementation.*

structs. LiteInst cannot use a linked list because it would need to look up the procedure every time a procedure is invoked, which is `O(n)` with linked lists, but know the total number of procedures beforehand, from the symbol table. This difference should not have noticeable impact on performance, since neither framework's procedure tool is traversing their data structures, but simply changing one part of it. Each struct has a field for the entry tally, and the API function `RTN_InsertCall()` will update this field with the incrementing function if the function is specified along with the field address and `IPOINT_BEFORE` to specify entry (rather than exit) instrumentation.

Listing 9 shows the pseudocode for entry instrumentation functions, specifically for LiteInst (but the concept is the same for Pin).

4.2 Dynamic Procedure Entry *and Exit* Counter

I extended the procedure counters to tally each procedure's entries *and* exits in a given program. For sake of simplicity, they will be referred to as "entry-exit counters." Intuitively, entry-exit counters should complete roughly twice the amount of work as procedure counters do, but it remains to be seen whether this intuition translates to measured results.[4]

4.2.1 Implementation

LiteInst's entry-exit counter behaves similarly to LiteInst's procedure counter, except with another array for exit tallies, another instrumentation function for procedure exits, and the setting of `ProbePlacement::BOUNDARY` to instruct LiteInst to instrument both ends of procedures. Procedure exits are counted in the same manner as procedure entries, with

[4]Procedures usually exit every time they are invoked, which make the entry-exit counter seemingly unhelpful. However, in practice, some procedures do not end up exiting due to complications with the dynamic linking process or other reasons.

```
// keeps track of all entry or exit tallies
int entries_array[num_procedures], exits_array[num_procedures]; // initialized to 0

uponEntryInLiteInst(Procedure invoked_procedure) {
    // invoked_procedure is a struct, ID is an integer
    entries_array[invoked_procedure.ID]++;
}

uponExitInLiteInst(Procedure invoked_procedure) {
    exits_array[invoked_procedure.ID]++;
}
```

Listing 10: *Pseudocode for entry and exit instrumentation functions in the entry-exit counters, following LiteInst's implementation.*

exits of every procedure (which may be a return, halt, or an jump to somewhere outside the procedure, as discussed in Chapter 3 replaced by a probe).

Pin's entry-exit counter is also similar to Pin's procedure counter, except each procedure struct has an additional field for the exit tally, and the instrumentation function contains one more call to `RTN_InsertCall()` with `IPOINT_AFTER` to increment the exit tally field when its procedure exits.

Listing 10 shows pseudocode for instrumentation functions in the entry-exit counters, specifically of the logic used in the LiteInst implementation (but is still translatable to Pin).

4.3 Dynamic Call Graph Generator

I created a Pintool and LiteInst tool to instrument entries and exits of all procedures in a program's symbols table, where entry and exit instrumentations work together to collect a graph of all procedure calls to other procedures. The resulting call graph does not store any information other than how many times one procedure has called another.

Dynamic call graphs capture the call tree of a specific program execution, allowing the programmer to understand the program control flow at function granularity. They can be used for program optimization [7, 18] and malware detection [17]. Dynamic call graph generation can be optimized [6] and improved to store contextual information [13]. In this thesis, the Pintool and LiteInst tool generate basic dynamic call graphs, which are sufficient for their evaluation.

```
stack<int> procedures_stack; // keeps track of invoked procedures that have not exited, in order of invocation
map<pair<int,int>, int> call_graph_map; // keeps track of tallies of caller-callee pairs in this execution

uponEntry(Procedure invoked_procedure) {
    // invoked_procedure is a struct, ID is an integer
    if ( ! procedures_stack.empty() ) {
        caller_ID = procedures_stack.top().ID;
        callee_ID = invoked_procedure.ID;
        pair = <caller_ID, callee_ID>;
        if ( pair in call_graph_map ) { call_graph_map.get(pair)++; }
        else { call_graph_map.insert(pair, 1); }
    }
    procedures_stack.push(invoked_procedure.ID);
}

uponExit(Procedure exiting_procedure) {
    assert ( exiting_procedure.ID == procedures_stack.top().ID ); // should be the most recently invoked
         procedure
    procedures_stack.pop();
}
```

Listing 11: *Pseudocode for entry and exit instrumentation functions in the call graph generators.*

4.3.1 Implementation

The LiteInst dynamic call graph generator represents a call graph as a map, with a key being a pair of procedures, and a value being the number of times this relationship has occurred. The pair of procedures in a key consists of a caller and a callee. A stack is used to keep track of the latest procedure invoked (represented by its ID). Upon invocation of a procedure, each entry instrumentation increments the value of the corresponding caller-callee pair, where the callee is the most recently invoked procedure, and the caller is the procedure on the top of the stack. The entry instrumentation function then pushes the callee procedure onto the stack. Each exit instrumentation pops off the top of the stack, which has the same ID as the exiting procedure.

The Pintool for dynamic call graph generator also uses a map and a stack for accumulating the context-insensitive call graph. The instrumentation function calls RTN_InsertCall() with IPOINT_BEFORE and a function similar to LiteInst version's entry instrumentation function, and RTN_InsertCall() again with IPOINT_AFTER and a function similar to LiteInst version's exit instrumentation function.

Listing 11 shows pseudocode for instrumentation functions in the call graph generators.

Figure 7 portrays what happens when the call graph generator is used on a hypothetical

```
void B() {
    C(); // doesn't call any procedures
}

void A() {
    B(); // calls C()
    D(); // doesn't call any procedures
}
```

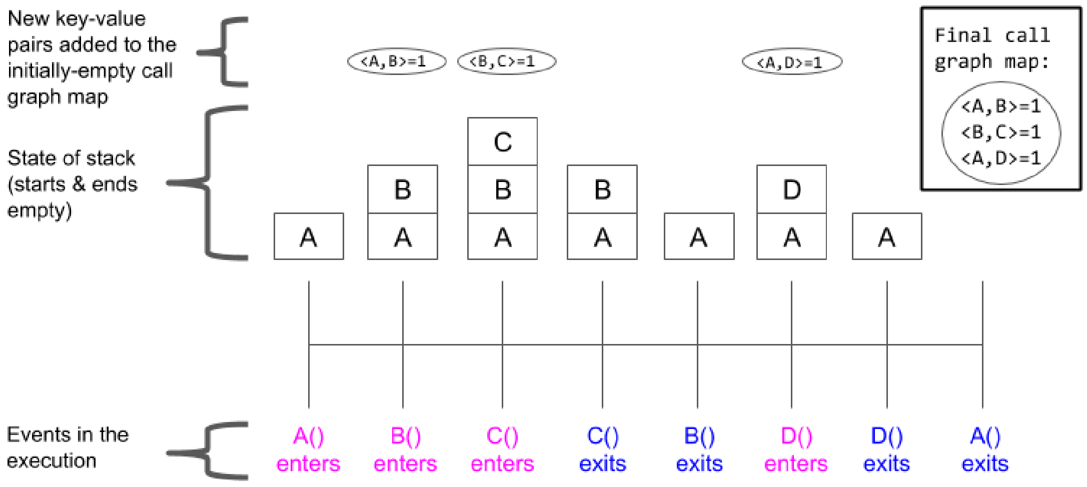


Figure 7: *Behavior of the call graph generators on the execution of a hypothetical program.*

program. In this program, A calls B, which calls C and exits, then B exits, then A calls D which then exits, then A itself exits. Every time a procedure calls another, the stack grows, and the pair of caller and callee procedures is inserted into the call graph as one edge. (The procedures are the nodes.) For example, when A calls B, B is pushed onto the stack, and the pair of <A,B> with a count of one (to indicate only one edge so far) is inserted into the map. When B exits, it is popped off the stack. The same pattern applies to the other steps.

5 Evaluation

A goal of this work is to determine the overheads in performance time of Pin and LiteInst, and to understand the source of these overheads. All tools created with these frameworks (three per each, as described in Chapter 4) were cross-validated with their counterparts from the other framework, then timed for performance.

5.1 Experimental Setup

All experiments were conducted on a machine with 2×18-core Intel Xeon E5-2695 v4 CPUs with a total of 256GB of RAM across two NUMA nodes. The CPUs' performance mode was selected, and their simultaneous multithreading disabled. The machine runs the Ubuntu 17.10 distribution with Linux kernel version 4.13.0.

The frameworks were measured for instrumenting at the procedure boundary granularity, with tasks such as tallying number of procedures. This was the granularity evaluated for LiteInst in the original paper.

Phases of program execution was measured as follows. The program starts with start-up in which the framework and tool prepare to instrument the binary. The program continues with running the instrumented binary, which makes up most of the overall time. After the instrumented binary finishes execution, the framework and the tool free up memory in the tear-down phase. Start-up and tear-down phases also constitute the setup phase of a program execution.

Measurements consist of the following.

- Total elapsed run time of the program, including the instrumentation process and the execution of the instrumented binary, was measured and normalized by elapsed run time of the uninstrumented binary. This information can reflect the slowdown to be expected when a programmer is using the corresponding framework.
- For LiteInst procedure counter and entry-exit counter, the different types of probes executed during run time were categorized, to better understand LiteInst overhead.

Elapsed run time of the instrumented binary (which is roughly the total amount of time minus the setup duration), normalized by elapsed run time of the uninstrumented binary, was also measured in order to indicate the efficiency of the instrumentation. However, this data is excluded from presentation, because for all evaluated benchmarks I found that the raw run time of the instrumented binary is within a few milliseconds of the corresponding raw total elapsed time, such that the trends of the instrumented binary results mirror those of the total elapsed time.

Data was acquired from 10 trials of each experimental configuration and then averaged. All binaries were compiled with -O0 because it is the highest level of gcc optimization that is compatible with LiteInst. The binaries did not differ in their execution behavior between trials, which allowed validation executions (Section 5.2) and probe categorization trials (Section 5.3.2) to be ran once instead of multiple times.

Below are the benchmarks ran with the tools; all are single-threaded benchmarks from the SPEC CPU2006 suite [20], and are designed to simulate real-world computation workloads:

- astar: finds path in a given binary map with three different algorithms.
- bzip2: compresses input files.
- h264ref: compresses videos.
- lbm: simulates behavior of fluids in 3D with the "Lattice Boltzmann Method."
- libquantum: simulates a quantum computer factoring input numbers.
- mcf: juggles the scheduling multiple public transportation vehicles.
- sjeng: plays chess and its variants against a player or list of inputs, using various Artificial Intelligence techniques.

5.2 Validation

In order to compare Pin and LiteInst performance in a fair manner, I implemented the same algorithms with comparable data structures for each pair of equivalent Pintool and LiteInst tools, and compared the output of each pair of tools against each other. Each tool was ran once to produce output for this purpose. (Note that results of a tool on a benchmark did not differ between executions, therefore it was sufficient to run each execution with output once for validation.) Since the output process itself has overhead, which is not essential to the behavior of the tools, I modified the tools to suppress output during timing trials.

Ideally, equivalent tools should produce identical output. In reality, equivalent tools often produced small number of discrepancies in their output on the benchmarks I tested. Manual examination was conducted for each benchmark with discrepancies to determine the root cause; while different benchmarks had different discrepancies, separate executions of the same tools on the same benchmarks had consistent discrepancies. Usually, these discrepancies were in the small number (<5) of entries or exits for a few (<10), rarely-called library functions (like `_start()`). Most benchmarks have upwards of 1 billion procedure entries (except for `lbm`, which has about 6000 procedure entries per execution), so these discrepancies would not be significant in affecting performance. Benchmarks with these or other kinds of negligible discrepancies (such as from name-mangling, or design differences that are tangential to the underlying DBI approach) were considered eligible for timing experiments.

As described in Sections 3.1.2 and 3.2.1, the original LiteInst framework caused some procedures to have significantly different number of entries than their exits, which corrupted the stack and the accuracy of the call graph generator. After implementing the extensions described in those same sections, LiteInst's call graph generator produced results equivalent to its Pin counterpart.

5.3 Results

5.3.1 Total Elapsed Run Time

The following (Table 1) shows the run time of each benchmark without instrumentation, averaged over 10 trials.

Native Run Times of Benchmarks			
Benchmark	Average Total Elapsed Time (Seconds)	Standard Deviation (Seconds)	Standard Deviation (Percentage of Total Time)
astar	104.0	0.6	0.6%
bzip2	65.6	0.3	0.4%
h264ref	45.7	0.1	0.1%
lbm	613.3	1.5	0.2%
libquantum	918.8	0.9	0.1%
mcf	413.2	6.2	1.5%
sjeng	7.5	0.0	0.0%

Figures 9, 10, and 11 show the elapsed times of procedure counters, entry-exit counters, and call graph generators, respectively, on the benchmarks (normalized with average run time of the uninstrumented benchmarks). Figure 8 presents the three sets of data together on a $\log_2$ scale, for comparison between different tools.

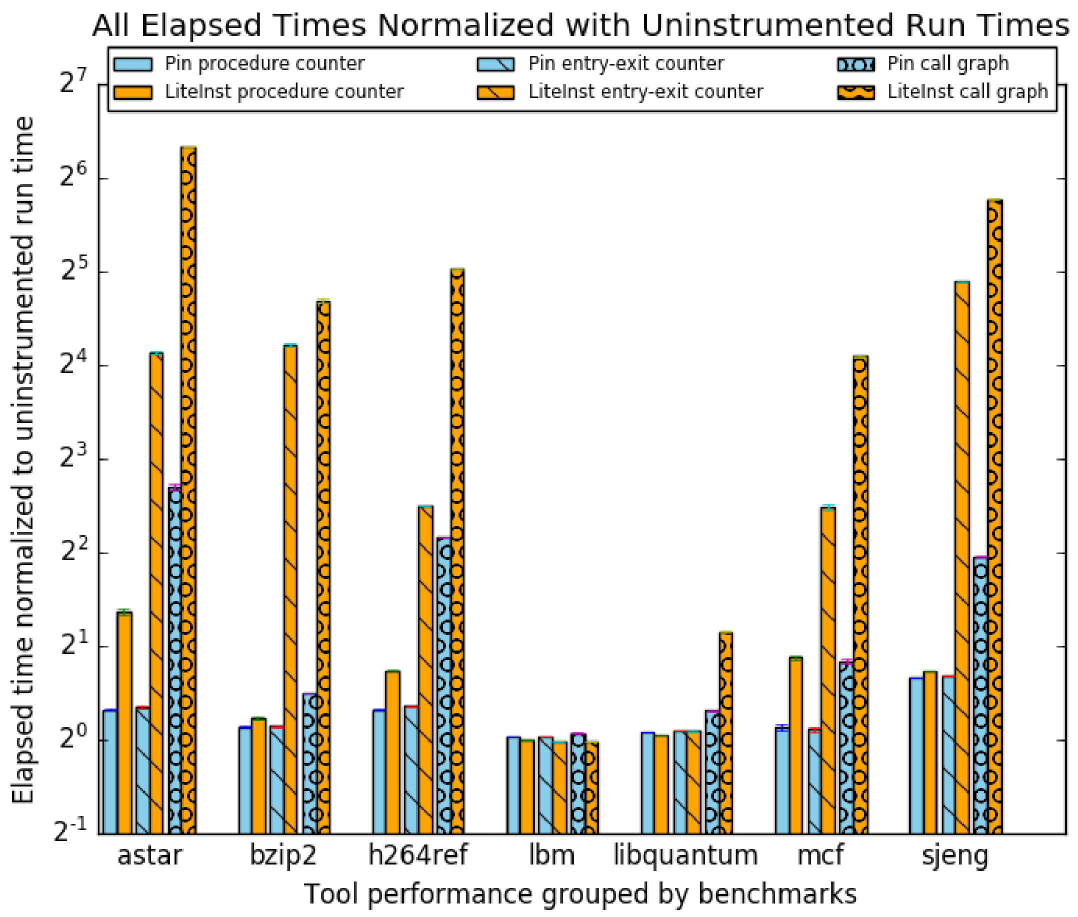


Figure 8: *Elapsed times of instrumented benchmarks under all tools, averaged over 10 trials each and normalized with the uninstrumented binary average run time. The y-axis is in* $\log_2$ *scale. Error bars show standard deviation.*

In general, the Pin and LiteInst procedure counters were almost as fast as the uninstrumented binary (Figure 9). Pin took less than 2× the native run time, and LiteInst was slightly slower, taking at most 2.6× the native run time on average.

As mentioned in Section 4.2, we should expect to see that the total elapsed run time of entry-exit counter on each benchmark is double that of the procedure counter, since the former instruments both ends of each procedure whereas the latter instruments only the beginning of procedures. Based on the results of Figure 9, Pin's and LiteInst's entry-exit counters should be 4× and 5.2× as slow as the native binary at most, respectively. However, this was not the case (Figure 10). Pin's entry-exit counter run time was still close to the native binary run time, with no more than 2× overhead. Meanwhile, LiteInst also performed almost as well as native binaries for libquantum and lbm (two benchmarks with the

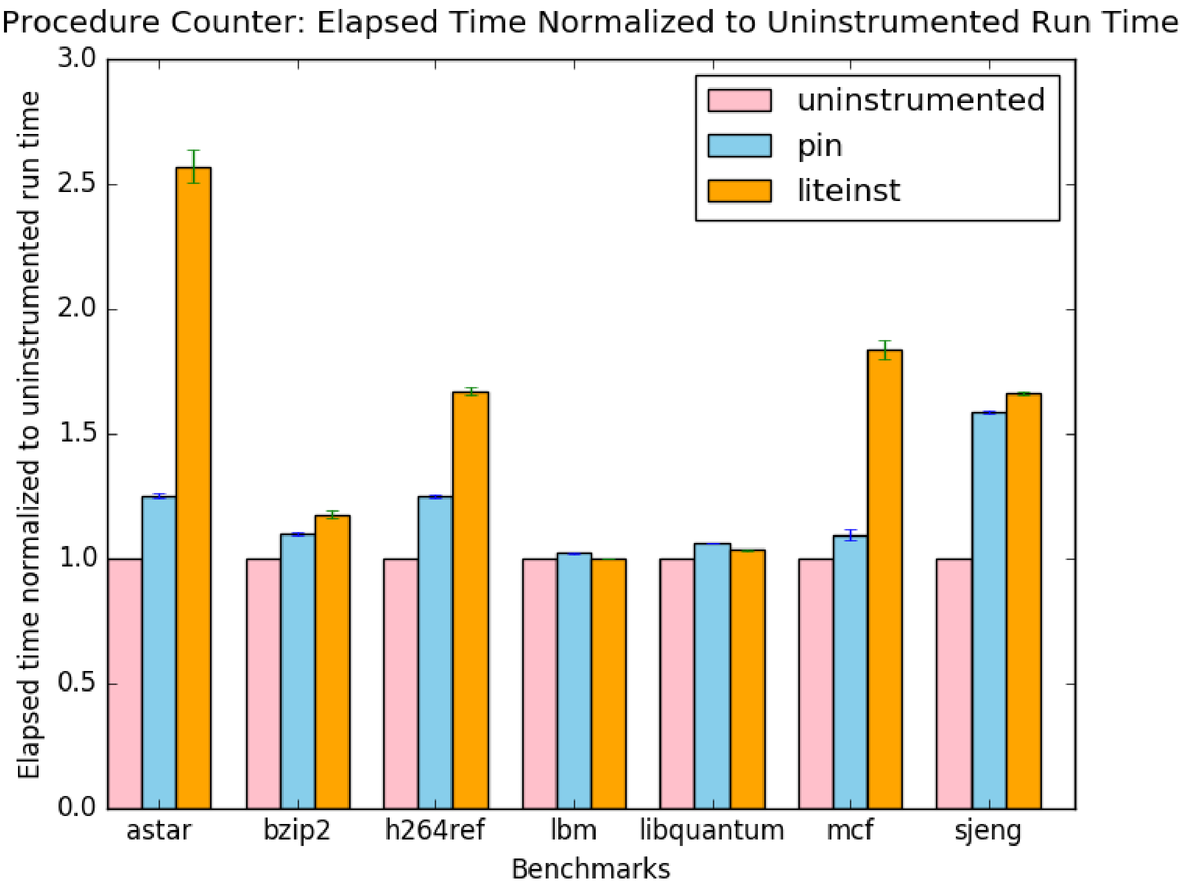


Figure 9: *Elapsed times of instrumented binary benchmarks under the procedure counter, averaged over 10 trials each and normalized with the uninstrumented binary average run time. The y-axis is in linear scale. Error bars show standard deviation.*

slowest uninstrumented run times, according to Table 1). However, for the other benchmarks, LiteInst took up to 30× the amount of run time that the uninstrumented binary took.

With the call-graph generator tool, Pin introduced about 1-9× overhead into the binaries (Figure 11). This is reasonable considering that the call graph algorithm was not optimized. LiteInst again performed well for `lbm` and `libquantum`, but introduced significant overhead to the faster benchmark, taking up to 80× the amount of native binary run time to instrument and run these benchmarks.

5.3.2 Probe Categorization

In general, LiteInst appeared to incur more overhead than Pin and the native binary, except for the two slowest benchmarks. The stark contrast between LiteInst's procedure counter

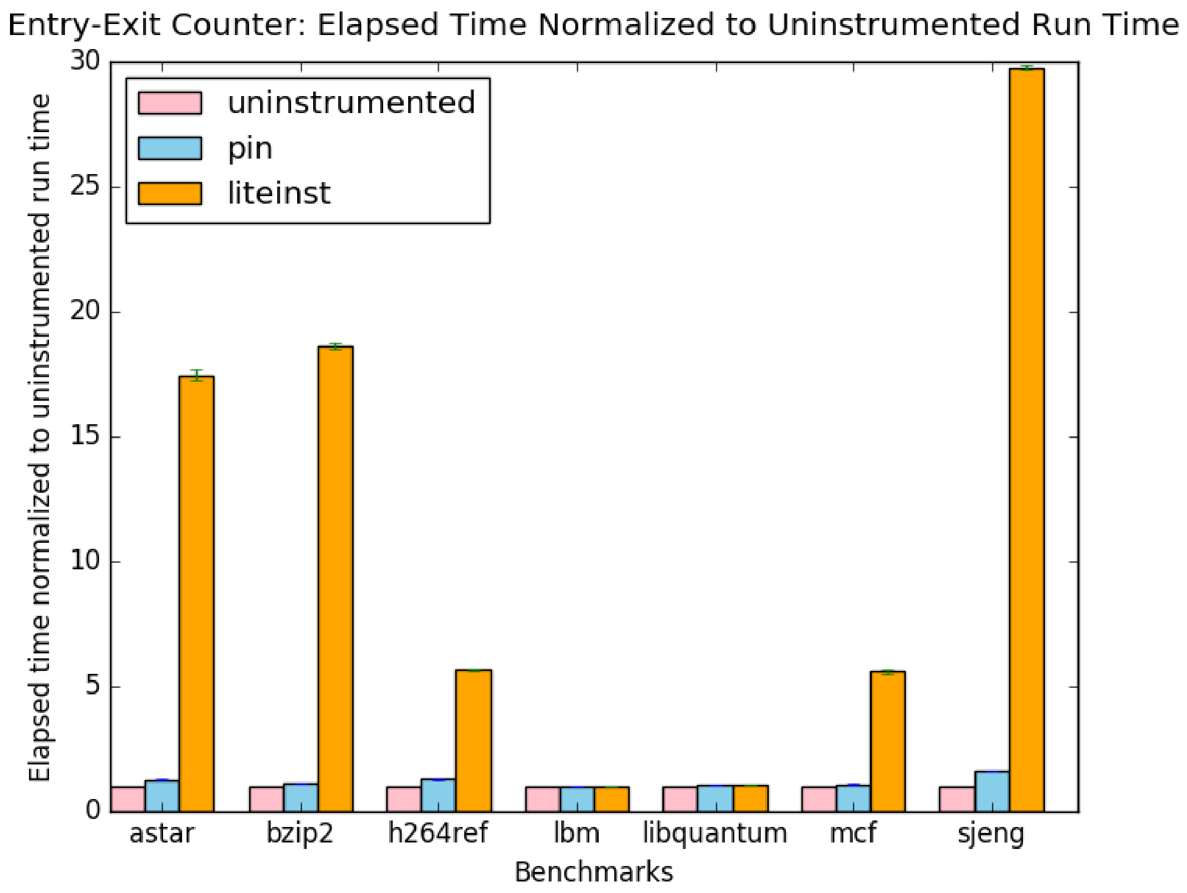


Figure 10: *Elapsed times of instrumented binary benchmarks under the entry-exit counter, averaged over 10 trials each and normalized with the uninstrumented binary average run time. The y-axis is in linear scale. Error bars show standard deviation.*

and entry-exit counter performances suggest that the latter does not simply carry out 2× as much work as the former. To discern the source of this extra work during run time, I measured the number of executed probes in each benchmark instrumented by these two LiteInst tools. Note that this measurement does not account for probes that were inserted statically but never executed.

Figure 12 shows the different types of probes used for the LiteInst procedure counter and entry-exit counter. (The LiteInst call graph generator was verified to use the same types or probes and with the same distribution as the entry-exit counter, which makes sense because they both instrument both ends of procedure boundaries.) LiteInst's procedure counter used only standard jump probes, while its entry-exit counter used both jump and illegal instruction probes. The two benchmarks with the least overhead (see Figures 8, 9, and 10),

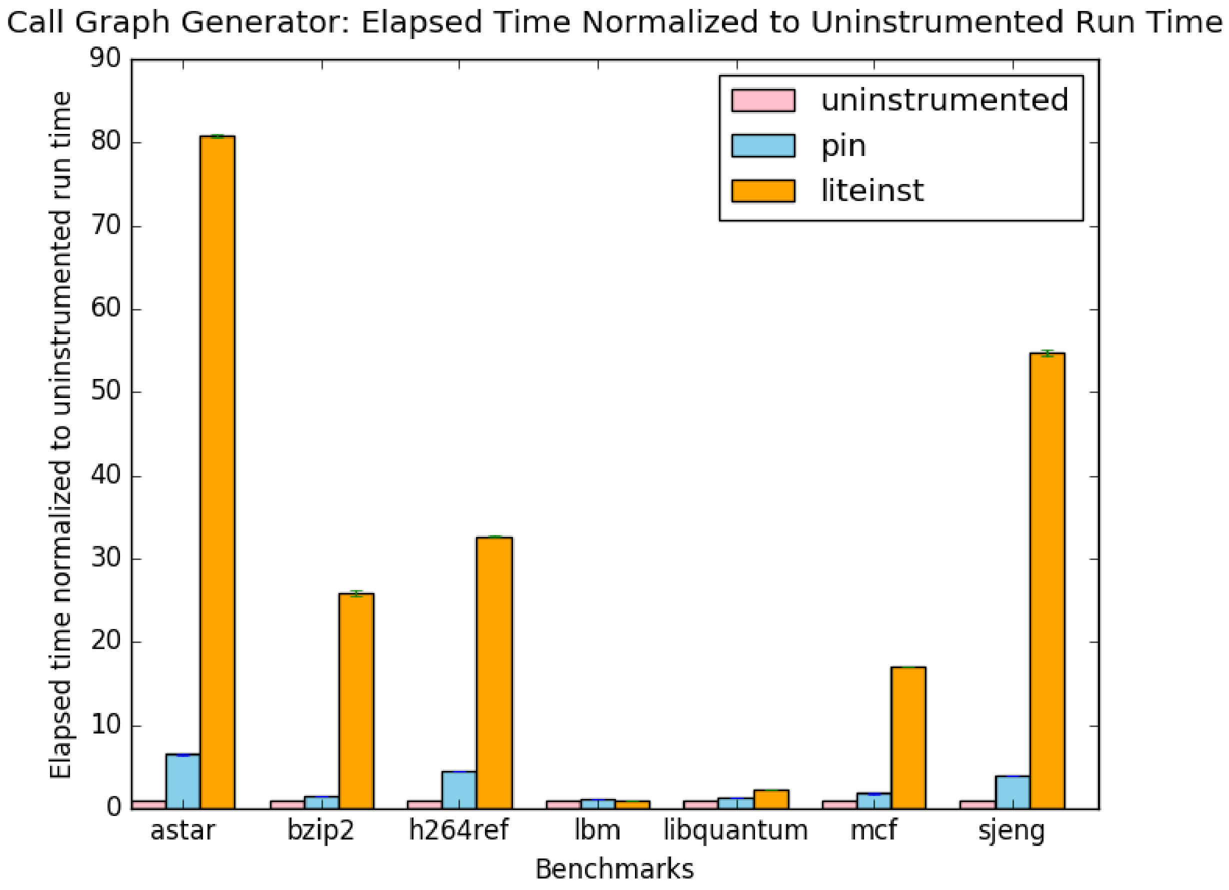


Figure 11: *Elapsed times of instrumented binary benchmarks under the call graph generator, averaged over 10 trials each and normalized with the uninstrumented binary average run time. The y-axis is in linear scale. Error bars show standard deviation.*

lbm and libquantum, were the only two benchmarks that did not have illegal instructions probes from the entry-exit counter.

For the LiteInst procedure counter and entry-exit counter, average number of executed probes per second was also calculated for each type of probes in each benchmark (Figure 13). As expected, the entry-exit counter used an average of twice as many probes per second (and overall) as the procedure counter for all benchmarks. lbm and libquantum had significantly fewer probes than the other benchmarks, for both tools; and astar had the most probes per second, for both tools.

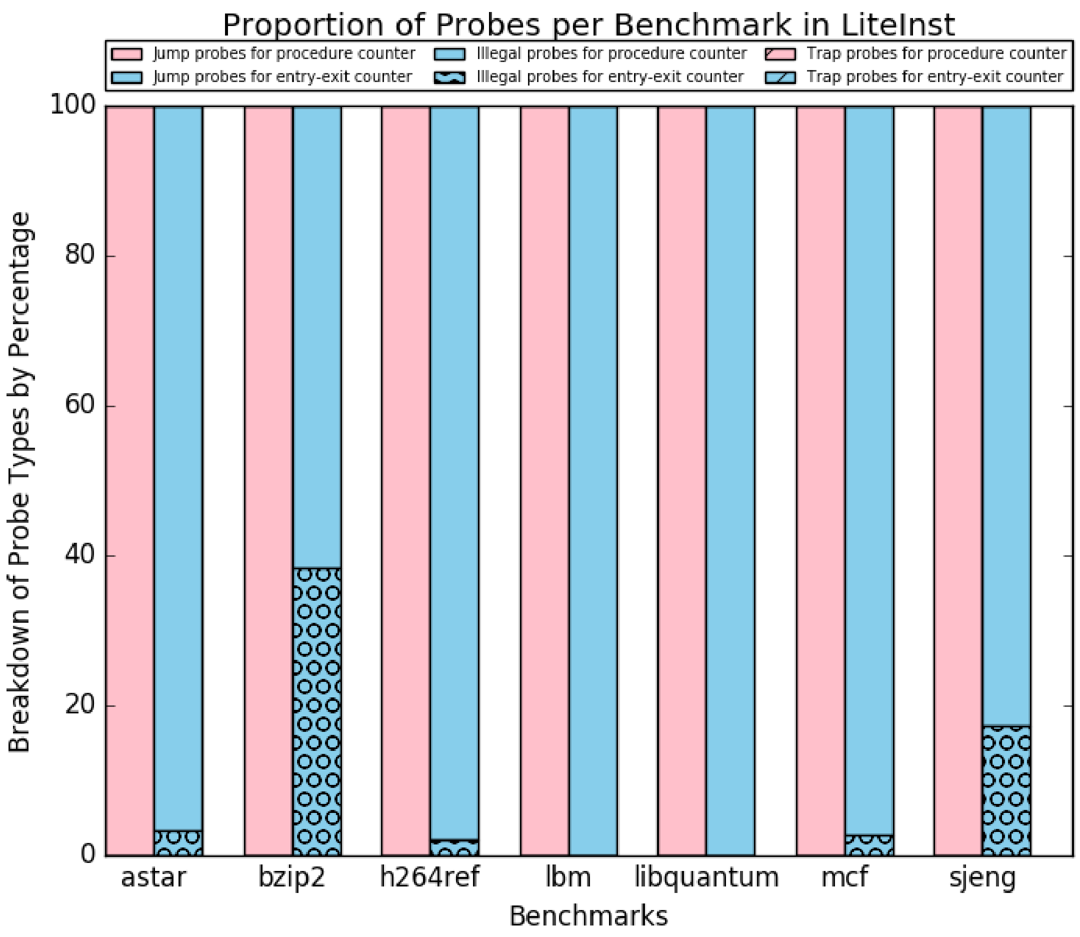


Figure 12: *Breakdown of executed probes in LiteInst's procedure counter and entry-exit counter for all benchmarks. The y-axis is in linear scale.*

5.4 Discussion

Overall, Pin and LiteInst seem to incur different amounts of overhead on the instrumentation tasks in this evaluation. Pin generally performs close to native binary run time (except for the call graph generator (which uses an unoptimized algorithm that the LiteInst call graph generator also uses). Meanwhile, LiteInst seemed to perform almost as well as Pin when instrumenting only procedure entries, but was significantly slower for most benchmarks when exit instrumentation was introduced. In addition, the LiteInst entry-exit counter performed *significantly more than* twice as slow as the LiteInst procedure counter, even though the former should theoretically perform about twice the amount of work (by inserting and executing twice the amount of probes) than the latter does.

The breakdown of executed probe types showed that while the LiteInst procedure counter used only jump probes for all benchmarks, the LiteInst entry-exit counter used jump as well

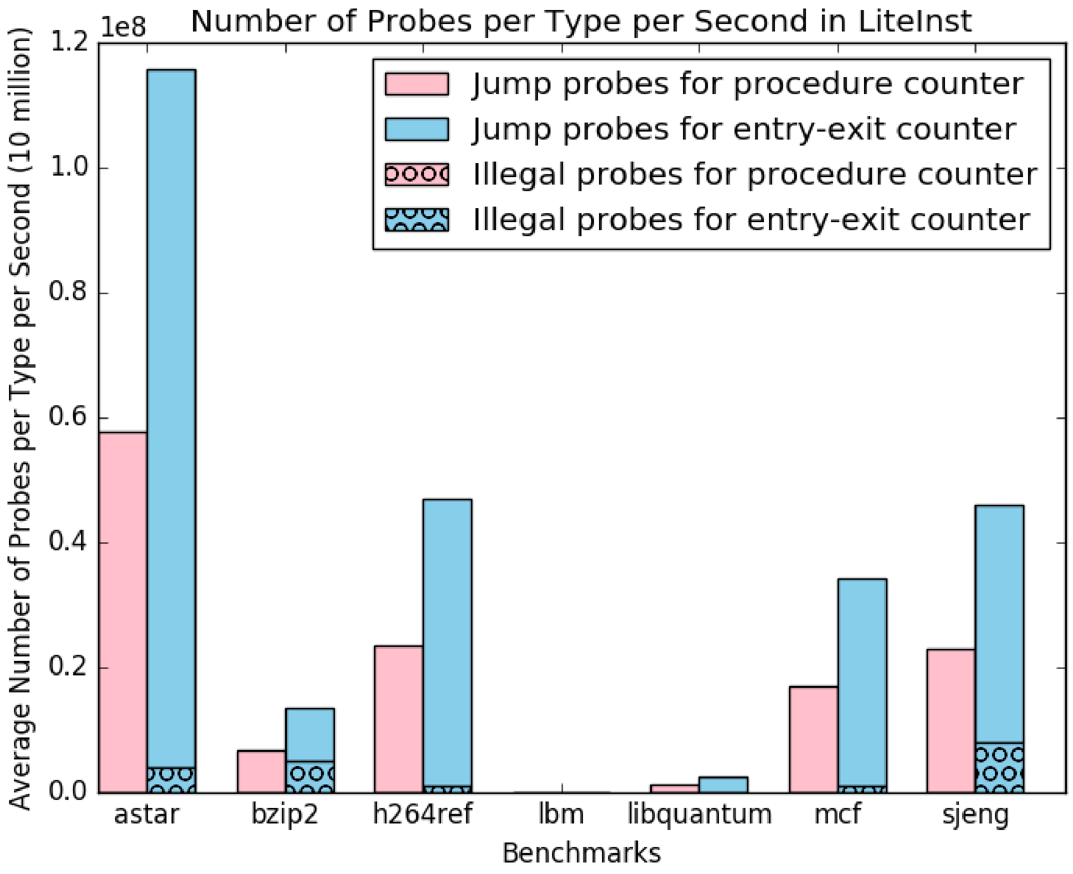


Figure 13: *Breakdown of executed probes per second in LiteInst's procedure counter and entry-exit counter for all benchmarks. Note that the y-axis is in linear scale, and the unit is 10 million.*

as illegal instruction probes for most benchmarks. Since illegal instruction probes are more expensive than jump probes, the former may have been responsible for the extra overhead of LiteInst's entry-exit counter. As the only difference between the two tools is whether exits are instrumented, this contrast in probe breakdown also suggests that the illegal instructions were all used for instrumenting return instructions or other forms of procedure exits.

Interestingly, LiteInst's performance was comparable and, in most cases, even faster than Pin across two benchmarks (lbm and libquantum) for all three tools, and both frameworks performed around native binary run time. These two benchmarks also had the longest uninstrumented run time, and lbm had significantly fewer procedure entries (about 6000) than all other benchmarks (upwards of 1 billion). These results seem to suggest that LiteInst may have a faster startup time than Pin, and that this performance advantage is reversed when too many probes are executed during run time.

The breakdown of executed probe types revealed that lbm and libquantum had few, if any, expensive illegal instruction probes with the LiteInst procedure counter or entry-exit counter, while the rest were cheap jump probes. All other benchmarks used some illegal instruction probes in addition to the jump probes. This difference in probe type breakdown supports the inference that the expensive probes contributed to LiteInst's overhead on the latter group of benchmarks.

However, the benchmark with the highest percentage of illegal instruction probes used by the LiteInst entry-exit counter was bzip2, which had moderate slowdown for LiteInst compared to the other benchmarks. astar had the second-lowest percentage of trap-based probes, but the highest overhead with LiteInst's procedure counter and call graph generator. Thus, expensive probe usage cannot be the only explanation for LiteInst's overhead.

According to the breakdown of executed probe types *per second*, lbm and libquantum had significantly fewer probes per second than all other benchmarks. On the other hand, astar had the most probes per second out of all benchmarks. Therefore, LiteInst seemed to perform very well on target programs with sparse instrumentations, and less well on those with dense instrumentations.

The above observations suggest the following characterizations for Pin and LiteInst. Pin may incur a fixed cost for setup, but can optimize the instrumentation functions during instrumentation such that overhead increases slowly with increasing number of instrumentations. On the other hand, LiteInst may have a smaller setup cost, but its overhead increases at a faster rate in response to increasing number of instrumentations or executed probes. The mechanism of probes themselves, especially expensive ones like the illegal instructions probes, contribute to this overhead.

It should be noted that this evaluation tested only one type of instrumentation workload, in which all procedures of a target program were instrumented. Other instrumentation workloads may yield additional insights. For example, LiteInst may perform better at rapidly toggling (activating or deactivating probes during run time) for hot code than Pin, since LiteInst seems to do well when handling a relatively fixed number of probes, while Pin may perform better at toggling for code that is not executed as often, which covers a bigger area of code. While LiteInst has been evaluated on rapid toggling [14], future work can focus on further evaluation of toggling in LiteInst with respect to Pin.

6 Future Work

The results of this evaluation suggest that LiteInst is suited for lightweight usage, while Pin is useful for heavyweight analyses. However, it should be noted that Pin is more mature than LiteInst: the former has been under the development of the Intel Corporation since before 2005 [23], while LiteInst is developed in an academic setting, and was only formally presented in 2017 [14]. As indicated by its authors [14], LiteInst can still benefit from optimizations from which Pin is already benefiting, such as the inlining of instrumentation functions in the trampoline to avoid costs for saving and restoring the full context.

Further work should be done to quantify the causes of difference in performance between LiteInst and Pin. Specifically, the results of this evaluation suggest that Pin has a higher setup cost than LiteInst, and that LiteInst performs well under a threshold number of instrumentation sites. Future work should aim to substantiate these claims, by measuring the setup times of the two frameworks (which was not part of this evaluation, because Pin's API does not allow it at the moment), evaluating the frameworks on more benchmarks, and identifying the threshold number or the program behavior that determines it. Evaluating rapidly toggling during run time in both Pin and LiteInst may be helpful for this purpose.

Section 3.2.1 mentions that, in order for LiteInst to correctly identify whether a register-indirect jump instruction is a tail call, LiteInst must verify the address of the jump instruction during run time (as opposed to statically). Extending LiteInst such that it treats all register-indirect jump instructions as internal control flow changes did mitigate issues of mismatching entries and exits; however, dynamic address checks will provide both theoretical backing and practical accuracy. The cost of such address checks may also contribute to LiteInst's overhead.

Lastly, it may be interesting to explore a potentially more efficient approach that combines both DBT and DPI. Such a hybrid could start instrumentation with a DPI approach, to take advantage of its fast startup. It can also use the DPI approach for hot code. Once the number of instrumentation sites exceed a threshold, this hybrid can switch to a DBT approach.

7 Conclusion

This thesis aimed to better understand the tradeoffs of the two DBI approaches exemplified by Pin and LiteInst. For this purpose, I extended the LiteInst framework and compared its performance on procedure counting, procedure entry and exit counting, and dynamic call graph generation against Pin. Evaluation of these three tasks on a set of SPEC CPU 2006 benchmarks found that, in general, Pin performed close to uninstrumented binary run time, while LiteInst took significantly more time. However, LiteInst performed as well as Pin for benchmarks that required less probes, especially expensive probes. Therefore, LiteInst may do well for lightweight tools, while Pin may better for heavyweight ones. Future work should involve further optimizing and extending procedure boundary instrumentation support for LiteInst, as well as conducting more measurements to support or contradict the outcome of this evaluation. Exploration of a hybrid approach that combines the two frameworks or approaches could also yield interesting results.

References

[1] Source instrumentation overview. URL http://www.stridewiki.com/index.php?title=Source_Instrumentation_Overview.

[2] N. Aaraj, A. Raghunathan, and N. K. Jha. Dynamic binary instrumentation-based framework for malware defense. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[3] ACM. Pin, 2018. URL https://dl.acm.org/citation.cfm?id=1065034.

[4] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[5] P. Arafa, H. Kashif, and S. Fischmeister. Redundancy suppression in time-aware dynamic binary instrumentation. *CoRR*, abs/1703.02873, 2017. URL http://arxiv.org/abs/1703.02873.

[6] M. Arnold and D. Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *International Symposium on Code Generation and Optimization*, 2005.

[7] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *In 2000 ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO '00)*, 2000.

[8] A. Bechini and C. A. Prete. Behavior investigation of concurrent java programs: an approach based on source-code instrumentation. *Future Generation Computer Systems*, 18(2):307 – 316, 2001.

[9] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* Ph.D. dissertation, Massachusetts Institute of Technology, 2004.

[10] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2003.

[11] B. Buck and J. K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, Nov. 2000.

[12] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2004.

[13] M. Chabbi, X. Liu, and J. Mellor-Crummey. Call paths for pin tools. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014.

[14] B. Chamith, B. J. Svensson, L. Dalessandro, and R. R. Newton. Instruction punning: Lightweight instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[15] A. S. Charif-Rubial, D. Barthou, C. Valensi, S. Shende, A. Malony, and W. Jalby. Mil: A language to build program analysis tools through static binary instrumentation. In *20th Annual International Conference on High Performance Computing*, 2013.

[16] P. K. Chittimalli and V. Shah. Gems: A generic model based source code instrumentation framework. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012.

[17] K. P. Deepta and A. Salim. Detecting malwares using dynamic call graphs and opcode patterns. In M. Singh, P. Gupta, V. Tyagi, A. Sharma, T. Ören, and W. Grosky, editors, *Advances in Computing and Data Sciences*, pages 91–101. Springer Singapore, 2017.

[18] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[19] Y. He, Y. Tao, and T. Zhang. Cppins: A source query language for instrumentation. In Y. Yuan, X. Wu, and Y. Lu, editors, *Trustworthy Computing and Services*, 2013.

[20] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[21] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2010.

[22] H. Li, J. Oh, H. Oh, and H. Lee. Automated source code instrumentation for verifying potential vulnerabilities. In *ICT Systems Security and Privacy Protection: 31st IFIP*

TC 11 International Conference, SEC 2016, Ghent, Belgium, May 30 - June 1, 2016, Proceedings, 2016.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[24] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, 2005.

[25] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[26] M. Olszewski, K. Mierle, A. Czajkowski, and A. D. Brown. Jit instrumentation: A novel approach to dynamically instrument operating systems. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, 2007.

[27] R. J. Rodriguez, J. A. Artal, and J. Merseguer. Performance evaluation of dynamic binary instrumentation frameworks. *IEEE Latin America Transactions*, 12(8):1572–1580, Dec 2014.

[28] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994.

[29] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical report, April 2001. URL https://www.microsoft.com/en-us/research/publication/vulcan-binary-transformation-in-a-distributed-environment/.

[30] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2014.