TOOLS FOR DEBUGGING AND ANALYSIS OF CONCURRENT PROGRAMS

by

Saeed Taheri

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

School of Computing

The University of Utah

July 2021

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Saeed Taheri**
has been approved by the following supervisory committee members:

Ganesh Gopalakrishnan ,	Chair(s)	—— Date Approved
Zvonimir Racamaric ,	Member	—— Date Approved
Hari Sundar ,	Member	—— Date Approved
Alexander Lex ,	Member	—— Date Approved
Martin Burtscher ,	Member	—— Date Approved

by **Mary Hall** , Chair/Dean of
the Department/College/School of **Computer Science**
and by **David B. Keida** , Dean of The Graduate School.

ABSTRACT

With the high growth in computation power and the invention of modern languages, concurrent software testing and debugging is vital to deliver reliable software. Finding bugs in concurrent/parallel software is notoriously challenging because 1) the interleaving space grows exponentially with the number of processing units (e.g., CPU cores), 2) the non-deterministic nature of concurrent software makes concurrent bugs difficult to reproduce, and 3) root-causing misbehaved executions of a concurrent program is non-trivial due to the complex interactions between concurrent components of the program. In this work, we have designed and implemented several frameworks and toolchains to overcome large-scale and real-world concurrent/parallel software debugging challenges. Our methods aim to facilitate the concurrent debugging process by providing efficient data collection and effective information retrieval mechanisms to target real-world software and bugs. First, we introduce PARLOT, a whole-program call tracing framework for HPC applications (MPI+X) that highly compresses the traces (up to more than 21000 times) while adding minimal overhead with an average required bandwidth of just 56 kB/s per core. Second, we present DiffTrace, a series of automated data abstraction, representation, and visualization techniques that differentiates the collected ParLOT traces and narrows the search down to just a few candidates of buggy traces. Finally, we illustrate GOAT, an end-to-end framework for automated tracing, analysis, and testing of concurrent Go applications. We also propose a set of coverage metrics to measure the quality of testing in CSP-like concurrent languages.

For my family

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Concurrent Software Correctness

In the life-cycle of computer programs, software testing and debugging play essential roles. Many approaches have been developed to test and debug software at different layers and from various aspects. Traditional and basic debugging techniques such as "printf" debugging, interactive debugging [4, 35], memory dumps, and profiling have been used widely in software development. However, modern concurrent system architectures and programming languages require novel software debugging techniques [34]. With the high growth in computation power and modern languages, programs are getting more sophisticated, complex, large, and heterogeneous to exploit the processing power efficiently (e.g., distributed memory and shared memory systems in Cloud and High-Performance Computing - HPC). Locating bugs in such programs is notoriously challenging because 1) the interleaving space grows exponentially with the number of processing units, 2) concurrent bugs are difficult to reproduce due to the non-deterministic nature of concurrent software, and 3) root-causing misbehaved executions of a concurrent program are non-trivial due to the complex interactions between concurrent program components. Researchers have developed techniques and tools to detect, prevent, and fix concurrent bugs in different levels of abstraction from multiple perspectives. Depending on the characteristics of target bugs [69], there are some advantages and limitations to each approach. Static analyzers (i.e., methods that analyze software without actually executing the program) offer rigorous guarantees, but they are often not practical for large-scale real-world programs. While dynamic analyzers (i.e., methods that analyze software based on execution evidence) cover a broader class of programs, they incur overhead to the program execution and may miss uncommon bugs. Hybrid analyzers (e.g., testing coverage analysis) combine ideas from both techniques to deliver reliable and practical debuggers. This dissertation has combined ideas from the broad spectrum of concurrency debugging approaches and implemented several frameworks and toolchains to assist parallel software developers in locating flaws in the program by automated and efficient data collection and analysis.

1.1.1 Background

Debugging concurrent programs require insight into the behavior of target bugs. A bug's behavior is described by its /textit[cause] and its *symptom* during execution. Adopting form the taxanomy proposed for concurrent Go bugs [104], we generally categorize common concurrency bugs in both shared and distrubutted memory systems based on the *symptom of the bug* (figure 1.1). This work aims not to focus on a specific bug class but rather to provide general tools to analyze the symptoms of unexpected behaviors in HPC and Cloud systems (i.e., shared memory and distributed memory systems) to help locate their cause.

1.1.1.1 Blocking Bugs

Blocking bugs lead the execution of the program into a *halt* state and prevent the program from making progress and reaching its final state.

- **Deadlock** is a situation where one or more processes in a system cannot proceed because they are blocked waiting for a shared resource held by another process. More generally, deadlock occurs when a process is waiting for an external *signal* from another process which is also attempting to acquire a resource held by other processes.

- **Livelock** is a condition where a thread is waiting for a resource that will never become available. It is similar to deadlock except that the state of the process involved in the livelock constantly changes regarding each other, non progressing.

- **Starvation** occurs in systems with different priorities for the execution of processes when one or more processes are constantly delayed indefinitely due to the higher priority processes receiving the required resources.

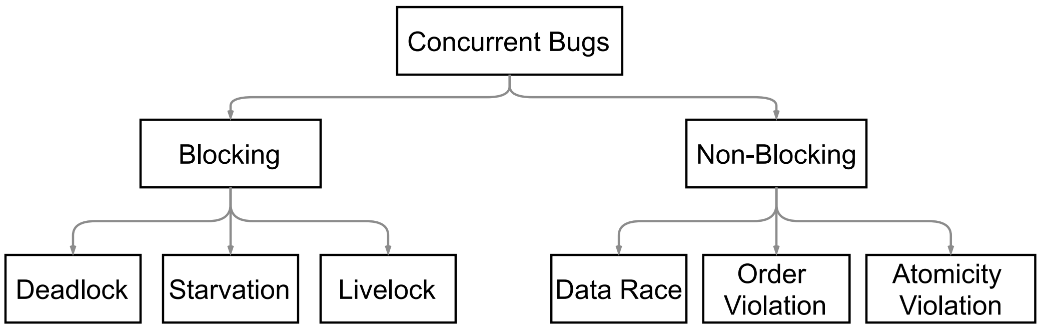


Figure 1.1: Taxanomy of concurrent bugs

1.1.1.2 Non-Blocking Bugs

Non-blocking bugs cause the program to produce incorrect results. The program execution terminates, but the generated output is invalidated because of the lack of proper *serilaization* for concurrent memory accesses.

- **Order violation** is the violation of the expected order of at least two memory accesses which produce corrupted results.
- **Atomicity violation** refers to the situation when two blocks of code (i.e., finite sequences of statements) execute concurrently and produce inconsistent results depending on the non-deterministic order of executions.
- **Data race** occurs when at least two concurrent threads access the same memory location, and at least one of the accesses is *write*.

1.1.1.3 Concurrent Debugging

Figure 1.2 displays the general taxonomy of methods for tackling concurrency debugging challenges. **Testing** exercises the program behavior over a set of inputs (manually written tests) or over the schedule space (feasible interleavings) to detect errors. Although testing effectively reveals flaws of the program, it does not guarantee the program's bug freedom. **Model checking** exhaustively test the program to cover possible control flow paths and verify the program properties. However, due to scalability problems, model checking is typically done symbolically and up to a certain bound; thus, they rarely achieve a guarantee. **Tracing** collects evidence from program execution, enabling offline analysis of the program's dynamic behavior. Similar to tracing, **Record and Replay** technique captures a single execution of the target program for later replay and analysis of the program's

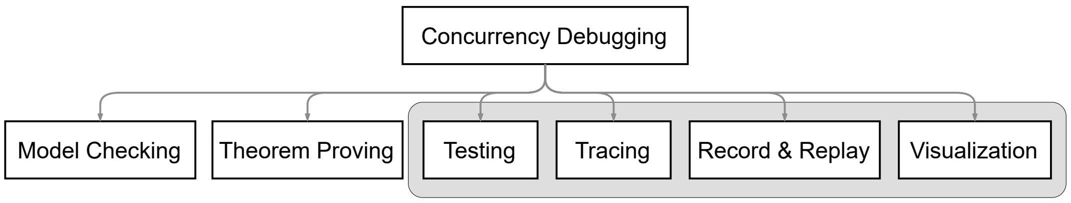


Figure 1.2: Taxanomy of concurrent debugging approaches. The highlighted area shows the techniques that we employed in this disseration.

non-deterministic behavior. Both methods usually incur overhead to the native execution of the program and require sophisticated mechanisms to make them practical for real-world programs and large-scale executions. **Theorem Proving** techniques, in principle, prove a system to be free of flaws often through verifying type systems. The enormous effort needed to use these tools makes them most appropriate for new implementations of small, critical cores. **Visualization** techniques provide transparent representations of complex concurrent execution of the program for developers to analyze visually.

In this dissertation, we combine and apply methods from *tracing* (chapters 2 and 4), *record and replay* (chapters 2,3 and 4), *testing* (chapter 4), and *visualization* (chapters 3 and 4) to facilitate the process of concurrent debugging.

1.2 Dissertation Statement

This dissertation aims to facilitate the concurrent debugging process by providing efficient data collection and effective information retrieval mechanisms to target real-world software and bugs. With that being said, our thesis statement is the following:

> Efficiently collecting data and systematically analyzing them is essential to gaining insight into the behavior of complex programs and help developers fix the flaws of the program. Also, given the rich set of concurrency primitives in modern languages, one needs to articulate nuanced concurrency coverage metrics and demonstrate their attainment.

1.3 Contributions of the Dissertation

In this dissertation, we present different contributions to facilitate concurrent debugging using efficient tools and automated frameworks.

First, we introduce PARLOT (Chapter 2), a new tracing approach that makes it possible to capture the whole-program call-return, call-stack, call-graph, and call-frequency information, including all library calls, for every thread and process of HPC applications at low overhead in both space and time. PARLOT is equipped with a new incremental data compression algorithm to drastically reduce the required tracing bandwidth (average of 56 kB/s per core), thus enabling the collection of whole-program traces, which would be infeasible without on-the-fly compression. PARLOT can instrument x86 applications at the binary level (regardless of the source language used) to collect whole-program call traces with the compression ratio of up to 21,000.

Second, we present DiffTrace (Chapter 3), a series of automated data abstraction, representation, and visualization techniques that differentiate the collected PARLOT traces and narrows the search space down to just a few candidates of buggy traces. In DiffTrace, we employ *concept lattices* to amalgamate the collected traces and hierarchically cluster them based on features extracted from the structure of traces. DiffTrace also utilizes the rigorous notion of Nested Loop Representations (NLRs) for summarizing traces and representing loops in a manner that informs the developer engaged in debugging. We evaluate the effectivness of DiffTrace by locating artificial bugs injected to ILCS [12], a hybrid (MPI + OpenMP) HPC framework.

Third, we illustrate GOAT (Chapter 4), a testing and analysis framework for concurrent Go applications that provides for whole-program trace collection (via an enhancement to the standard tracer package) and knowledge discovery about the program's dynamic behavior. We show the effectiveness of controlled preemptions for concurrency bug exposure in the context of a real-world language. We demonstrate the effectivness of GOAT by detecting all 68 GoKer [112] blocking bugs, many of which are undetected by existing tools. Lastly, we propose a set of coverage requirements to measure testing quality in CSP-like concurrent languages. The proposed metrics characterize the dynamic behavior of concurrency primitives, enabling measurement of quality and progress of schedule-space exploration.

1.4 Organization of the Dissertation

This dissertation is organized as follows: In Chapter 2, we present the design and evaluation of PARLOT, the whole-program tracing mechanism; Chapter 3 illustrates the series of trace abstractions and representations towards locating the flaw by clustering and diffing traces; with Chapter 4, we describe the end-to-end analysis and testing framework for concurrent Go applications called GOAT, and propose a set of concurrency coverage metrics to measure the quality of schedule space exploration; finally, in Chapter 5, we summarize all the contributions and conclude the dissertation.

CHAPTER 2

WHOLE-PROGRAM DYAMIC TRACING

This chapter is based on the work published at the Workshop on Programming and Performance Visualization Tools (ESPT) 2018 [99][1]. We present PARLOT, a framework for efficient whole-program call tracing for HPC (MPI+X) applications using Intel Pin [70] dynamic binary isntrumentation that includes following key features: (1) It describes a technique that makes low-overhead on-the-fly compression of whole-program call traces feasible. (2) It presents a new, efficient, incremental trace-compression approach that reduces the trace volume dynamically, which lowers not only the needed bandwidth but also the tracing overhead. (3) It collects all caller/callee relations, call frequencies, call stacks, as well as the full trace of all calls and returns executed by each thread, including in library code. (4) It works on top of existing dynamic binary instrumentation tools, thus requiring neither source-code modifications nor recompilation. (5) It supports program analysis and debugging at the thread, thread-group, and program level. PARLOT establishes that comparable capabilities are currently unavailable. Our experiments with the NAS parallel benchmarks running on the Comet supercomputer with up to 1,024 cores show that ParLoT can collect whole-program function-call traces at an average tracing bandwidth of just 56 kB/s per core.

2.1 Introduction

Understanding and debugging HPC programs is time-consuming for the user and computationally inefficient. This is especially true when one has to track control flow in terms of function calls and returns that may span library and system codes. Traditional

[1]©2019 Springer. Adapted, with permission, from Taheri S., Devale S., Gopalakrishnan G., Burtscher M. (2019) ParLoT: Efficient Whole-Program Call Tracing for HPC Applications. In: Bhatele A., Boehme D., Levine J., Malony A., Schulz M. (eds) Programming and Performance Visualization Tools. ESPT 2017, ESPT 2018, VPA 2017, VPA 2018. Lecture Notes in Computer Science, vol 11027. Springer, Cham. https://doi.org/10.1007/978-3-030-17872-7_10

software engineering quality assurance methods are often inapplicable to HPC where concurrency combined with large problem scales and sophisticated domain-specific math can make programming very challenging. For example, it took months for scientists to debug an MPI laser-plasma interaction code [34].

HPC bugs may be a combination of both flawed program logic and unspecified or illegal interactions between various concurrency models (e.g., PThreads, MPI, OpenMP, etc.) that coexist in most large applications [34]. Moreover, HPC software tends to consume vast amounts of CPU time and hardware resources. Reproducing bugs by rerunning the application is therefore expensive and undesirable. A natural and field-proven approach for debugging is to capture detailed execution traces and compare the traces against corresponding traces from previous (stable) runs [3,18]. A *key requirement* is to do this collection *as efficiently as possible* and in *as general and comprehensive a manner* as possible.

Existing tools in this space do not meet our criteria for efficiency and generality. The highly acclaimed STAT [3] tool has helped isolate bugs based on building equivalence classes of MPI processes and spotting outliers. We would like to go beyond the capabilities offered by STAT and support the collection of *whole-program* traces that can then be employed by a gamut of back-end tools. Also, STAT is usually brought into the picture when a failure (e.g., a deadlock or hang) is encountered. We would like to move toward an "always on" collection regime, as we cannot anticipate when a failure will occur – or, more importantly, *whether the failure will be reproducible.* There are no reported debugging studies on using STAT in continuous collection ("always on") mode. In CSTG [18], the collection is orchestrated by the user around chosen collection points and employs heavy-weight unix `backtrace` calls. These again are different from ParLoT, where collection points would not be a priori chosen.

The thrust of the work in this paper is to avoid many of the drawbacks of existing tracing-based tools. We are interested in avoiding source-code modifications and recompilation — thus making binary instrumentation-based tools the only practical and widely deployable option. We also believe in the value of creating tools that are *portable across a wide variety of platforms*.

Our goal is to use *compression* for trace aggregation and to offer a generic and low-overhead tracing method that (1) collects dynamic call information during execution (all function calls and returns) for debugging, performance evaluation, phase detection [85], etc., (2) has low overhead, (3) and requires little tracing bandwidth. *Providing all these features in a single tool that operates based on binary instrumentation is an unsolved problem.* In this paper, we describe a new tracing approach that fulfills these requirements, which we implemented in our proof-of-concept PARLOT tool.

With PARLOT, users can easily build a host of post-processors to examine executions from many vantage points. For instance, they can write post-processors to detect unexpected (or "outlier") executions. If needed, they can drill down and detect abnormal behaviors *even in the runtime and support library stack* such as MPI-level activities. In HPC, it is well-known (especially on newer machines) that bugs are often due to broken libraries (MPI, OpenMP), a broken runtime, or OS-level activities. Having a single low-overhead tool that can "X-ray" an application to this depth is a goal met by PARLOT— a unique feature in today's tool eco-system.

To further motivate the need for whole-program function call traces, consider the expression `f()+g()`. In C, there is no sequence point associated with the + operator [81]. If these function calls have inadvertent side-effects causing failure, it is important to know in which order `f()` and `g()` were invoked—something that is easy to discern using PARLOT's traces. One may be concerned that such a tool introduces excessive execution slowdown. PARLOT goes to great lengths to minimize these overheads to a level that we believe most users will find acceptable. The mindset is to *"pay a little upfront to dramatically reduce the number of overall debug iterations"*.

As proof of concept, we gathered preliminary results from using the PARLOT tracing mechanism to compare different runs. We injected various bugs into the MPI-related functions of ILCS [12], a parallelization framework for iterative local searches. We ran PARLOT on top of executions of buggy and bug-free versions of ILCS and collected traces. Since PARLOT's traces maintain the order of the function calls, we were able to split the traces at multiple points of interest and to feed different chunks of traces to a Concept Lattice data structure [28] [32]. Having the totally ordered sequence of function calls of the whole program for each active process/thread enabled us to quickly narrow down the search space to locate the cause of the abnormal behavior in the buggy version of ILCS.

This paper does not pursue debugging per se but rather a thorough benchmarking of

ParLoT. It makes the following main contributions:

- It introduces a new tracing approach that makes it possible to capture the whole-program call-return, call-stack, call-graph, and call-frequency information, including all library calls, for every thread and process of HPC applications at low overhead in both space and time.
- It describes a new incremental data compression algorithm to drastically reduce the required tracing bandwidth, thus enabling the collection of whole-program traces, which would be infeasible without on-the-fly compression.
- It presents ParLoT, a proof-of-concept tool that implements our compression-based low-overhead tracing approach. ParLoT is capable of instrumenting x86 applications at the binary level (regardless of the source language used) to collect whole-program call traces.

The remainder of this paper is organized as follows. Section 2.2 introduces the basic ideas and infrastructure behind ParLoT and other tracing tools. Section 2.3 describes the design of ParLoT in detail. Sections 2.4 and 2.5 present our evaluation of different aspects of ParLoT and compare it with a similar tool. Section 2.6 concludes the paper with a summary and future work.

2.2 Background and Related Work

Recording a log of events during the execution of an application is essential for better understanding the program behavior and, in case of a failure, to locate the problem. Recording this type of information requires instrumentation of the program either at the source-code or the binary-code level. Instrumenting the source code by adding extra libraries and statements to collect the desired information is easy for developers. However, doing so modifies the code and requires recompilation, often involving multiple different tools and complex hierarchies of makefiles and libraries, which can make this approach cumbersome and frustrating for users. Instrumenting an executable at the binary level using a tool is typically easier, faster, and less error prone for most users. Moreover, binary instrumentation is language independent, portable to any system that has the appropriate instrumentation tool installed, and provides machine-level insight into the behavior of the application.

2.2.1 Binary Instrumentation

Executables can be instrumented *statically*, where the additional code is inserted into the binary before execution, which results in a persistent modified executable, or *dynamically*, where the modification of the executable is not permanent. In dynamic binary instrumentation, code behavior can be monitored at runtime, making it possible to handle dynamically-generated and self-modifying code. Furthermore, it may be feasible to attach the instrumentation to a running process, which is particularly useful for long-running applications and infinite loops.

Many different tools for investigating application behavior have been designed on top of such Dynamic Binary Instrumentation (DBI) frameworks. For instance, Dyninst [72] provides a dynamic instrumentation API that gives developers the ability to measure various performance aspects. It is used in tools like Open-SpeedShop [90] and TAU [92] as well as correctness debuggers like STAT [3]. Moreover, VampirTrace [57] uses it to provide a library for collecting program execution logs.

Valgrind [79] is a shadow-value DBI framework that keeps a copy of every register and memory location. It provides developers with the ability to instrument system calls and instructions. Error detectors such as Memcheck [80] and call-graph generators like CALLGRIND [108] are built upon Valgrind.[2]

We implemented PARLOT on top of PIN [70], a DBI framework for the IA-32, x86-64, and MIC instruction-set architectures for creating dynamic program analysis tools. There is also version of PIN available for the ARM architecture [36]. PARLOT mutates PIN to trace the entry (call) and exit (return) of every executed function. Note that our tracing and compression approaches can equally be implemented on top of other instrumentation tools. For example, PMaC [102] is a DBI tool for the PowerPC/AIX architecture upon which PARLOT could also be based.

[2]Given the absence of tools similar to PARLOT, we employ CALLGRIND as a "close-enough" tool in our comparisons elaborated in §2.4.3. In this capacity, CALLGRIND is similar to PARLOT(M), a variant of PARLOT that only collects traces from the `main` image. We perform such comparison to have an idea of how we fare with respect to one other tool. In §2.5, we also present a self-assessment of PARLOT separately.

2.2.2 Efficient Tracing

When dealing with large-scale parallel programs, any attempt to capture reasonably frequent events will result in a vast amount of data. Moreover, transferring and storing the data will incur significant overhead. For example, collecting just one byte of information per executed instruction yields on the order of a gigabyte of data per second on a single high-end core. Storing the resulting multi-gigabyte traces from many cores can be a challenge, even on today's large hard disks.

Hence, to be able to capture whole-program call traces, we need a way to decrease the space and runtime overhead. *Compression* can encode the generated data using a smaller number of bits, help reduce the amount of data movement across the memory hierarchy, and lower storage and network demands. Although the encoded data will later have to be decoded for analysis, compressing them during tracing enables the collection of *whole-program* traces.

The use of compression by itself is not new. Various performance evaluation tools [1, 61, 92] already employ compression during the collection of performance analysis data. Tools such as ScalaTrace [83] also exploit the repetitive nature of time-step simulations [26]. Aguilar et al. [2] proposed a lossy compression mechanism using the Nami library [27] for online MPI tracing. Mohror and Karavanic [74] investigated similarity-based trace reduction techniques to store and analyze traces at scale.

Many performance and debugging tools for HPC applications [3, 78] rely on mechanisms such as MRNet [87] to accelerate the collection and aggregation of traces based on an overlay network to overcome the challenge of massive data movement and analysis. However, our experiments show that, due to the high compression ratio of ParLoT traces, such mechanisms for data movement and aggregation may be unnecessary.

The novelty offered by ParLoT lies in the combination of compression speed, efficacy, and low timing jitter made possible by its *incremental* lossless compression algorithm, which is described in §2.3. It immediately compresses all traced information while the application is running, that is, ParLoT does not record the uncompressed trace in memory. As a result, just a few kilobytes of data need to be written out per thread and per second, thus requiring only a small fraction of the disk or network bandwidth. The traces are decompressed later when they are read for offline analysis. From the decompressed full function-call trace, the complete call-graph, call-frequency, and caller-callee information can be extracted. This can be done at the granularity of a thread, a group of threads, or the whole application. We now elaborate on the design of PARLOT that makes these innovations possible.

2.3 Design of PARLOT

Our experimental results in §2.5 highlight why *compression* is essential to make our approach work. We used PARLOT to record a unique 16-bit identifier for every function call and return. Tracing just this small amount of information without compression when running the Mantevo miniapps [37] on Stampede 1 resulted in about 2 MB/s of data per core on average. Extrapolating this value to all 102,400 cores of Stampede 1 (not counting the accelerators) yields 205 GB/s of trace data, which exceeds the Lustre filesystem's parallel write performance of 150 GB/s. Enabling PARLOT's compression algorithm reduced the emitted trace data by a factor of 100 on average, a ratio that is quite stable w.r.t scaling, making it possible to trace full-scale programs while leaving over 98% of the I/O bandwidth to the application. Therefore, PARLOT should also work for codes with higher bandwidth requirements than the ones we tested.

Figure 2.1 provides a general overview of PARLOT's workflow. Basic blocks within program executables are *dynamically* instrumented before being executed. The collected data are compressed on-the-fly at runtime.

2.3.1 Tracing Operation

PARLOT uses the PIN API as its instrumentation mechanism to gather traces. In particular, it instructs PIN to instrument every thread launch and termination in the application as well as every function entry and exit. The thread-launch instrumentation code initializes the per-thread tracing variables and opens a file into which the trace data from that thread will be written. The thread-termination code finalizes any ongoing compression, flushes out any remaining entries, and closes the trace file. PARLOT assigns every static function in each image (main program and all libraries) a unique unsigned 16-bit ID, which it records in a separate file together with the image and function name. This file allows the trace reader to map IDs back to function-name/image pairs.

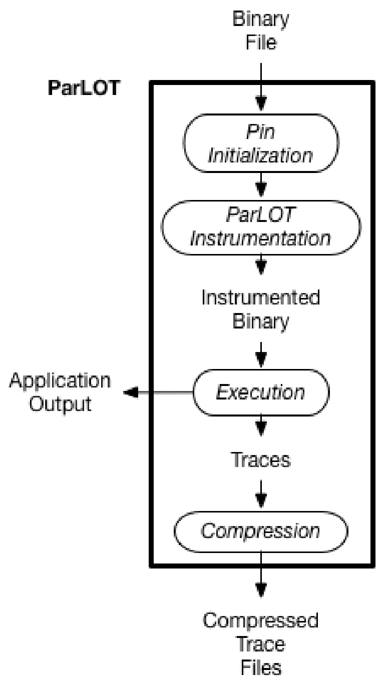


Figure 2.1: Overview of PARLOT

For every function *entry*, PARLOT executes extra code that has access to the thread ID, function ID, and current stack-pointer (SP) value. Based on the SP value, it performs call-stack correction if necessary (see §2.3.4), adds the new function to a data structure it maintains that holds the call stack (which is separate from the application's runtime stack), and emits the function ID into the trace file via an incremental compression algorithm (see §2.3.2). All of this is done independently for each thread. Similarly, for every function *exit*, PARLOT also executes extra code that has access to the thread ID, function ID, and current SP value. Based on the SP value, it performs call-stack correction if necessary, removes the function from its call-stack data structure, and emits the reserved function ID of zero into the trace file to indicate an exit. As before, this is done via an incremental compression algorithm. We use zero for all exits rather than emitting the function ID and a bit to specify whether it is an entry or exit because using zeros results in more compressible output. This way, half of the values in the trace will be zero.

2.3.2 Incremental Compression

PARLOT immediately compresses the traced information even before it is written to memory. It does, however, keep a sliding window (circular buffer) of the most recent uncompressed trace events, which is needed by the compressor. It compresses each function ID before the next function ID is known. The conventional approach would be to first record uncompressed function IDs in a buffer and later compress the whole buffer once it fills up. However, this makes the processing time very non-uniform. Whereas almost all function IDs can be recorded very quickly since they just have to be written to the buffer, processing a function ID that happens to fill the buffer takes a long time as it triggers the compression of the entire buffer. This results in sporadic blocking of threads during which time they make no progress towards executing the application code. Initial experiments revealed that such behavior can be detrimental when one thread is polling data from another thread that is currently blocked due to compression. For example, we observed a several order of magnitude increase in entry/exit events of an internal MPI library function when using block-based compression.

To remedy this situation, the compressor must operate incrementally, i.e., each piece of trace data must be compressed when it is generated, without buffering it first, to ensure that there is never a long-latency compression delay. Few existing compression algorithms have been implemented in such a manner because it is more difficult to code up and probably a little slower. Nevertheless, we were able to implement our algorithm (discussed next) in this way so that each trace event is compressed with similar latency.

2.3.3 Compression Algorithm

We used the CRUSHER framework [11,14,16,109] to automatically synthesize an effective and fast lossless compression algorithm for our traces. CRUSHER is based on a library of data transformations extracted from various compression algorithms. It combines these transformations in all possible ways to generate algorithm candidates, which it then evaluates on a set of training data. We gathered uncompressed traces from some of the Mantevo miniapps [37] for this purpose. This evaluation revealed that a particular word-level Lempel-Ziv (LZ) transformation followed by a byte-level Zero-Elimination (ZE) transformation works well. In other words, ParLOT's trace entries, which are two-byte words, are first transformed using LZ. The output is interpreted as a sequence of bytes, which is transformed using ZE for further compression. The output of ZE is written to secondary storage.

LZ implements a variant of the LZ77 algorithm [116]. It uses a 4096-entry hash table to identify the most recent prior occurrence of the current value in the trace. Then it checks whether the three values immediately before that location match the three trace entries just before the current location. If they do not, the current trace entry is emitted and LZ advances to the next entry. If the three values match, LZ counts how many values following the current value match the values following that location. The length of the matching substring is emitted and LZ advances by that many values. Note that all of this is done incrementally. The history of previous trace entries available to LZ for finding matches is maintained in a 64k-entry circular buffer.

ZE emits a bitmap in which each bit represents one input byte. The bits indicate whether the corresponding bytes are zero or not. Following each eight-bit bitmap, ZE emits the non-zero bytes.

As mentioned above, we had to implement the two transformations incrementally to minimize the maximum latency. This required breaking them up into multiple pieces. Depending on the state the compressor is in when the next trace entry needs to be processed, the appropriate piece of code is executed and the state updated. If the LZ code produces an output, which it only does some of the time, then the appropriate piece of the ZE code is executed in a similar manner.

2.3.4 PIN and Call-Stack Correction

To be able to decode the trace, i.e., to correctly associate each exit with the function entry it belongs to, our trace reader maintains an identical call-stack data structure. Unfortunately, and as pointed out in the PIN documentation [53], it is not always possible to identify all function exits. For example, in optimized code, a function's instructions may be inlined and interleaved with the caller's instructions, making it sometimes infeasible for PIN to identify the exit. As a consequence, we have to ensure that ParLOT works correctly even when PIN misses an exit. This is why the SP values are needed.

During tracing, ParLOT not only records the function IDs in its call stack but also the associated SP values. This enables it to detect missing exits and to correct the call stack accordingly. Whenever a function is entered, it checks if there is at least one entry in the call stack and, if so, whether its SP value is higher than that of the current SP. If it is lower, we must have missed at least one exit since the runtime stack grows downwards (the SP value decreases with every function entry and increases with every exit). If a missing exit is detected in this manner, PARLOT pops the top element from its call stack and emits a zero to indicate a function exit. It repeats this procedure until the stack is empty or its top entry has a sufficiently high SP value. The same call-stack correction technique is applied for every function exit whose SP value is inconsistent. Note that the SP values are only used for this purpose and are not included in the compressed trace.

The result is an internally consistent trace of function entry and exit events, meaning that parsing the trace will yield a correct call stack. This is essential so that the trace can be decoded properly. Moreover, it means that the trace includes exits that truly happened in the application but that were missed by PIN. Note, however, that our call-stack correction is a best-effort approach and may, in rare cases, temporarily not reflect what the application actually did. For example, this can happen for functions that do not create a frame on the runtime stack. When implementing PARLOT on top of another DBI framework, call-stack correction may not be needed, resulting in even lower PARLOT overhead.

2.4 Evaluation Methodology

2.4.1 Benchmarks and System

We performed our evaluations on the MPI-based NAS Parallel Benchmarks (NPB) [6]. NPB includes four inputs sizes. To keep the runtimes reasonable, we show results for the class *B* (small-medium) and class *C* (medium-large) inputs.

We compiled the NPB codes with the mpicc and mpif77 wrappers of MVAPICH 2.2.1, which are based on icc/ifort 14.0.2 using the prescribed -g and -O1 optimization flags. Quick tests showed that higher optimization levels do not significantly improve the performance.

We ran all experiments on Comet at the San Diego Supercomputer Center [94], whose filesystem is NFS and Lustre. Comet has 1944 compute nodes, each of which has dual-socket Intel Xeon E5-2680 v3 processors with a total of 28 cores (14 per socket) and 128 GB of main memory. Note that we only used 16 cores per node as many of the NPB programs require a core count that is a power of two. To study the scaling behavior, we ran experiments on 1, 4, 16 and 64 compute nodes, i.e., on up to 1024 cores.

2.4.2 Metrics

We use the following metrics to quantify and compare the performance of the tracing tools. Unless otherwise noted, all results are based on the median of three identical experiments.

- The **tracing overhead** is the runtime of the target application when it is being traced divided by the runtime of the same application without tracing. This lower-is-better ratio measures by how much the tracing (and the compression when enabled) slows down the target application.

- The **tracing bandwidth** is the size of the trace information divided by the application runtime. To make the results easier to compare, we generally list the tracing bandwidth per core, i.e., the tracing bandwidth divided by the number of cores used. This lower-is-better metric is expressed in kilobytes per second (kB/s) per core. It specifies the average needed bandwidth to record the trace data.

- The **compression ratio** is the size of the uncompressed trace divided by the size of the generated (compressed) trace. This higher-is-better ratio measures the factor by which the compression reduces the trace size. In other words, without compression, the tracing bandwidth would be higher by this factor.

2.4.3 Tracing Tools

We compare our ParLoT tool, implemented on top of PIN 3.5, with Callgrind 3.13. ParLoT was compiled with gcc 4.9.2 using PIN's make system and Callgrind with Valgrind's make system. We created the following versions of ParLoT to evaluate different aspects of its design.

- **ParLoT(m)** is the normal ParLoT tool configured to only collect function-call traces from the main image of the application.

- **ParLoT(a)** is the normal ParLoT tool configured to collect function-call traces from all images of the application, including library function calls.

- **Pin-Init** is a crippled version of ParLoT from which the tracing code has been removed. The purpose of Pin-Init is to see how much of the overhead is due to PIN.
- **ParLoT-NC** is the normal ParLoT tool but with compression disabled. It writes out the captured data in uncompressed form. The purpose of ParLoT-NC is to show the performance impact of the compression.

It proved surprisingly difficult to find a tool that is similar to ParLoT because there appear to be no other tools that generate whole program call traces. In the end, we settled on Callgrind as the most similar tool we could find and used it for our comparisons. Callgrind is based on the Valgrind DBI tool. It collects function-call graphs combined with performance data to show the user what portion of the execution time has been spent

Table 2.1: Overhead added by each tool

Input	Tool	# Nodes	bt	cg	ep	ft	is	lu	mg	sp	GM
B	ParLoT(m)	1	1.6	1.8	2.6	2.1	2.5	1.3	2.5	1.3	1.9
		4	1.8	1.9	1.9	1.7	1.8	1.8	1.5	1.7	1.8
		16	2.2	2.6	2.0	1.9	1.8	2.7	2.4	2.2	2.2
		64	2.1	2.2	2.4	2.0	4.3	4.4	2.0	2.1	2.5
		AVG	1.9	2.1	2.2	1.9	2.6	2.6	2.1	1.8	**2.1**
	ParLoT(a)	1	1.8	2.7	4.2	2.8	4.2	1.7	4.8	1.7	2.8
		4	2.6	3.1	3.4	2.8	3.0	2.8	2.8	2.7	2.9
		16	3.5	4.2	3.4	2.9	2.8	4.3	4.5	3.7	3.6
		64	3.1	3.3	3.8	3.0	5.4	4.7	3.2	3.3	3.7
		AVG	2.8	3.3	3.7	2.9	3.9	3.4	3.8	2.8	**3.2**
	Callgrind	1	8.6	6.0	8.9	10.1	2.5	7.5	3.3	6.6	6.1
		4	6.0	3.6	2.9	3.5	1.5	5.2	1.2	5.8	3.2
		16	4.3	3.3	2.2	2.2	1.7	4.6	1.8	4.3	2.8
		64	2.3	2.0	1.7	2.1	4.1	4.0	1.5	2.5	2.3
		AVG	5.3	3.7	3.9	4.5	2.4	5.3	2.0	4.8	**3.6**
C	ParLoT(m)	1	1.4	1.3	2.5	1.9	2.3	1.1	1.7	1.1	1.6
		4	1.6	1.7	1.8	1.6	1.7	1.3	1.8	1.4	1.6
		16	1.8	2.4	2.5	1.5	1.8	2.2	2.4	1.8	2.0
		64	2.2	2.7	2.4	1.6	4.5	3.4	2.4	2.2	2.6
		AVG	1.8	2.0	2.3	1.7	2.6	2.0	2.1	1.6	**1.9**
	ParLoT(a)	1	1.5	1.6	3.2	2.0	2.8	1.2	2.5	1.2	1.9
		4	1.9	2.4	2.6	2.1	2.6	1.7	3.1	1.7	2.2
		16	2.7	3.5	4.1	2.1	2.8	3.2	4.0	2.5	3.0
		64	3.6	4.1	4.2	2.2	5.5	4.4	4.2	3.0	3.8
		AVG	2.4	2.9	3.5	2.1	3.4	2.6	3.5	2.1	**2.7**
	Callgrind	1	8.5	4.4	13.2	13.1	3.3	7.9	5.9	5.1	6.9
		4	8.7	4.5	4.8	6.4	1.7	6.4	2.8	6.3	4.6
		16	6.9	3.9	3.1	2.8	1.8	6.4	2.1	6.1	3.7
		64	4.4	3.5	2.1	2.5	4.2	5.2	2.1	3.8	3.3
		AVG	7.1	4.1	5.8	6.2	2.8	6.5	3.2	5.3	**4.6**

in each function.

Each CALLGRIND trace file contains a sequence of function names (or their code) plus numerical data for each function on its caller-callee relationship with other functions. Moreover, it contains cost information for each function in terms of how many machine instructions it read. This information is collected using hardware performance counters. The format of the file is plain ASCII text. Interestingly, all numerical values are expressed relative to previous values, i.e., they are delta (or difference) encoded. This simple form of compression is enabled by default in CALLGRIND.

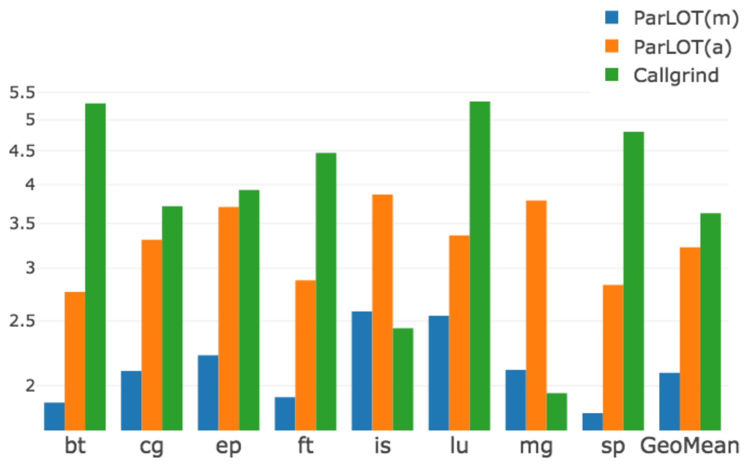


Figure 2.2: Average tracing overhead on the NPB applications - Input B

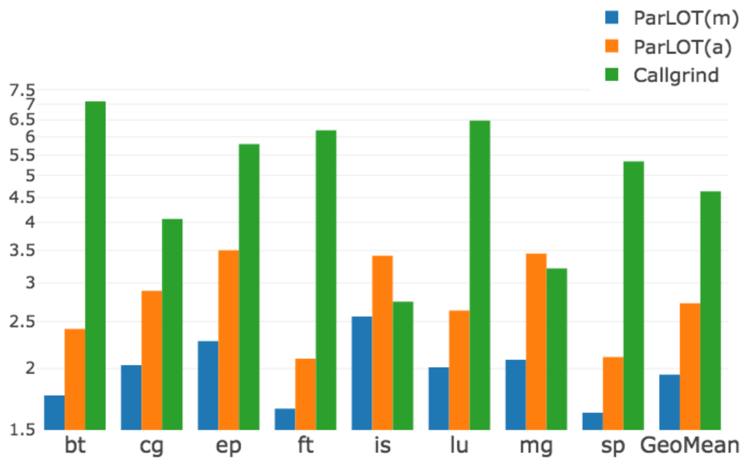


Figure 2.3: Average tracing overhead on the NPB applications - Input C

We believe the information traced by CALLGRIND is reasonably similar to the information traced by PARLOT(M). Whereas CALLGRIND's traces include performance data that PARLOT does not capture, PARLOT records the whole-program call trace, which CALLGRIND does not capture. The full function-call trace is a strict superset of the call-graph information that CALLGRIND records because the call graph can be extracted from the function-call trace but not vice versa. In particular, CALLGRIND cannot recreate the order of the function calls a thread made whereas PARLOT can.

2.5 Results

2.5.1 Tracing Overhead

Table 2.1 shows the tracing overhead of PARLOT(M), PARLOT(A), and CALLGRIND on each application of the NPB benchmark suite for different node counts. The last column of the table lists the geometric mean over all eight programs. The AVG rows show the average over the four node counts.

On average, both PARLOT(M) and PARLOT(A) outperform CALLGRIND. The bolded numbers in Table 2.1 for input C show that the average overhead is 1.94 for PARLOT(M), 2.73 for PARLOT(A), and 4.63 for CALLGRIND. Figures 2.2 and 2.3 show these results in visual form.

The key takeaway point is that the overhead of PARLOT is roughly a factor of two to three, which we believe users may be willing to accept, for example, if it helps them

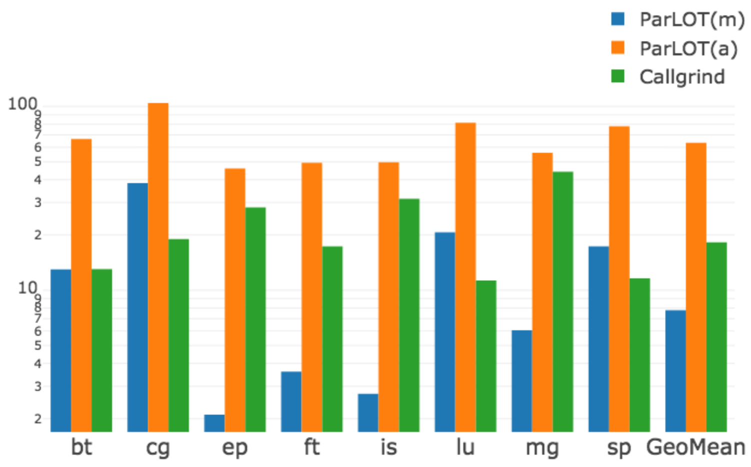


Figure 2.4: Average required bandwidth per core (kB/s) on the NPB applications - Input B

Table 2.2: Required bandwidth per core (kB/s)

Input	Tool	# Nodes	bt	cg	ep	ft	is	lu	mg	sp	GM
		1	4.7	21.9	3.8	1.5	0.8	2.4	5.6	5.4	3.7
		4	14.3	41.1	1.9	3.5	2.2	21.5	6.5	15.9	8.1
	ParLoT(m)	16	14.3	46.6	1.5	4.9	3.4	31.8	6.5	18.6	9.4
		64	18.6	43.6	1.3	4.6	4.5	27.1	5.6	29.6	9.9
		AVG	13.0	38.3	2.1	3.6	2.7	20.7	6.1	17.4	**7.8**
		1	48.7	89.4	47.2	45.6	60.0	53.6	60.8	54.3	56.2
		4	61.8	101.2	45.2	55.1	53.2	71.1	54.9	73.6	62.7
B	ParLoT(a)	16	74.0	116.9	47.4	48.9	47.8	100.9	55.8	84.6	68.0
		64	81.8	110.2	44.2	48.0	37.8	100.3	52.7	99.9	66.5
		AVG	66.6	104.4	46.0	49.4	49.7	81.5	56.0	78.1	**63.3**
		1	1.6	7.7	7.4	4.6	39.5	2.6	34.4	2.7	6.7
		4	6.5	16.0	22.1	15.7	45.5	8.6	45.5	7.8	16.3
	Callgrind	16	17.2	24.6	37.4	23.8	29.9	16.2	51.5	15.8	24.9
		64	26.8	27.7	45.9	25.1	11.0	17.8	45.3	20.2	25.0
		AVG	13.0	19.0	28.2	17.3	31.5	11.3	44.2	11.6	**18.2**
		1	1.8	17.0	5.2	1.2	0.7	0.8	3.6	1.4	2.2
		4	7.5	44.9	3.0	2.5	2.1	20.1	7.1	13.7	7.6
	ParLoT(m)	16	16.3	55.0	1.8	6.1	3.4	34.1	7.2	20.7	10.7
		64	17.5	61.4	1.3	5.9	4.4	38.3	5.6	26.1	10.9
		AVG	10.8	44.6	2.8	3.9	2.7	23.3	5.9	15.5	**7.8**
		1	17.8	53.4	26.3	20.9	48.3	25.3	52.6	19.5	30.0
		4	51.8	95.8	36.8	43.8	51.4	58.4	54.2	65.8	55.2
C	ParLoT(a)	16	75.4	121.0	44.3	61.4	46.9	101.1	56.5	101.3	71.4
		64	80.6	135.2	43.5	46.3	37.1	117.9	54.1	99.0	69.0
		AVG	56.4	101.4	37.7	43.1	45.9	75.7	54.3	71.4	**56.4**
		1	0.4	3.1	2.0	1.1	14.6	0.7	7.0	0.8	1.9
		4	1.8	8.9	7.7	4.5	31.7	2.8	21.0	2.8	6.4
	Callgrind	16	6.0	15.8	22.9	10.8	26.5	7.5	39.1	7.0	13.7
		64	14.3	19.6	35.8	12.2	11.1	11.9	40.7	12.8	17.4
		AVG	5.6	11.8	17.1	7.1	21.0	5.7	26.9	5.8	**9.8**

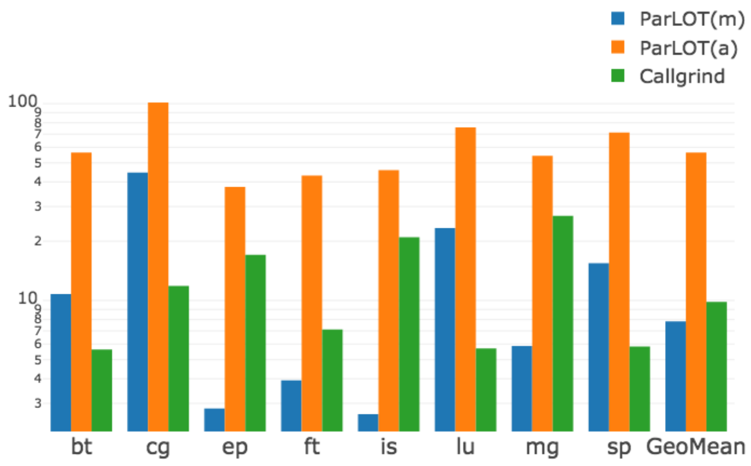


Figure 2.5: Average required bandwidth per core (kB/s) on the NPB applications - Input C

debug their applications. This is promising especially when considering how detailed the collected trace information is and that most of the overhead is due to PIN (see §2.5.4). Note that PARLOT's overhead is typically lower than that of CALLGRIND, which collects less information.

The overhead of PARLOT increases as we scale the applications to more compute nodes. However, the increase is quite small. Going from 16 to 1024 cores, a 64-fold increase in parallelism, only increases the average overhead by between 1.3- and 2.1-fold. In contrast, CALLGRIND's overhead decreases with increasing node count, making it more scalable. Having said that, CALLGRIND's overhead is larger for the C inputs whereas PARLOT's overhead is larger for the smaller B inputs. In other words, PARLOT scales better to larger inputs than CALLGRIND.

PARLOT's scaling behavior can be explained by correlating it with the expected function-call frequency. When distributing a fixed problem size over more cores, each core receives less work. As a consequence, less time is spent in the functions that process the work, resulting in more function calls per time unit, which causes more work for PARLOT. In contrast, when distributing a larger problem size over the same number of cores, each core receives more work. Hence, more time is spent in the functions that process the work, resulting in fewer function calls per time unit, which causes less work for PARLOT and therefore less tracing overhead. Hence, we believe PARLOT's overhead to be even lower on long-running inputs, which is where our tracing technique is needed the most.

In summary, PARLOT's overhead is in the single digits for all evaluated applications and configurations, including for 1024-core runs. It appears to scale reasonably to larger node counts and well to larger problem sizes.

2.5.2 Required Bandwidth

Table 2.2, Fig. 2.4 and Fig. 2.5 show how much trace bandwidth each tool requires during the application execution. On average, PARLOT(M) requires less bandwidth than CALLGRIND, especially for smaller inputs. PARLOT(A)'s bandwidth is much higher as it collects call information from all images and not just the main image like PARLOT(M) does.

We see that the required bandwidth for different input sizes of the NPB applications are almost equal in PARLOT. According to the NPB documentation, the number of iterations for inputs B and C are the same for all applications. They only differ in the number of elements or the grid size. It is clear that the required bandwidth of PARLOT is independent of the problem size, unlike CALLGRIND, where the input size has a linear impact on the results.

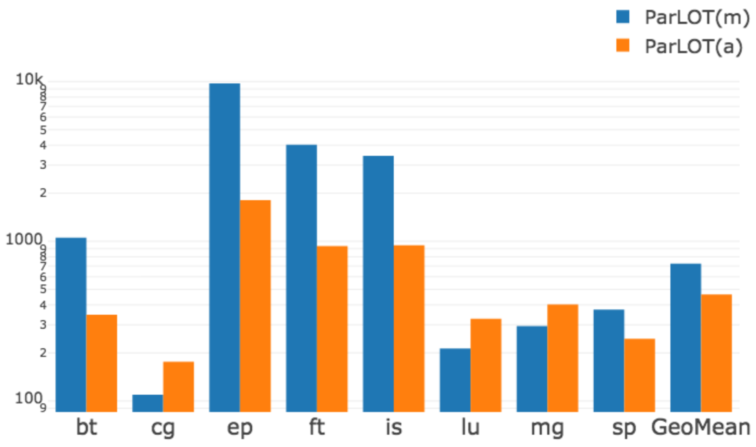


Figure 2.6: Average compression ratio of PARLOT on the NPB applications - Input B

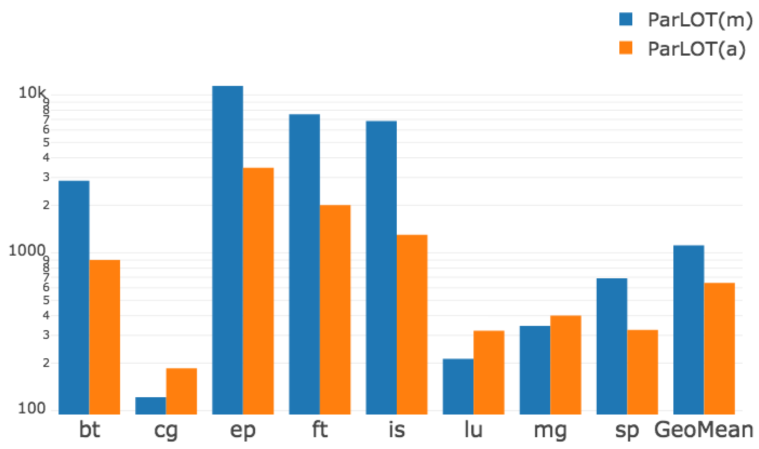


Figure 2.7: Average compression ratio of PARLOT on the NPB applications - Input C

2.5.3 Compression Ratio

Table 2.3 shows the compression ratios for all configurations and inputs. On average, PARLOT stores between half a kilobyte and a kilobyte of trace information in a single byte. We observe that the average compression ratio for PARLOT(A) on input C is 644.3, and its corresponding required bandwidth from Table 2.2 is 56.4 kB/s. This means PARLOT can collect **more than 36 MB** worth of data per core per second while only needing 56

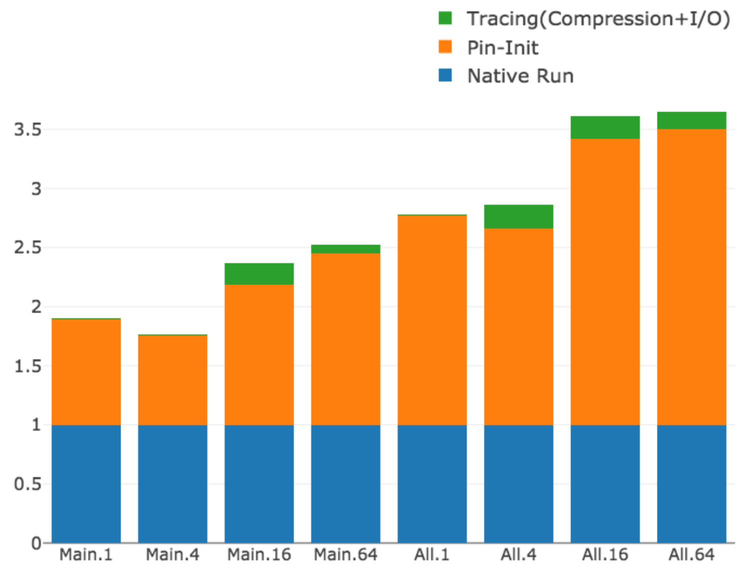


Figure 2.8: Tracing overhead breakdown - Input B

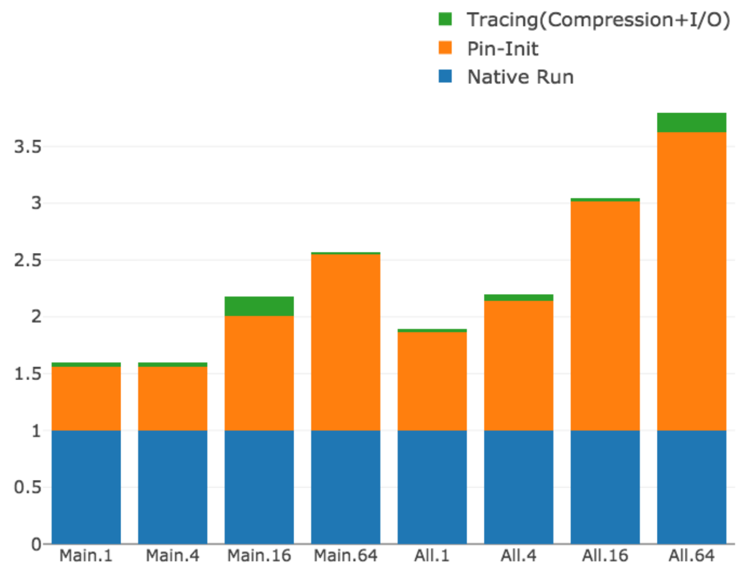


Figure 2.9: Tracing overhead breakdown - Input C

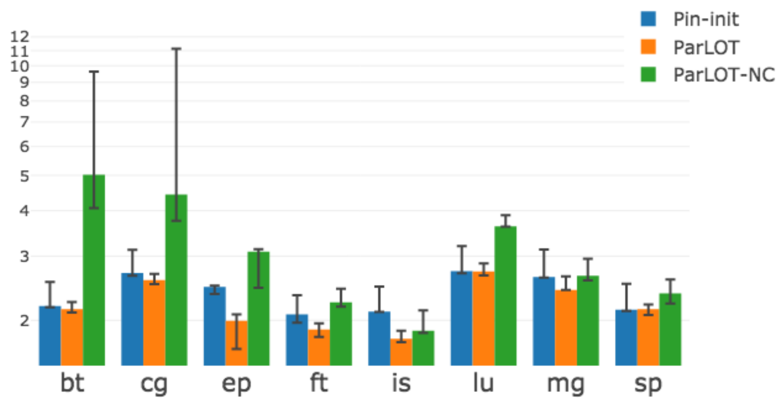


Figure 2.10: Variability of PARLOT(M) overhead on 16 nodes - Input B

kB/s of the system bandwidth, *leaving the rest of the available bandwidth to the application.* In comparison, CALLGRIND collects **less than 100 kB** of data but still adds more overhead compared to either PARLOT(A) or PARLOT(M). The average amount of trace data that can be collected by PARLOT(A) is **360x** (85x for PARLOT(M)) larger than that for CALLGRIND. In the best observed case, the compression ratio of PARLOT exceeds 21000. This is particularly impressive because it was achieved with relatively low overhead and incremental

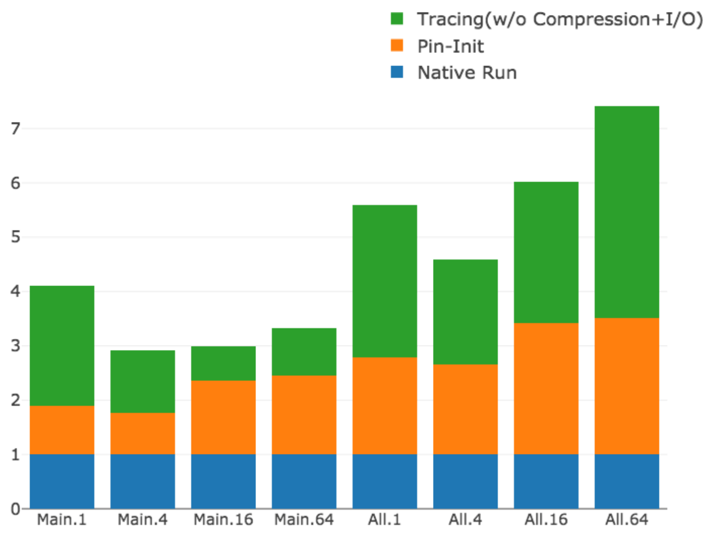


Figure 2.11: PARLOT-NC tracing overhead breakdown - Input B

on-the-fly compression. Generally, the compression ratios of PARLOT(M) are higher than those of PARLOT(A) because the variety of distinct function calls on the main image is smaller than when tracing all images, thus compression performs better on PARLOT(M). Also by looking at Fig. 2.4, Fig. 2.5, Fig. 2.6 and Fig. 2.7, we find EP to have the highest compression ratio of the NPB applications. At the same time, it has the minimum required bandwidth. The opposite is true for CG, which exhibits the lowest compression ratio and the highest required bandwidth. CG is a conjugate gradient method with irregular memory accesses and communications whereas EP is an embarrassingly parallel random number generator. CG's whole-program trace contains a larger number of distinct calls and more complex patterns than that of EP, thus resulting in a higher bandwidth and lower compression ratio.

PARLOT's compression mechanism works better on larger input sizes because larger inputs tend to result in longer streams of similar function calls (e.g., a call that is made for every processed element).

2.5.4 Overheads

Tables 2.4 and 2.5 present the average overhead added to each application for different versions of PARLOT. The last row of these two tables presents the geometric mean. This information captures how much each phase of PARLOT slows down the native execution.

Table 2.3: Compression ratio

Input	Tool	# Nodes	bt	cg	ep	ft	is	lu	mg	sp	GM
B	ParLoT(m)	1	3 035.9	94.4	12 456.2	12 173.5	9 718.4	167.7	99.1	878.3	1 255.2
		4	586.6	82.5	10 368.4	1 737.1	909.2	140.3	255.0	338.2	559.4
		16	346.7	113.3	8 563.9	1 077.4	1 200.6	179.0	387.6	123.0	496.8
		64	252.2	147.8	7 611.0	1 122.6	1 908.0	366.8	437.3	152.9	591.1
		AVG	1 055.4	109.5	9 749.9	4 027.6	3 434.0	213.5	294.7	373.1	**725.6**
	ParLoT(a)	1	514.5	137.4	3 335.8	1 226.7	543.2	314.6	260.9	303.9	500.2
		4	315.7	137.2	1 266.9	436.2	316.2	287.3	329.6	199.7	330.7
		16	226.9	181.6	1 246.7	1 026.5	927.1	299.3	469.3	171.5	430.4
		64	329.2	247.3	1 394.1	1 043.9	1 984.6	410.3	548.5	307.2	597.6
		AVG	346.6	175.9	1 810.9	933.3	942.8	327.9	402.1	245.6	**464.7**
C	ParLoT(m)	1	8 619.0	111.2	13 068.0	21 335.6	21 856.5	350.0	247.4	1 977.4	2 371.4
		4	1 910.6	110.5	12 418.7	6 520.3	2 256.6	112.8	268.0	472.7	928.2
		16	580.8	133.2	11 017.4	1 239.3	1 347.9	164.5	396.9	143.1	582.8
		64	322.8	131.9	9 155.0	1 065.1	1 896.3	223.7	465.7	168.9	585.7
		AVG	2 858.3	121.7	11 414.7	7 540.1	6 839.3	212.7	344.5	690.5	**1 117.0**
	ParLoT(a)	1	2 579.4	181.8	7 377.0	5 143.1	1 520.4	408.2	314.8	650.7	1 107.4
		4	448.6	161.3	3 194.6	1 062.9	527.3	274.7	319.4	237.4	477.4
		16	285.1	185.7	1 765.5	588.9	1 106.3	273.6	467.4	141.7	426.9
		64	290.0	214.7	1 512.9	1 237.3	2 038.7	329.0	496.2	270.8	565.8
		AVG	900.8	185.9	3 462.5	2 008.1	1 298.2	321.4	399.4	325.2	**644.4**

Table 2.4: Tracing overhead of versions of ParLoT(m)- Input B - *Pin*: Pin-Init, *P*: ParLoT, P_{nc}: ParLoT-NC

Input: B	Nodes :	1			4			16			64		
	Detail Tools:	*Pin*	*P*	P_{nc}	*Pin*	*P*	P_{nc}	*Pin*	*P*	P_{nc}	*Pin*	*P*	P_{nc}
Main	bt	1.5	1.5	5.6	1.7	1.7	5.0	2.1	2.1	5.0	1.8	2.1	3.5
	cg	1.7	1.8	2.3	1.8	1.8	2.6	2.7	2.5	4.4	2.3	2.1	4.6
	ep	2.9	2.6	20.4	1.9	1.8	5.3	2.4	1.9	3.0	2.6	2.3	2.6
	ft	1.8	2.1	6.1	1.7	1.7	2.7	2.0	1.8	2.2	2.1	1.9	2.1
	is	2.4	2.4	4.8	1.7	1.7	2.0	2.1	1.7	1.8	4.5	4.3	5.7
	lu	1.3	1.3	1.4	1.7	1.7	2.2	2.7	2.7	3.6	3.0	4.3	6.1
	mg	2.5	2.5	2.7	1.5	1.5	1.5	2.6	2.4	2.6	1.9	1.9	1.8
	sp	1.3	1.3	2.4	1.7	1.7	3.5	2.1	2.1	2.3	1.9	2.0	2.5
	GM	1.8	1.9	4.1	1.7	1.7	2.9	2.3	2.1	3.0	2.4	2.5	3.3

Table 2.5: Tracing overhead of versions of *P*a - Input B - *Pin*: Pin-Init, *P*: ParLoT, P_{nc}: ParLoT-NC

Input: B	Nodes :	1			4			16			64		
	Detail Tools:	*Pin*	*P*	P_{nc}	*Pin*	*P*	P_{nc}	*Pin*	*P*	P_{nc}	*Pin*	*P*	P_{nc}
All	bt	1.7	1.8	6.1	2.3	2.5	6.1	3.2	3.5	9.0	2.8	3.1	7.5
	cg	2.6	2.7	3.8	2.8	3.0	4.4	4.0	4.2	11.3	3.3	3.2	10.3
	ep	4.3	4.1	22.2	3.1	3.4	7.1	3.1	3.3	4.5	4.1	3.8	4.1
	ft	2.8	2.7	6.8	2.6	2.7	3.8	2.8	2.9	3.6	3.1	3.0	3.5
	is	4.4	4.2	7.0	2.8	2.9	3.4	2.9	2.8	3.2	5.3	5.4	8.8
	lu	1.7	1.7	2.3	2.5	2.7	4.8	3.9	4.3	10.4	4.4	4.6	23.4
	mg	4.8	4.7	5.3	2.5	2.7	3.0	4.3	4.4	5.2	2.7	3.1	3.2
	sp	1.7	1.7	3.0	2.4	2.6	5.0	3.2	3.6	5.6	2.7	3.3	11.6
	GM	2.7	2.7	5.5	2.6	2.8	4.5	3.4	3.6	6.0	3.5	3.6	7.4

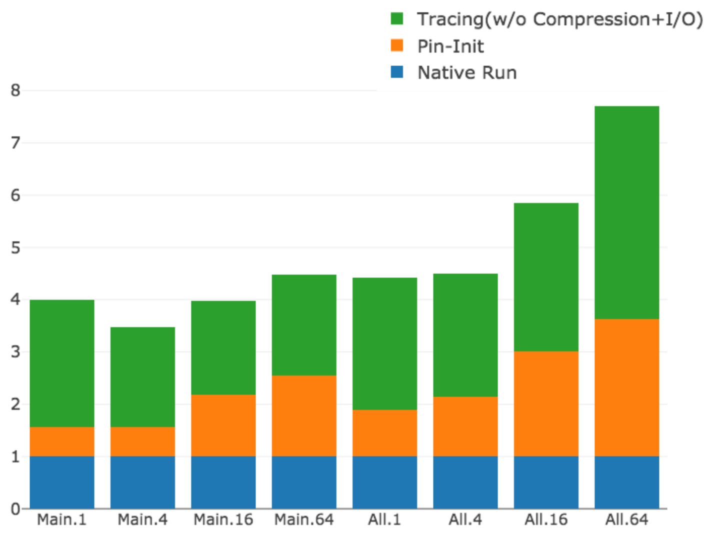


Figure 2.12: PARLOT-NC tracing overhead breakdown - Input C

In general, one expects the following inequality to hold: the overhead of PIN-INIT should be less than that of PARLOT, which should be less than that of PARLOT-NC. This is not always the case because of the non-deterministic runtimes of the applications. In fact, the variability across three runs of each experiment is shown in Fig. 2.10 where we present the minimum, maximum and median overheads. These overheads are for input size B and 16 nodes. This variability explains the seeming inconsistencies in Tables 2.4 and 2.5.

On average, PIN-INIT adds an overhead of 3.28 and PARLOT(A) adds an overhead of 3.42. This means that **almost 96% of PARLOT(A)'s overhead is due to PIN**. The results of PARLOT(M) and other inputs follow the same pattern as shown in Fig. 2.8 and 2.9. The overhead that PARLOT (excluding the overhead of PIN-INIT) *adds* to the applications is very small. If we were to switch to a different instrumentation tool that is not as general as PIN but more lightweight, the overhead would potentially reduce drastically.

2.5.5 Compression Impact

Fig. 2.11 and Fig. 2.12 show the overhead breakdown of PARLOT-NC, which illustrate the impact of compression. They also highlight the importance of incorporating compression directly in the tracing tool. On average, PARLOT-NC slows down the application execution almost **2x** more than PARLOT(A). The average overhead across Table 2.5 for

ParLoT(a) is **3.4**. The corresponding factor for ParLoT-NC is **6.6**. The numbers of ParLoT(m) and input C follow the same pattern. For example, ParLoT-NC slows down the application execution almost **1.66x** more than ParLoT(m).

Clearly, compression not only lowers the storage requirement but also the overhead. This is important as it shows that the extra computation to perform the compression is more than amortized by the reduction in the amount of data that need to be written out.

This result validates our approach and highlights that incremental, on-the-fly compression is likely essential to make whole-program tracing possible at low overhead.

2.6 Discussion and Conclusion

In this paper, we present ParLoT, a portable low overhead dynamic binary instrumentation-based whole-program tracing approach that can support a variety of dynamic program analyses, including debugging. Key properties of ParLoT include its on-the-fly trace collection and compression that reduces timing jitter, I/O bandwidth, and storage requirements to such a degree that whole-program call/return traces can be collected efficiently even at scale.

We evaluate various versions of ParLoT created by disabling/enabling compression, not collecting any traces, etc. In order to provide an intuitive comparison against a well known tool, we also compare ParLoT to Callgrind. Our metrics include the tracing overhead, required bandwidth, achieved compression ratio, initialization overhead, and the overall impact of compression. Detailed evaluations on the NAS parallel benchmarks running on up to 1024 cores establish the merit of our tool and our design decisions. ParLoT can collect more than 36 MB worth of data per core per second while only needing 56 kB/s of bandwidth and slowing down the application by 2.7x on average. These results are highly promising in terms of supporting whole program tracing and debugging, in particular when considering that most of the overhead is due to the DBI tool and not ParLoT.

The traces collected by ParLoT cut through the entire stack of heterogeneous (MPI, OpenMP, PThreads) calls. This permits a designer to project these traces onto specific APIs of interest during program analysis, visualization, and debugging.

A number of improvements to ParLoT remain to be made. These include allowing users to selectively trace at specific interfaces: doing so can further increase compression efficiency by reducing the variety of function calls to be handled by the compressor. We also discuss the need to bring down initialization overheads, i.e., by switching to a less general-purpose DBI tool.

Acknowledgment

This research was supported by the NSF. We thank our colleague Dr. Hari Sundar from the University of Utah who provided insight and expertise that greatly assisted the research. We also thank the Texas Advanced Computing Center (TACC) and the San Diego Supercomputer Center (SDSC) for the infrastructure they provided for running our experiments.

CHAPTER 3

WHOLE-PROGRAM TRACE ANALYSIS

This chapter is based on the work published at the 2019 IEEE International Conference on Cluster Computing [98][1]. We present a tool called **DiffTrace** that approaches debugging via *whole program* tracing and diffing of typical and erroneous traces. After collecting these traces, a user-configurable front-end filters out irrelevant function calls and then summarizes loops in the retained function calls based on state-of-the-art loop extraction algorithms. Information about these loops is inserted into concept lattices, which we use to compute salient dissimilarities to narrow down bugs. DiffTrace is a clean start that addresses debugging features missing in existing approaches. Our experiments on an MPI/OpenMP program called ILCS and initial measurements on LULESH, a DOE miniapp, demonstrate the advantages of the proposed debugging approach.

3.1 Introduction

Debugging high-performance computing code remains a challenge at all levels of scale. Conventional HPC debuggers [4,35] excel at many tasks such as examining the execution state of a complex simulation in detail and allowing the developer to re-execute the program close to the point of failure. However, they do not provide a good understanding of why a program version that worked earlier failed upon upgrade or feature addition. Innovative solutions are needed to highlight the salient differences between two executions in a manner that makes debugging easier as well as more systematic. A recent study conducted under the auspices of the DOE [34] provides a comprehensive survey of existing debugging tools. It classifies them under four software organizations (serial, multithreaded, multi-process, and hybrid), six method types (formal methods, static analysis, dynamic

[1]©2019 IEEE. Adapted, with permission, from S. Taheri, I. Briggs, M. Burtscher and G. Gopalakrishnan, "DiffTrace: Efficient Whole-Program Trace Analysis and Diffing for Debugging", 2019 IEEE International Conference on Cluster Computing (CLUSTER), 2019, pp. 1-12, doi: 10.1109/CLUSTER.2019.8891027

analysis, nondeterminism control, anomaly detection, and parallel debugging), and lists a total of 30 specific tools. Despite this abundance of activity and tools, many significant problems remain to be solved before debugging *can be approached by the HPC community as a collaborative activity* so that HPC developers can extend a common framework.

Almost all debugging approaches seek to find outliers ("unexpected executions") amongst thousands of running processes and threads. The approach taken by most existing tools is to look for symptoms in a specific bug-class that they cover. Unfortunately, this approach calls for a programmer having a good guess of what the underlying problem might be, and to then pick the right set of tools to deploy. If the guess is wrong, the programmer has no choice but to refine their guess and look for bugs in another class, re-executing the application and hoping for better luck with another tool. This iterative loop of re-execution followed by applying a best-guess tool for the suspected bug class can potentially consume large amounts of execution cycles and wastes an expert developer's time. More glaring is the fact that these tools must recreate the execution traces yet again: they do not have means to hand off these traces to another tool or cooperate in symbiotic ways.

We cannot collect all relevant pieces of information necessary to detect all possible bug classes such as resource leaks, deadlocks, and data races. Each such bug requires its attributes to be kept. Also, debugging is not fully automatable (it is an undecidable problem in general) and must involve human thinking: at least to reconcile what is observed against the deeper application-level semantics. However, (1) we believe that it is still possible to collect one standard set of data and use it to make an initial triage in such a way that it can guide a later, deeper debugging phase to locate which of the finer bug gradations (e.g., resource leaks or races) brought the application down. Also, (2) we believe that it is possible to engage the human *with respect to understanding structured presentations of information*.

Our DiffTrace framework addresses both issues. The common set of data it uses is a *whole program function call trace* collected per process/thread. DiffTrace relies on novel ways to diff a normal trace and a fault-laden trace to guide the debugging engineer closer to the bug. While our work has not (yet) addressed situations in which millions of threads and thousands of processes run for days before they produce an error, we strongly believe that we can get there once we understand the pros and cons of our initial implementation of the DiffTrace tool, which is described in this paper. The second issue is handled in DiffTrace by offering a novel collection of modalities for understanding program execution diffs. We now elaborate on these points by addressing the following three problems.

3.1.1 Problem 1 – Collecting Whole-Program Heterogeneous Function-Call Traces Efficiently

Not only must we have the ability to record function calls and returns at one API such as MPI, increasingly we must collect calls/returns at multiple interfaces (e.g., OpenMP, PThreads, and even inner levels such as TCP). The growing use of heterogeneous parallelization necessitates that we understand MPI and OpenMP activities (for example) to locate cross-API bugs that are often missed by other tools. Sometimes, these APIs contain the actual error (as opposed to the user code), and it would be attractive to have this debugging ability.

Solution to Problem 1: In DiffTrace, we choose Pin-based whole program binary tracing, with tracing filters that allow the designer to collect a suitable mixture of API calls/returns. We realize this facility using ParLOT, a tool designed by us and published earlier [99]. In our research, we have thus far demonstrated the advantage of ParLOT with respect to collecting both MPI and OpenMP traces from a *single run of a hybrid MPI/OpenMP program*. We demonstrate that, from this single type of trace, it is possible to pick out MPI-level bugs and/or OpenMP-level bugs. While whole-program tracing may sound extremely computation and storage intensive, ParLOT employs lightweight on-the-fly compression techniques to keep these overheads low. It achieves compression ratios exceeding 21,000 [99], thus making this approach practical, demanding only a few kilobytes per second per core of bandwidth.

3.1.2 Problem 2 – Need to Generalize Techniques for Outlier Detection

Given that outlier detection is central to debugging, it is essential to use efficient representations of the traces to be able to systematically compute *distances* between them without involving human reasoning. The representation must also be versatile enough to be able to "diff" the traces with respect to *an extensible number of vantage points*. These vantage points could be diffing traces concerning process-level activities, thread-level activities, a combination thereof, or even finite sequences of process/thread calls (say, to locate *changes* in caller/callee relationships).

Solution to Problem 2: DiffTrace employs *concept lattices* to amalgamate the collected traces. Concept lattices have previously been employed in HPC to perform structural clustering of process behaviors [107] to present performance data more meaningfully to users. The authors of that paper use the notion of *Jaccard distances* to cluster performance results that are closely related to process structures (determined based on caller/callee relationships). In DiffTrace, we employ incremental algorithms for building and maintaining concept lattices from the ParLOT-collected traces. In addition to Jaccard distances, in our work, we also perform hierarchical clustering of traces and provide a tunable threshold for outlier detection. We believe that these uses of concept lattices and refinement approaches for outlier detection are new in HPC debugging.

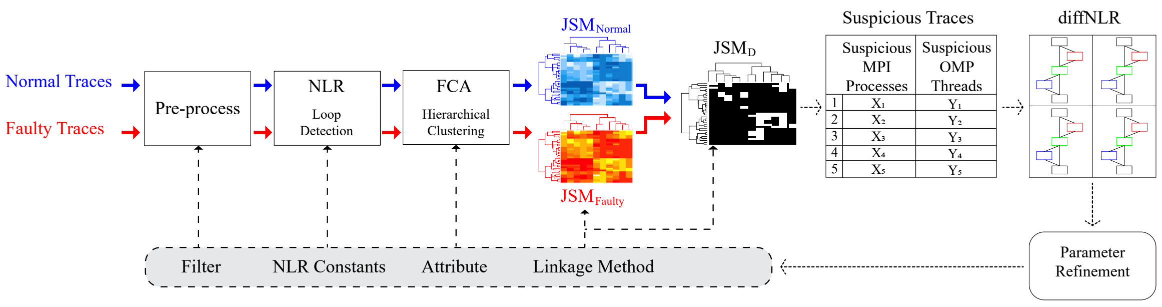
3.1.3 Problem 3 – Loop Summarization

Most programs spend most of their time in loops. Therefore, it is important to employ state-of-the-art algorithms for loop extraction from execution traces. It is also important to be able to diff two executions with respect to changes in their looping behaviors. In our experience, presenting such changes using good visual metaphors tends to highlight many bug types immediately.

Solution to Problem 3: DiffTrace utilizes the rigorous notion of Nested Loop Representations (NLRs) for summarizing traces and representing loops. Each repetitive loop structure is given an identifier, and nested loops are expressed as repetitions of this identifier exponentiated (as with regular expressions). This approach to summarizing loops can help manifest bugs where the program does not hang or crash but nevertheless runs differently in a manner that informs the developer engaged in debugging.

Organization: §3.2 illustrates the contributions of this paper on a simple example. §3.3 presents the algorithms underlying DiffTrace in more detail. §3.4 summaries the experimental methodology before showing a medium-sized case study involving MPI and OpenMP. §3.5 shows initial measurements and examples on LULESH [58], a DOE common mini app. §3.6 summarizes selected related works. §3.7 concludes the paper with a discussion.

Figure 3.1: DiffTrace Overview



3.2 DiffTrace Overview

3.2.1 High-level Overview

DiffTrace employs ParLOT's whole-program function-call and return trace-collection mechanism, where ParLOT captures traces via Pin [70] and incrementally compresses them using a new compression scheme [99]. ParLOT can capture functions at two levels: the *main image* (which does not include library code) and *all images* (including all library code). As the application runs, ParLOT generates per-thread trace files that contain the compressed sequence of the IDs of the executed functions. The compression mechanism is light-weight yet effective, thus reducing not only the required bandwidth and storage but also the runtime relative to not compressing the traces. As a result, ParLOT can capture whole-program traces at low overhead while leaving most of the disk bandwidth to the application. Using whole-program traces substantially reduces the number of overall debug iterations because it allows us to repeatedly analyze the traces offline with different filters.

Figure 3.1 provides an overview of the DiffTrace toolchain in terms of the blue flows (fault-free) and red flows (faulty). In a broad sense, code-level faults in HPC applications (e.g., the use of wrong subscripts) turn into observable code-level misbehaviors (e.g., an unexpected number of loop iterations), many of which turn into application-level issues. In our study of DiffTrace, we evaluate success merely in terms of the efficacy of observing these misbehaviors in response to injected code-level faults (we rely on a rudimentary fault injection framework complemented by manual fault injection).

The preprocessing stage removes calls/returns at the ignored APIs. The nested loop recognition (NLR) mechanism then extracts loops from traces. The resulting information not only serves as a lossless abstraction to ease the rest of the trace analysis but also serves as a *per-thread measure of progress*. The FCA (Formal Concept Analysis) stage conducts a systematic way to arrange objects (in our case threads) and attributes (we support a rich collection of attributes including the set of function calls a thread makes, the set of *pairs* of function calls made—this reflects calling context—etc.). Weber et al.'s work [28, 107] employs FCA exactly in this manner (including the use of pairs of calls), however, for grouping performance information. Our new contribution is showing that FCA can play a central role in debugging HPC applications.

While faults induce asymmetries ("aberrations") in program behaviors, one cannot locate faults merely by locating the asymmetries in an overall collection of process traces. The reason is that even in a collection of MPI processes or threads within these processes, some processes/threads may serve as a master while others serve as workers. Thus, we must have a base level of similarities computed even for normal behaviors and then compute how *this similarity relation changes* when faults are introduced. This is highlighted by the blue and red rectangular patches in Figure 3.1 that, respectively, iconify the *Jaccard similarity matrices* computed for the normal behavior (above) and the erroneous behavior (below). This is shown as the "diff Jaccard similarity matrix" in greyscale at the juncture of JSM_{normal} and JSM_{faulty}.

After the JSM_D matrix is computed, we invoke a hierarchical clustering algorithm that computes the "B-score" and helps rank suspicious traces/processes. The diffNLR representation is then extracted. Intuitively, this is a diff of the loop structures of the normal and abnormal threads/processes. This diagram shows (as with git diff and text diff) a *main stem* comprised of green rectangles ("common looping structure") and red/blue *diff rectangles* showing how the loop structures of the normal and erroneous threads differ with respect to the main stem. We show that this presentation often helps the debugging engineer locate the faults.

Last but not least, we strongly believe that a framework such as DiffTrace can serve as an important HPC community resource. Each debugging tool designer who uses DiffTrace can extend it by incorporating new attributes and clustering methods, but otherwise retain the overall tool structure. Such a "playground" for developing and exploring new methods for debugging does not exist in HPC. There is also the intriguing possibility that many of the 30-odd tools mentioned in §4.1 *can be made to focus on the problems highlighted by diffNLR*, thus gaining efficiency (this will be part of our future work).

In this paper, we describe DiffTrace as a *relative debugging* [86] tool, in that bugs are caught with respect to JSM$_D$ which is a *change* from the previous code version found working. However, many types of faults may be apparent just by analyzing JSM$_{faulty}$: for instance, processes whose execution got truncated will look highly dissimilar to those that terminated normally. In those use cases of DiffTrace, the B-score based ranking can then be made on JSM$_{faulty}$ directly.

3.2.2 Example Walk-through

We now employ Figure 3.2—a textbook MPI odd/even sorting example—to illustrate DiffTrace. Odd/even sorting is a parallel variant of bubble sort and operates in two alternating phases: in the *even phase*, the even processes exchange (conditionally swap) values with their right neighbors, and in the *odd phase*, the odd processes exchange values with their right neighbors.

A waiting trap in this example is this: the user may have swapped the `Recv; Send` order in the `else` part, creating head-to-head `‘‘Send || Send’’` deadlock under low-buffering (MPI EAGER limit). We will now show how DiffTrace helps pick out this root-cause.

Figure 3.2: Simplified MPI implementation of Odd/Even Sort

	Main Function	oddEvenSort()
1	`int main(){`	`oddEvenSort(rank, cp){`
2	`int rank,cp;`	`...`
3	`MPI_Init()`	`for (int i=0; i < cp; i++)`
4	`MPI_Comm_rank(..., &rank);`	`{`
5	`MPI_Comm_size(..., &cp);`	`int ptr = findPtr(i, rank);`
6	`// initialize data to sort`	`...`
7	`int *data[data_size];`	`if (rank % 2 == 0) {`
8	`...`	`MPI_Send(..., ptr, ...);`
9	`oddEvenSort(rank, cp);`	`MPI_Recv(..., ptr, ...);`
10	`...`	`} else {`
11	`MPI_Finalize();`	`MPI_Recv(..., ptr, ...);`
12	`}`	`MPI_Send(..., ptr, ...);`
13		`}`
14		`...`
15		`}`
16		`}`

3.2.3 Pre-processing

Using ParLOT's decoder, each trace is first decompressed. Next, the desired functions are extracted based on predefined (Table 3.1) or custom regular expressions (i.e., *filters*) and kept for later phases. Table 3.2 shows the pre-processed traces (T_i) of odd/even sort with four processes. T_i is the trace that stores the function calls of process i.

3.2.4 Nested Loop Representation

Virtually all dynamic statements are found within loops. Function calls within a loop body yield *repetitive patterns* in ParLOT traces. Inspired by ideas for the detection of repetitive patterns in strings [77] and other data structures [59], we have adapted the Nested Loop Recognition (NLR) algorithm by Ketterlin et al. [60] to detect repetitive patterns in ParLOT traces (cf. Section 3.3.1). Detecting such patterns can be used to measure the progress of each thread, revealing unfinished or broken loops that may be the consequence of a fault.

For example, the loop in line 3 of `oddEvenSort()` (Figure 3.2) iterates four times when run with four processes. Thus each T_i contains four occurrences of either [`MPI_Send-MPI_Recv`] (even i) or [`MPI_Recv-MPI_Send`] (odd i). By keeping only MPI functions and converting each T_i into its equivalent NLR, Table 3.2 can be reduced to Table 3.3 where **L0** and **L1** represent the *loop body* [`MPI_Send-MPI_Recv`] and [`MPI_Recv-MPI_Send`], respectively. The integer after the ˆ symbol in NLR represents the *loop iteration count*. Note that, since the first and last processes only have one-way communication with their neighbors, T_0 and T_3

Table 3.1: Pre-defined Filters

Category	Sub-Category	Description
Primary	Returns	Filter out all returns
	PLT	Filter out the ".plt" function calls for external functions/procedures that their address needs to be resolved dynamically from Procedure Linkage Table (PLT)
MPI	MPI All	Only keep functions that start with "MPI_"
	MPI Collectives	Only keep MPI collective calls (MPI_Barrier, MPI_Allreduce, etc)
	MPI Send/Recv	Only keep MPI_Send, MPI_Isend, MPI_Recv, MPI_Irecv and MPI_Wait
	MPI Internal Library	Keep all inner MPI library calls
OMP	OMP All	Only keep OMP calls (starting with GOMP_)
	OMP Critical	Only keep OMP_CRITICAL_START and OMP_CRITICAL_END
	OMP Mutex	Only keep OMP_Mutex calls
System	Memory	Keep any memory related functions (memcpy, memchk, alloc, malloc, etc)
	Network	Keep any network related functions (network, tcp, sched, etc)
	Poll	Keep any poll related functions (poll, yield, sched, etc)
	String	Keep any string related functions (strlen, strcpy, etc)
Advanced	Custom	Any regular expression can be captured
	Everything	Does not filter anything

perform only half as many iterations.

3.2.5 Hierarchical Clustering via FCA

Processes in HPC applications are known to fall into predictable equivalence classes. The widely used and highly successful STAT tool [3] owes most of its success for being able to efficiently collect stack traces (nested sequences of function calls), organize them as prefix-trees, and equivalence the processes into teams that evolve in different ways. Coalesced stack trace graphs (CSTG, [18]) have proven effective in locating bugs within Uintah [17] and perform stat-like equivalence class formation, albeit with the added detail of maintaining calling contexts. Inspired by these ideas, FCA-based clustering provides the next logical level of refinement in the sense that (1) we can pick any of the multiple attributes one can mine from traces (e.g., pairs of function calls, memory regions accessed by processes, locks held by threads, etc.), and (2) form this equivalencing relation quite naturally by computing the Jaccard distance between processes/threads. In general, such a classification is powerful enough to distinguish structurally different threads from one another (e.g., MPI processes from OpenMP threads in hybrid MPI+OpenMP applications) and reduce the search space for bug location to a few representative classes of traces that are distinctly dissimilar.[2]

A *formal context* is a triple $K = (G, M, I)$ where G is a set of **objects**, M is a set of **attributes**, and $I \subseteq G \times M$ is an incidence relation that expresses *which objects have which attributes*. Table 3.4 shows the formal context of the preprocessed odd/even-sort traces. We can employ as attributes either the function calls themselves or the detected loop bodies (each detected loop is assigned a unique ID, and one can diff with respect to these IDs). The context shows that all traces include the functions MPI_Init(), MPI_Comm_size(), MPI_Comm_rank() and MPI_Finalize(). The even traces contain the loop *L0* and the odd traces the loop *L1*.

Figure 3.3 shows the concept lattice derived from the formal context in Table 3.4 and is interpreted as follows:

- The top node indicates that all traces share MPI_Init(), MPI_Comm_size(), MPI_Comm_rank()

[2]As emphasized earlier, we perform "sky subtraction" as in astronomy to locate comets; in our case, we diff the diffs, which is captured in JSM$_D$.

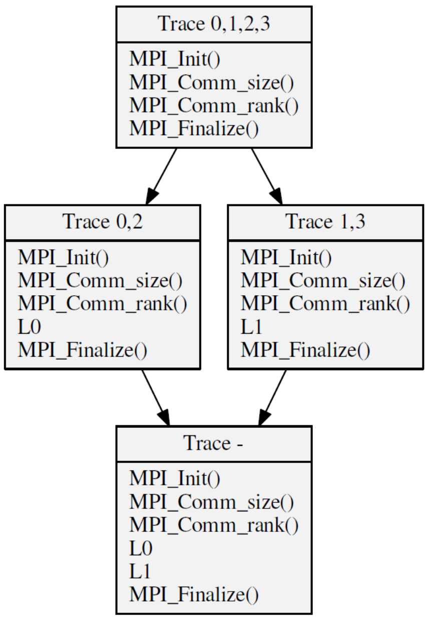
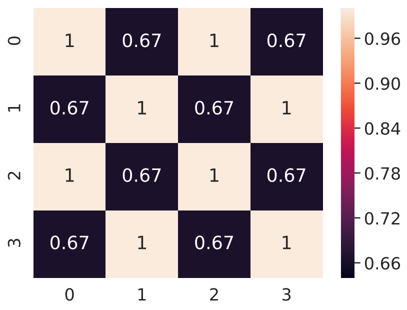


Figure 3.3: Sample Concept Lattice from Object-Attribute Context in Table 3.4



	0	1	2	3
0	1	0.67	1	0.67
1	0.67	1	0.67	1
2	1	0.67	1	0.67
3	0.67	1	0.67	1

Figure 3.4: Pairwise Jaccard Similarity Matrix (JSM) of MPI Processes in Sample Code

and MPI_Finalize().

- The bottom node signifies that none of the traces share all attributes.
- The middle nodes show that T_0 and T_2 are different from T_1 and T_3.

The complete pairwise Jaccard Similarity Matrix (JSM) can easily be computed from concept lattices. For large-scale executions with thousands of threads, it is imperative to employ incremental algorithms to construct concept lattices (detailed in Section 3.3.2). Figure 3.4 shows the heatmap of the JSM obtained from the concept lattice in Figure 3.3. DiffTrace uses the JSM to form equivalence classes of traces by hierarchical clustering. Next, we show how the differences between two hierarchical clusterings from two executions (faulty vs. normal) reveal which traces have been affected the most by the fault.

3.2.6 Detecting Suspicious Traces via JSM$_D$

JSM$_{normal}$[i][j] (JSM$_{faulty}$[i][j]) shows the Jaccard similarity score of T_i and T_j from the normal trace (T_i' and T_j'). As explained earlier, we compute JSM$_D$ to detect outlier executions, where JSM$_D$ = $|\text{JSM}_{faulty} - \text{JSM}_{normal}|$.

We sort the suggestion table based on the *B-score* similarity metric of two hierarchical clusterings [25] (cf. Section 3.3.3). A single iteration through the DiffTrace loop (with a single set of parameters shown as a dashed box in Figure 3.1) may still not detect the

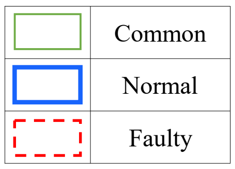


Figure 3.5: Legend

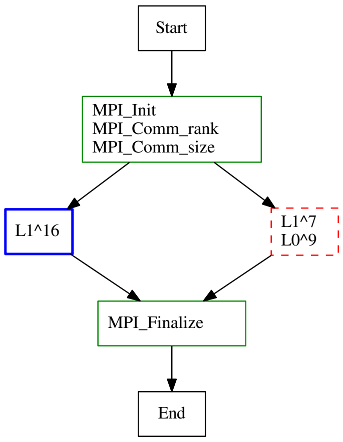


Figure 3.6: swapBug

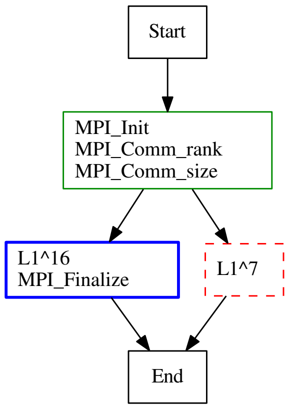


Figure 3.7: dlBug

root-cause of a bug. The user can then (1) alter the linkage method employed in computing the hierarchical clustering (reorder the dendrograms built to achieve the clustering), (2) alter the FCA attributes, (3) adjust the NLR constants (loops are extracted with realistic complexity by observing repetitive patterns inside a preallocated buffer), and/or (4) the front-end filters. This is shown in the iterative loop in Figure 3.1.

3.2.7 Evaluation

To evaluate the effectiveness of DiffJSM, we planted two artificial bugs (*swapBug* and *dlBug*) in the code from Figure 3.2 and ran it with 16 processes. *swapBug* swaps the order of MPI_Send and MPI_Recv in rank 5 after the seventh iteration of the loop in line 3 of `oddEvenSort`, simulating a potential deadlock. *dlBug* simulates an actual deadlock in the same location (rank 5 after the seventh iteration). Upon collection of ParLOT traces from the execution of the buggy code versions, DiffTrace first decompresses them and filters out all non-MPI functions. Then two major loops are detected, **L0** and **L1** (identical to the ones in Table 3.3), that are supposed to loop 16 times in the even and odd traces, respectively (except for the first and last traces, which loop just eight times).

After constructing concept lattices and their corresponding JSMs, trace 5 appears as the trace that got affected the most by the bugs because row 5 (showing the similarity score of T_5 relative to all other traces) ($\text{JSM}_{normal}[5][i]$ for $i \in [0,16)$) changed the most after the bug was introduced. The differences between the suggested suspicious trace (T'_s) and its corresponding normal trace (T_s) is visualized by *diffNLR*.

3.2.7.1 diffNLR

To highlight the differences in an easy-to-understand manner, DiffTrace visually separates the common and different blocks of a pair of pre-processed traces via *diffNLR*, a graphical visualization of the `diff` algorithm [76].

`diff` takes two sequences S_A and S_B and computes the minimal *edit* to convert S_A to S_B. This algorithm is used in the GNU `diff` utility to compare two text files and in git for efficiently keeping track of file changes. Since ParLOT preserves the order of function calls, each trace T_i is totally ordered. Thus *diff* can expose the differences of a pair of Ts. *diffNLR* aligns common and different blocks of a pair of sequences (e.g., traces) horizontally and vertically, making it easier for the analyst to see the differences at a glance. For simplicity, our implementation of *gdiff* only takes one argument x that denotes *the suspicious trace*.

diffNLR$(x) \equiv$ diffNLR(T_x, T'_x) where T_x is the trace of thread/process x of a normal execution and T'_x is the corresponding trace of the faulty execution.

Figure 3.6 shows the diffNLR(5) of *swapBug* where T_5 iterates over the loop [MPI_Recv - MPI_Send] 16 times (L1^16) after the MPI initialization while the order swap is well reflected in T'_5 (L1^7 - L0^9). Both processes seem to terminate fine by executing MPI_Finalize(). However, diffNLR(5) of *dlBug* (Figure 3.7) shows that, while T_5 executed MPI_Finalize, T'_5 got stuck after executing L1 seven times and never reached MPI_Finalize.

This example illustrates how our approach can locate the part of each execution that was impacted by a fault. Having an understanding of *how the application should behave normally* can reduce the number of iterations by picking the right set of parameters sooner.

3.3 Algorithms Underlying DiffTrace

3.3.1 Nested Loop Recognition (NLR)

We build NLRs based on the work by Ketterlin and Clauss [60], who use this algorithm for trace compression, and the work by Kobayashi and MacDougall [62], who propose a similar bottom-up strategy to build loop nests from traces, replacing each recognized loop with a new symbol. We adapt these algorithms to function-call traces wherein we record identical loops at different locations by introducing a single new (made-up) function ID that represents the entire loop. This process is restarted once the whole trace has been analyzed for depth-2 loops and so on until a function-ID replacement is performed. DiffTrace-NLR works by incrementally pushing trace entries (function IDs) onto a stack of *elements* (i.e., function IDs representing detected loop structures). Whenever an element is pushed onto the stack S, the upper elements of the stack are recursively examined for potential loop detection or loop extensions (Procedure 1).

```
Reduce($S$):
    **for** $i$ : 1 ... $3K$ **do**
        $b = i/3$
        **if** *Top 3 b-long elements of S are isomorphic* **then**
            pop $i$ elements from $S$
            $LB = S[b:1]$, $LC = 3$
            $LS = (LB, LC)$
            push $LS$ to $S$
            add $LB$ to the Loop Table
            Reduce($S$)
        **end**
        **if** $S[i]$ *is a loop (LS) and* $S[i-1:1]$ *isomorphic to its loop bodyLB* **then**
            $LC = LC + 1$
            pop $i - 1$ elements from $S$
            Reduce($S$)
        **end**
    **end**
```

Procedure 1: `Reduce` procedure adapted from the NLR algorithm

We store all distinct loop bodies (LBs) in a hash-table, assigning each a unique ID, which can be applied as a heuristic to detect loops not only in the current trace but also in other traces of the same execution. The maximum length of the subsequences to examine is decided by a fixed K. The complexity of the NLR algorithm is $\Theta(K^2N)$ where N is the size of the input. While loop detection has been researched in other contexts, its use to support debugging is believed to be novel.

3.3.2 Concept Lattice Construction

The efficiency of algorithms for concept lattice construction depends on the sparseness of the formal context [63]. Ganter's *Next Closure* algorithm [28] constructs the lattice from a *batch* of contexts and requires the whole context to be present in main memory and is, therefore, inefficient for long HPC traces.

We have implemented Godin's *incremental* algorithm [32] to extract attributes (Table 3.5) from each trace (object) and inject them into an initially empty lattice. Notice that our representation already includes compression of the attributes as (1) either the observed frequency is recorded, (2) the log10 of the frequency is recorded, or (3) "no frequency" (presence/absence) of a function call is recorded. *These are versatile knobs to adjust for bug-location and similarity calculation*.

Every time a new object with its set of attributes is added to the lattice, an *update* procedure minimally modifies/adds/deletes edges and nodes of the lattice. The extracted attributes are in the form $\{attr{:}freq\}$. *attr* is either a single entry of the trace NLR or a consecutive pair of entries. *freq* is a parameter to adjust the impact of the frequency of each *attr* in the concept lattice. The complexity of Godin's algorithm is $O(2^{2K}|G|)$, where K is an upper bound for the number of attributes (e.g., distinct function calls in the whole execution) and $|G|$ is the number of objects (e.g., the number of traces).

3.3.3 Hierarchical Clustering, Construction, and Comparison

DiffJSMs provide pair-wise dissimilarity measurements that can be used to combine traces (forming initial clusters). To obtain outliers (suspicious traces), we form dendrograms for which a *linkage* function is required to measure the distance between sets of traces. We currently employ SciPy (version 1.3.0. [55]) for these tasks. SciPy provides a wide range of linkage functions such as single, complete, average, weighted, centroid, median, and ward.

3.3.3.1 Ranking Table

As shown in Figure 3.1, each component of DiffTrace has some tunable parameters and constants, and the suggested suspicious traces are a function of them. Thus, a metric is needed to serve as the sorting key of the suspicious traces. Each parameter combination, in essence, creates a different DiffJSM, giving us "the distance between two hierarchical clusterings". Fowlkes et al. [25] proposed a method for comparing two hierarchical clusterings by computing their *B-score*. While we have not evaluated the full relevance of this idea, our initial experiments show that sorting suspicious traces based on the B-score of DiffJSMs is effective and brings interesting outliers to attention.

3.4 Case Study: ILCS

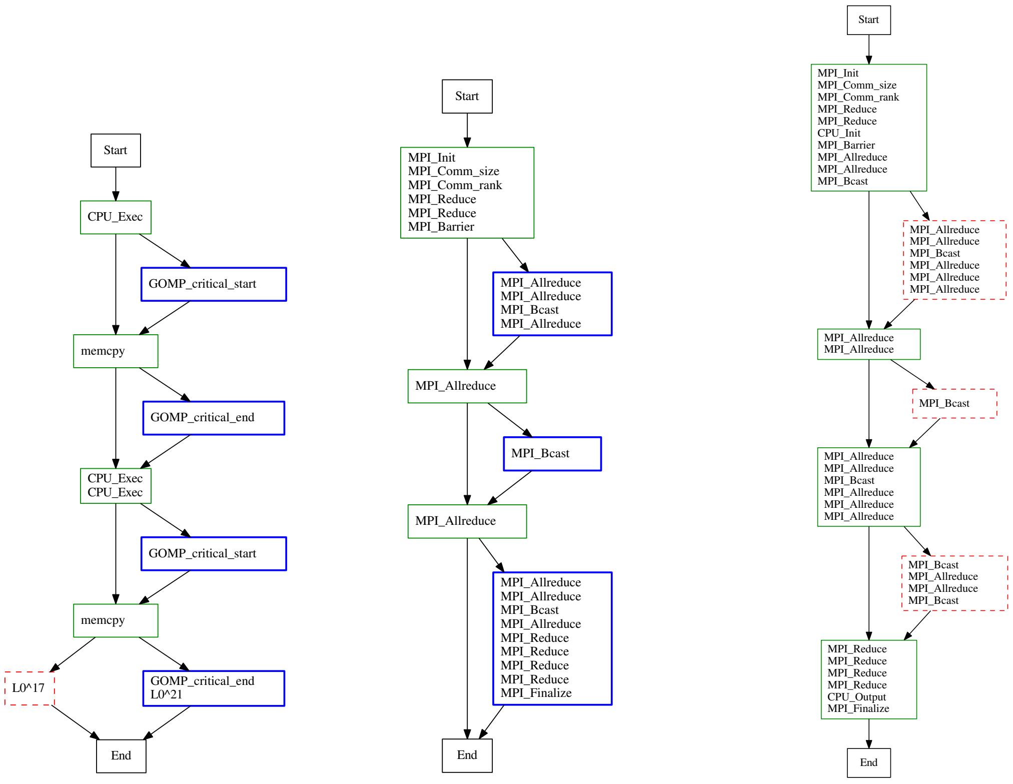
ILCS is a scalable framework for running iterative local searches on HPC platforms [12]. Providing serial CPU and/or single-GPU code, ILCS executes this code in parallel between compute nodes (MPI) and within them (OpenMP and CUDA).

To evaluate DiffTrace, we manually injected MPI-level and OMP-level bugs into the Traveling Salesman Problem (TSP) running on ILCS (Listing 3.1). The injected bugs simulate real HPC bugs such as deadlocks. Moreover, we inserted "hidden" faults that do not crash the program such as violations of critical sections and semantic bugs. The goal was to see how effectively DiffTrace can analyze the resulting traces and how close it can get to the root cause of the fault. .

```
main(argc, argv) {
 ... // initialization
 MPI_Init();
 MPI_Comm_size();
 MPI_Comm_rank(my_rank);
 ... // Obtain number of local CPUs and GPUs
 MPI_Reduce(lCPUs, gCPUs, MPI_SUM); // Total # of CPUs
 MPI_Reduce(lGPUs, gGPUs, MPI_SUM); // Total # of GPUs
 champSize = CPU_Init();
 ... // Memory allocation for storing local and global champions w.r.t. champSize
 MPI_Barrier();
 #pragma omp parallel num_threads(lCPUs+1)
 {rank = omp_get_thread_num();
  if (rank != 0) { // worker threads
   while (cont) {
    ... // calculate seed
    local_result = CPU_Exec();
    if (local_result < champ[rank]) { // update local champion
     #pragma omp critical
     memcpy(champ[rank], local_result);}}
  } else { //master thread
   do {
    ...
    MPI_AllReduce(); //broadcast the global champion
    ...
    MPI_AllReduce(); //broadcast the global champion P_id
    ...
    if (my_rank == global_champion_P_id) {
     #pragma omp critical
     memcpy(bcast_buffer, champ[rank]);
    }
    MPI_Bcast(bcast_buffer); // broadcast the local champion to all nodes
   } while (no_change_threshold);
   cont = 0; // signal worker threads to terminate
  }}
 if (my_rank == 0) {CPU_Output(champ);}
 MPI_Finalize();}

/* User code for TSP problem */
CPU_Init() {/* Read coordinates, calculate distances, initialize champion structure,
    return structure size */}
CPU_Exec() {/* Find local champions (TSP tours) */}
CPU_Output() {/* Output champion */}
```

Listing 3.1: ILCS Overview



(a) diffNLR(6.4) (b) diffNLR(4) (c) diffNLR(5)

Figure 3.8: Three diffNLR outputs

We collected ParLOT (main image) traces from the execution of ILCS-TSP with 8 MPI processes and 4 OpenMP threads per process on the XSEDE-PSC Bridges supercomputer whose compute nodes have 128 GB of main memory and contain 2 Intel Haswell (E5-2695 v3) CPUs with 14 cores each running at 2.3 - 3.3 GHz. Note that we did not provide any GPU code to ILCS.

The collected traces (faulty and normal) are fed to DiffTrace. We enabled the MPI, OpenMP, and custom (ILCS-TSP user code) filters and set the NLR constant K to 10 for all experiments. The current version of DiffTrace is implemented and built using C++ GCC 5.5.0, Pin 3.8, Python 2.7, and Scipy 1.3.0.

We present the results in the form of ranking tables that show which traces (processes and threads) DiffTrace considers "suspicious". Since DiffTrace output is highly dependent to "parameters", each row in ranking tables starts with parameters whom the suspicious traces are the result of.

The linkage method that converts JSMs to flat clustering is "ward" for all of the top reported suspicious traces that we removed from tables for better readability. Ward linkage function in SciPy uses *Ward variance minimization algorithm* to calculate the distance between newly formed clusters [55]. Furthermore, we show diffNLRs for selected traces.

3.4.1 ILCS-TSP Workflow

The TSP code starts with a random tour and iteratively shortens it using the 2-opt improvement heuristic [54] until a local minimum is reached. ILCS automatically and asynchronously distributes unique seed values to each worker thread, runs the TSP code, reduces the results to find the best solution, and repeats these steps until the termination criterion is met. It employs two types of threads per node: a *master* thread (MPI process) that handles the communication and local work distribution and a set of *worker* threads (OpenMP threads) that execute the provided TSP code. The master thread forks a worker thread for each detected CPU core. Each worker thread continually calls `CPU_Exec()` to evaluate a seed and records the result (lines 14-20). Once the worker threads are running, the master thread's primary job is to scan the results of the workers to find the best solution computed so far (i.e., the local champion). This information is then globally reduced to determine the current system-wide champion (lines 22-32). ILCS terminates the search when the quality has not improved over a certain period (lines 33-34).

3.4.2 OpenMP Bug: Unprotected Memory Access

The memory accesses performed by the `memcpy` calls on lines 20 and 30 are protected by an OpenMP critical section. Not protecting them results in a data race that might lead to incorrect final program output. To simulate this scenario, we modified the ILCS source code to omit the critical section in worker thread 4 of process 6.

Table 3.6 lists the top suspicious traces that DiffTrace finds when injecting this bug. Each row presents the results for different filters and attributes. For example, the filter "11.mem.ompcit.cust.0K10" removes all function returns and .plt calls from the traces and only keeps memory-related calls, OpenMP critical-section functions, and the custom function "CPU_Exec". The "K10" at the end of filter means that the filtered traces are converted into an NLR with K=10. The bold numbers in the rightmost column of the table flag trace 6.4 (i.e., process 6, thread 4) as the trace that was affected the most by the bug.

The corresponding diffNLR(6.4) presented in Figure 3.8a clearly shows that the normal execution of ILCS (green and blue blocks) protects the `memcpy` while the buggy execution (green and red blocks) does not. Here, L0 represents `CPU_Exec`, which is called multiple times in both the fault-free and the buggy version (the call frequencies are different due to the asynchronous nature of ILCS).

3.4.3 MPI Bug: Deadlock Caused by Fault in Collective

By forcing process 2 to invoke MPI_Allreduce (line 24) with a wrong size, we can inject a *real deadlock*. Because the deadlock happens early in the execution, the resulting traces are very different from their fault-free counterparts. Consequently, DiffTrace marks almost all processes as suspicious (cf. Table 3.7). Clearly, this is not helpful for debugging. Nevertheless, diffNLR still yields useful information. Since most of the traces are suspicious, we do not know which one the real culprit is and randomly selected trace 4. By looking at the diffNLR(4) output shown in Figure 3.8b, we immediately see that both the normal and the buggy trace are identical up to the invocation of MPI_Allreduce. This gives the user the first (correct) hint as to where the problem lies. Beyond this point, the bug-free process continues to the end of the program (it reaches the MPI_Finalize call) whereas the buggy process does not. The last entry in the buggy trace is a call to MPI_Allreduce (the last green box), indicating that this call never returned, that is, it deadlocked. This provides the user with the second (correct) hint as to the type of the underlying bug.

3.4.4 MPI Bug: Wrong Collective Operation

By changing the MPI_MIN argument to MPI_MAX in the MPI_Allreduce call on line 24 of Listing 3.1, the semantics of ILCS change. Instead of computing the best answer, the modified code computes the worst answer. Hence, this code variation terminates but is likely to yield the wrong result. We injected this bug into process 0.

The first few suspicious processes listed in Table 3.8 are inconclusive. However, the filters that include MPI all agree that process 5 changed the most. Looking at the corresponding diffNLR(5) output in Figure 3.8c makes it clear why process 5 was singled out. In the buggy run, it executes many more MPI_Bcast calls than in the bug-free run because the frequency in which local "optimums" are produced has changed. Though this should affect all traces equally, which has reflected in the diffNLR of other traces. We are presenting these tables and figures to show that DiffTrace can reveal the impact of silent bugs like the wrong operation. Such data representation via suggested tables and diffNLRs helps developers to gain insight into the general behavior of the execution. More accurate results can be obtained by refining the parameters and collecting more profound traces (e.g., ParLOT(all images)). This would be part of our future work to find the set of parameters for different classes of bugs to maximize accuracy.

3.5 LULESH2 Examples

Our ultimate goal is to apply DiffTrace to complex HPC codes. As a more complex example, we have executed the single-cycle LULESH2[58] with 8 MPI processes and 4 OMP threads (system configuration described in §3.4) and collected ParLOT (main image) function calls.

Before bug injection, we analyzed LULESH2 traces and computed some statistics to gain insight into the general control flow of LULESH2 and also to evaluate DiffTrace's performance and effectiveness. Our primary results show that ParLOT instruments and captures **410 distinct function** calls on average per process, and stores them in compressed trace files of size less than **2.8 KB** on average per thread. Upon decompression, each per process trace file turns into a sequence of **421503** function calls on average. The equivalent NLR of each trace file reduces the sequence size by a factor of **1.92** and **16.74**, for constant K set to 10 and 50, respectively.

For further evaluation of DiffTrace, we injected a fault into the LULESH source code so that the process with rank 2 would not invoke the function `LagrangeLeapFrog` that is in charge of updating "domain" distances and send/receive MPI messages from other processes.

Table 3.9 reflects the ID of processes (rightmost column) that DiffTrace's ranking system suggests as the most affected traces by the bug. Since the fault in process 2 prevents other processes from making progress and successfully terminate, all of the process IDs appeared in the table. The generated diffNLRs clearly showed the point at which each process stopped making progress.

3.6 Related Work

Three major recent studies have emphasized the need for better debugging tools *and* the need to build a community that can share debugging methods and infrastructure: the DOE report mentioned earlier [34], an NSF workshop [15], and an ASCR report on extreme heterogeneity [105]. Our key contribution in this paper is a fresh approach to debugging that (1) incorporates methods to debug across the API-stack by resorting to binary tracing and thereby being able to "dial into" MPI bugs and/or OpenMP bugs (as shown in the ILCS case study), (2) makes initial triage of debugging methods possible via function-call traces, and (3) enables the verification community to cohere around DiffTrace by allowing other tools to extend our toolchain (they can tap into it at various places).

Many HPC debugging efforts have emphasized the need to highlight dissimilarities and incorporate progress measures on loops. We now summarize a few of them. AutomaDeD [9][64] captures the application's control flow via Semi Markov Models and detects outlier executions. PRODOMETER [73] detects loops in AutomaDeD models and introduces the notion of *least progressed tasks* by analyzing *progress dependency graphs*. DiffTrace's DiffNLR method does not (yet) incorporate progress measures; it only computes changes in loop *structure*. Prodometer's methods are ripe for symbiotic incorporation into DiffTrace. We also plan to incorporate *happens-before* computation as a progress measure using FCA-based algorithms by Garg et al. [30, 31]. FCA-based approaches have been widely used in data mining, machine learning, and information retrieval [52].

In terms of computing differences with previous executions, we draw inspirations from Zeller's delta-debugging [114] and De Rose et al.'s relative debugging [86]. The power of equivalence classes for outlier detection is researched in STAT [3], which merges stack traces from processes into a prefix tree, looking for equivalence-class outliers. STAT uses the StackWalker API from Dyninst [72] to gather stack traces and efficiently handles scaling issues through tree-based overlay networks such as MRnet [87]. D4 [68] detects concurrency bugs by statically analyzing source-code changes, and DMTracker [29] detects anomalies in data movement. The communication patterns of HPC applications can be automatically characterized by diffing the communication matrix with common patterns [88] or by detecting repetitive patterns [84]. ScalaTrace [83] captures and compresses communication traces for later replay. Synoptic [7] is applied to distributed system logs to find bugs.

3.7 Discussions & Future Work

DiffTrace is the first tool we know of that situates debugging around *whole program* diffing, and (1) provides user-selectable front-end filters of function calls to keep; (2) summarizes loops based on state-of-the-art algorithms to detect loop-level behavioral differences; (3) condenses the loop-summarized traces into concept lattices that are built using incremental algorithms; (4) and clusters behaviors using hierarchical clustering and ranks them by similarity to detect and highlight the most salient differences. We deliberately chose the path of a clean start that addresses missing features in existing tools and missing collectivism in the debugging community. Our initial assessment of this design is encouraging.

In our future work we will improve DiffTrace components as follows: (1) Optimizing them to exploit multi-core CPUs, thus reducing the overall analysis time; (2) Converting ParLOT traces into Open Trace Format (OTF2) by logically timestamping trace entries to mine temporal properties of functions such as *happened-before* [65]; (3) Conducting systematic bug-injection to see whether concept lattices and loop structures can be used as elevated features for precise bug classifications via machine learning and neural network techniques; and (4) Taking up more challenging and real-world examples to evaluate DiffTrace against similar tools, and release it to the community.

Acknowledgements: Supported in part by NSF awards CCF 1817073 and 1704715.

Table 3.2: The generated traces for odd/even execution with four processes

T_0	T_1	T_2	T_3
...	...	...	...
main	main	main	main
MPI_Init	MPI_Init	MPI_Init	MPI_Init
MPI_Comm_Rank	MPI_Comm_Rank	MPI_Comm_Rank	MPI_Comm_Rank
MPI_Comm_Size	MPI_Comm_Size	MPI_Comm_Size	MPI_Comm_Size
...	...	...	...
oddEvenSort	oddEvenSort	oddEvenSort	oddEvenSort
...	...	...	...
findPtr	findPtr	findPtr	findPtr
MPI_Send	MPI_Recv	MPI_Send	MPI_Recv
MPI_Recv	MPI_Send	MPI_Recv	MPI_Send
...	...	...	...
findPtr	findPtr	findPtr	findPtr
MPI_Send	MPI_Recv	MPI_Send	MPI_Recv
MPI_Recv	MPI_Send	MPI_Recv	MPI_Send
...	...	...	...
MPI_Finalize	MPI_Finalize	MPI_Finalize	MPI_Finalize

Table 3.3: NLR of Traces

T_0	T_1	T_2	T_3
MPI_Init	MPI_Init	MPI_Init	MPI_Init
MPI_Comm_Rank	MPI_Comm_Rank	MPI_Comm_Rank	MPI_Comm_Rank
MPI_Comm_Size	MPI_Comm_Size	MPI_Comm_Size	MPI_Comm_Size
L0 ˆ 2	L1 ˆ 4	L0 ˆ 4	L1 ˆ 2
MPI_Finalize	MPI_Finalize	MPI_Finalize	MPI_Finalize

Table 3.4: Formal Context of odd/even sort example

	MPI_Init()	MPI_Comm_Size()	MPI_Comm_Rank()	L0	L1	MPI_Finalize()
Trace 0	×	×	×	×		×
Trace 1	×	×	×		×	×
Trace 2	×	×	×	×		×
Trace 3	×	×	×		×	×

Table 3.5: Attributes mined from traces

Attributes {attr:freq}			
attr		freq	
Single	each entry of the trace	**Actual**	observed frequency
		Log10	log10 of the observed frequency
Double	each pair of consecutive entries	**noFreq**	no frequency

Table 3.6: Ranking table - OpenMP bug: unprotected shared memory access by thread 4 of process 6

Filter	Attributes	B-score	Top Processes	Top Threads
11.plt.mem.cust.0K10	doub.noFreq	0.244	7, 3, 4	**6.4**, 7.3, 1.4, 3.3, 3.4, 4.2
11.plt.mem.cust.0K10	doub.log10	0.244	7, 3, 4	**6.4**, 7.3, 1.4, 3.3, 3.4, 4.2
01.plt.mem.cust.0K10	doub.noFreq	0.244	7, 3, 4	**6.4**, 7.3, 1.4, 3.3, 3.4, 4.2
01.plt.mem.cust.0K10	doub.log10	0.244	7, 3, 4	**6.4**, 7.3, 1.4, 3.3, 3.4, 4.2
01.mem.ompcrit.cust.0K10	sing.log10	0.262	3	**6.4**, 7.1, 3.3, 4.1, 5.1, 6.1
01.mem.ompcrit.cust.0K10	sing.noFreq	0.262	3	**6.4**, 7.1, 3.3, 4.1, 5.1, 6.1
11.mem.ompcrit.cust.0K10	sing.log10	0.262	3	**6.4**, 7.1, 3.3, 4.1, 5.1, 6.1
11.mem.ompcrit.cust.0K10	sing.noFreq	0.262	3	**6.4**, 7.1, 3.3, 4.1, 5.1, 6.1
11.plt.mem.cust.0K10	doub.actual	0.273	7	**6.4**, 2.4, 3.4, 4.2, 4.4
01.plt.mem.cust.0K10	doub.actual	0.273	7	**6.4**, 2.4, 3.4, 4.2, 4.4

Table 3.7: Ranking table - MPI bug: wrong collective size in process 2

Filter	Attributes	B-score	Top Processes	Top Threads
11.mpicol.cust.0K10	sing.log10	0.439	0, 7, 2, 4, 5, 6	1.1, 1.3, 3.1, 3.2, 3.4
11.mpicol.cust.0K10	sing.noFreq	0.439	0, 7, 2, 4, 5, 6	1.1, 1.3, 3.1, 3.2, 3.4
11.mpi.cust.0K10	doub.noFreq	0.457	0, 7, 2, 4, 5, 6	1.4, 3.3, 3.4
11.mpi.cust.0K10	doub.actual	0.457	0, 7, 2, 4, 5, 6	1.4, 3.3, 3.4
11.mpiall.cust.0K10	doub.noFreq	0.457	0, 7, 2, 4, 5, 6	1.4, 3.3, 3.4
11.mpiall.cust.0K10	doub.actual	0.457	0, 7, 2, 4, 5, 6	1.4, 3.3, 3.4
11.mpicol.cust.0K10	doub.noFreq	0.457	0, 7, 2, 4, 5, 6	1.4, 3.3, 3.4
11.mpicol.cust.0K10	doub.actual	0.457	0, 7, 2, 4, 5, 6	1.4, 3.3, 3.4
11.mpi.cust.0K10	sing.log10	0.465	0, 7, 2, 4, 5, 6	1.1, 1.3, 3.1, 3.2, 3.4
11.mpi.cust.0K10	sing.noFreq	0.465	0, 7, 2, 4, 5, 6	1.1, 1.3, 3.1, 3.2, 3.4
11.mpiall.cust.0K10	sing.log10	0.465	0, 7, 2, 4, 5, 6	1.1, 1.3, 3.1, 3.2, 3.4
11.mpiall.cust.0K10	sing.noFreq	0.465	0, 7, 2, 4, 5, 6	1.1, 1.3, 3.1, 3.2, 3.4
11.mpi.cust.0K10	doub.noFreq	0.543	0, 7, 2, 4, 5, 6	1.4, 3.3, 3.4
11.mpi.cust.0K10	doub.actual	0.543	0, 7, 2, 4, 5, 6	1.4, 3.3, 3.4

Table 3.8: Ranking Table - MPI-Bug: Wrong Collective Operation ,Injected to Process 0

Filter	Attributes	B-score	Top Processes	Top Threads
01.plt.cust.0K10	doub.log10	0.271	2	6.2, 7.3, 2.2, 5.2, 5.3
11.plt.cust.0K10	doub.log10	0.271	2	6.2, 7.3, 2.2, 5.2, 5.3
01.plt.cust.0K10	sing.actual	0.276	1	3.1, 1.4, 6.4, 3.4
11.plt.cust.0K10	sing.actual	0.276	1	3.1, 1.4, 6.4, 3.4
01.plt.cust.0K10	doub.noFreq	0.285	2	6.2, 7.3, 2.2, 5.2, 5.3
11.plt.cust.0K10	doub.noFreq	0.285	2	6.2, 7.3, 2.2, 5.2, 5.3
01.plt.cust.0K10	sing.log10	0.292	1, 4, 5	3.1, 4.3
11.plt.cust.0K10	sing.log10	0.292	1, 4, 5	3.1, 4.3
01.**mpicol**.cust.0K10	sing.actual	0.312	**5**	3.2, 6.4, 5.4, 4.2
11.**mpicol**.cust.0K10	sing.actual	0.312	**5**	3.2, 6.4, 5.4, 4.2
11.**mpi**.cust.0K10	sing.actual	0.331	**5**	3.2, 6.4, 5.4, 4.2
11.**mpiall**.cust.0K10	sing.actual	0.331	**5**	3.2, 6.4, 5.4, 4.2
01.**mpiall**.cust.0K10	sing.actual	0.331	**5**	3.2, 6.4, 5.4, 4.2
01.**mpi**.cust.0K10	sing.actual	0.331	**5**	3.2, 6.4, 5.4, 4.2
11.**mpi**.cust.0K10	sing.actual	0.371	**5**	3.2, 6.4, 5.4, 4.2
11.**mpiall**.cust.0K10	sing.actual	0.371	**5**	3.2, 6.4, 5.4, 4.2

Table 3.9: Ranking Table for LULESH

Filter	Attributes	B-score	Top Processes
11.1K10	sing.noFreq	0.295	**2** , 3 , 4 , 5 , 6 , 7
01.1K10	sing.noFreq	0.354	0 , 1 , **2** , 3 , 4 , 5
01.1K10	sing.actual	0.383	**2** , 3 , 4 , 5 , 6 , 7
11.1K10	sing.noFreq	0.408	**2** , 3 , 4 , 5 , 6 , 7
11.1K10	sing.noFreq	0.408	**2** , 3 , 4 , 5 , 6 , 7
01.1K10	doub.noFreq	0.433	4 , 5 , 6
01.1K10	doub.noFreq	0.433	4 , 5 , 6
11.1K10	doub.noFreq	0.433	5 , 1 , 6
01.1K10	doub.noFreq	0.455	1 , **2** , 3 , 4 , 7
11.1K10	doub.noFreq	0.458	5 , 1 , 6
11.1K10	doub.noFreq	0.458	4 , 5 , 6 , 7
01.1K10	sing.log10	0.459	1 , **2** , 3 , 4 , 5 , 6
01.1K10	doub.noFreq	0.472	0 , 1 , **2** , 3 , 4 , 5
01.1K10	sing.log10	0.475	1 , 3 , 4 , 5 , 6 , 7
01.1K10	sing.log10	0.478	1 , **2** , 3 , 4 , 5 , 6
01.1K10	sing.log10	0.478	1 , **2** , 3 , 4 , 5 , 6

CHAPTER 4

AUTOMATED CONCURRENCY TESTING FRAMEWORK FOR GO

This chapter is based on the work submitted to the 2021 IEEE International Symposium on Workload Characterization (IISWC). We present GOAT, a combined static and dynamic concurrency testing and analysis tool that facilitate the process of debugging for real-world programs. Key ideas in GOAT include 1) automated dynamic tracing to capture the behavior of concurrency primitives, 2) systematic schedule space exploration to accelerate the bug occurance and 3) deadlock detection with supplementary visualizations and reports. We also propose a set of coverage requirements that characeterize the dynamic behavior of concurrency primitives and provide metrics to measure the quality of tests. Our evaluation on 68 curated real-world bug scenarios demonstrates that GoAT is significantly effective in detecting rare bugs, and its schedule perturbation method based on schedule yielding detects these bugs with less than three yields. These results together with the ease of deploying GoAT on real-world Go programs holds significant promise in field-debugging of Go programs.

4.1 Introduction

Go [46] is a statically typed language initially developed by Google. It employs channel-based Hoare's Communicating Sequential Processes (CSP) [38] semantics in its core and provides a productivity-enhancing environment for concurrent programming. Go enjoys accelerating acceptance in a wide variety of communities including container software systems [51, 71], distributed key-value databases [41, 97], and web server libraries [42]. It involves shared memory, message passing, non-deterministic message reception and selection, dynamic process creation, and programming styles that tend to create thousands of *goroutines* (i.e., application-level threads) and discard them to be garbage collected when they reach their final state. The combination of these features is well known for

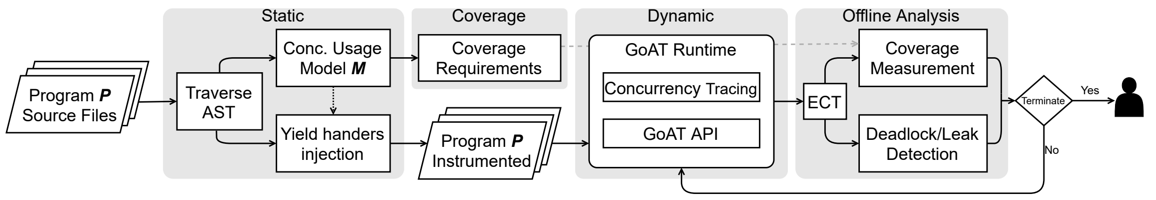


Figure 4.1: GOAT Overview

Go's popularity, yet they also make Go challenging to debug. Our work is especially relevant considering that there are no widely practical tools for debugging concurrent Go; even well-curated concurrency bug benchmark suites are only just now beginning to appear [104, 112].

In general, concurrent bugs are notoriously difficult to find and reproduce due to the non-deterministic choices that the scheduler makes during execution. In Go, constructs like *select* and buffered channels entangle the process of debugging by introducing extra randomness to the dynamic behavior of the program. Recent static [66, 67, 82, 93] and dynamic [20, 95, 96, 106, 115] techniques have been proposed to address these challenges. GoBench [112] gathers a collection of real concurrency bugs (GoReal) and simplified bug kernels (GoKer) from top 9 open-source projects written in Go and evaluated the effectivness such techniques in detecting the bug collection. Although static methods are proved to be rigorously effective in detecting flaws in small programs, they are not practical for realistic programs and often produce false positives. On the other hand, dynamic analysis approaches cover a more significant subset of real-world programs by constructing and analyzing an *execution model*. However, they focus on a specific class of bugs based on the symptom or cause of the bug. Also, for large codebases with thousands of LOC, it is non-trivial to capture an accurate dynamic execution model using source instrumentation or source-to-source translation. Furthermore, our experiments (section 4.4) observe that some buggy programs take more than 1,000 runs under different schedules before the bug is hit. Concurrent testing methods [5] are proposed to complement static and dynamic approaches in tackling challenges of concurrent debugging. To the best of our knowledge, there exist no such testing methods applicable to Go.

We implemented GOAT (**Go** **A**nalysis and **T**esting), a debugging framework for concurrent Go applications to address this lack. GOAT (figure 4.1) combines static and dynamic approaches to automatically analyze the behavior of concurrent components and facilitate the process of testing and debugging Go applications. Several classic ideas from literature are combined with novel ones to support modern concepts of Go in GOAT, which pursues three primary objectives:

Objective 1: *Accurate Dynamic Execution Modeling*— In order to study the behavior of concurrent components and track the state of the program during execution, a dynamic execution model has to be constructed and compared against a predefined model (e.g., formally defined specifications or the developer's mental representation of the program). It is crucial for debuggers and software analysis tools to construct their execution models as close as possible to the actual program execution context. Since a bug might occur at various levels of abstraction, *whole-program dynamic tracing* provides a practical and uniform way to track multiple facets of the program during execution [98]. We have enhanced the built-in tracing mechanism of Go [100] to capture the dynamic behavior of concurrency primitives in the form of a *sequence of events*, namely *execution concurrency trace* or ECT. Each event in ECT represents an *action* that corresponds to exactly one statement in the source code. An ECT provides a detailed model of how a concurrent program behaves dynamically and assists debugging procedures (e.g., bug detection, root-cause analysis, execution visualization). Our experiments show that by replaying the program's ECT, GOAT detects all blocking bugs of GoKer [112] many of which are undetected by existing debugging tools.

Objective 2: *Systematic Exploration of Schedule-Space*— Since the scheduler's non-deterministic behavior is the primary reason for Heisenbugs (i.e., errors that are uncommon to occur and hard to reproduce), these bugs may not manifest during conventional testing. By adopting ideas from *systematic concurrency testing* approaches [10, 13, 21–24, 39, 56, 75, 101, 111, 113], we perturb the native scheduler of Go to explore the unconventional but feasible execution interleaving. First, we statically identify the source location of concurrency primitive usages in a given program. We then inject handlers of context-switching calls around these locations to manage schedule perturbation. At its simplest form, handlers randomly (with a certain probablity and within a bound) decide if the current goroutine should continue executing or *yield* to other goroutines to execute first. Such yields change the blocking behavior of the program within the space of feasible states and exercise untested interleavings, consequently heighten the propensity for bug detection. The results of our experiments indicate that just a few random schedule perturbations can accelerate the exposure of rare bugs.

Objective 3: *Testing Quality Measurement*— A test suite's thoroughness is often judged by the coverage of certain aspects of the software, such as its source-code statements (a higher statement coverage indicates more thorough testing). In the context of concurrent software, exisiting coverage metrics [21, 39, 103, 110] characterize (quantify) the behavior of concurrency primitives which enables the quality measurement of schedule-space exploration. Such characterizations involve defining an initial set of requirements and a method for assessing whether or not those requirements are met during testing. Since Go combines traditional synchronization and serialization primitives (mutex, conditional variables) with message-passing and introduces new concepts such as *select-case* (non-deterministic communication and synchronization), new coverage requirements are required to characterize the behavior of Go concurrency. Using the GOAT's infrastructure, we studied the underlying causes of bugs in GoKer benchmark [112] and proposed a set of coverage requirements that 1) coherently characterize the dynamic behavior of concurrency primitives under various scheduling scenarios and 2) enable measurement of schedule-space exploration until reaching a threshold, or exposing the bug. By analyzing the test's ECT, we can identify if coverage requirements are met during testing. We demonstrate that our novel coverage metric is effective in measuring the schedule-space exploration progress.

To summarize, here are our main contributions:

- We introduce GOAT, a testing and analysis framework that facilitates whole-program trace collection (via an enhancement to the standard tracer package) and knowledge discovery about the program's dynamic behavior.

- We show the effectiveness of controlled preemptions for concurrency bug exposure in the context of a real-world language

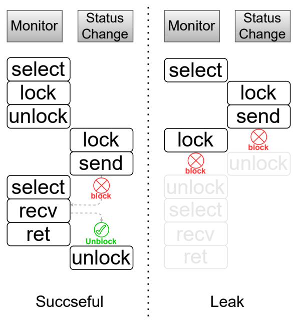
- We propose a set of coverage requirements that characterize the dynamic behavior of concurrency primitives, enabling measurement of quality and progress of schedule-space exploration.

```go
package main
import "sync"

type Container struct{
  sync.Mutex
  stop  chan struct{}
}

func main() {
  container := &Container{
        stop:make(chan struct{})}
  go Monitor(container)
  go StatusChange(container)
}
```

```go
func Monitor(cnt *Container){
  for{
    select{
    case <- cnt.stop:
      return
    default:
      cnt.Lock()
      cnt.Unlock()
}}}
func StatusChange(cnt *Container){
  cnt.Lock()
  defer cnt.Unlock()
  cnt.stop <- struct{}{}
}
```



Listing 1: Simplified version of bug `moby28462`

The rest of this paper is as follows: Section 4.2 discusses the fundamentals about concurrency debugging in Go and ideas behind GOAT. Section 4.3 illustreate the design and implementation of GOAT's components. The evaluation of GOAT on GoKer bug benchmark is illustrated in section 4.4. Section 4.5 discusses the related work and finally, section 4.6 summarizes and concludes.

4.2 Background

4.2.1 Go Concurrency

Go introduces a new concurrency model, mixing shared-memory features of languages like Java/C/C++ and message-passing concepts such as Erlang's, with an ad-hoc scheduler that orchestrates Go's concurrent components interactions while shielding the user from many low-level aspects of the runtime. The language is equipped with a rich vocabulary of *serialization* features to facilitate the memory model constraints [49]; they include synchronous and asynchronous communication, memory protection, and barriers for efficient synchronization:

- **Goroutines** are functions that execute concurrently on logical processors having their own stack.

- **Channels** are typed conduits through which goroutines communicate. Channels are unbuffered by default, providing synchronous (rendezvous) or asynchronous (via buffered channels) messaging between goroutines.

- **Synchronization** features such as *(RW)mutex*, *waitGroup*, *conditional variables*, *select*, and *context* are included in the language to provide more and flexible synchronization, data access serialization, memory protection, and error handling.
- **Scheduler** maintains goroutines in FIFO queues and binds them on OS threads to execute on processing cores.

This design facilitates the construction of data flow models that efficiently utilize multiple CPU cores and encourages developers to *share memory through communication* for safe and straightforward concurrency and parallelism. This rich mixture of features has, unfortunately, greatly exacerbated the complexity of debugging. In fact, the popularity of Go has outpaced its debugging support [40, 104, 112]. There are some encouraging developments in support of debugging, such as a data race checker [106] that has now become a standard feature of Go and has helped catch many a bug. However, the support for blocking bugs such as deadlocks and Go-specific bug-hunting support for Go idioms (e.g., misuse of channels and locks) remain insufficiently addressed.

Listing 1 shows a simplified version of a reported bug in Docker [43]. An instance of the `Container` type (lines 4-7) is created in the `main` function (lines 10-11). In line 12, a goroutine is spawned to execute function `Monitor` that continuously checks the container status and returns once it receives from the container's channel (lines 18-19). The default case of the `select` statement (line 20) allows `Monitor` to continue monitoring without getting blocked on the channel receive (line 18). Concurrent to the `main` and `Monitor` goroutines, another goroutine is created in line 13 to execute function `StatusChange` which changes the status of the container by sending to the container's channel. The container's lock is released after the send action completes and function returns (`defer` statement in line 26).

Native execution of this program terminates successfully without issuing any error or warning. Based on the Go specification and memory model, there is no constraint on the goroutines spawned from the `main` function to join back before the `main` goroutine[1] terminates. A deadlock detector within the runtime periodically checks that the scheduler queues of all *runnable* goroutines never become empty until the `main` goroutine terminates.

[1]In the remainder of the paper, we use *main function* and *main goroutine* interchangeably.

In other words, the runtime throws a deadlock exception when the `main` goroutine is blocked, and no other goroutine is in the queue to execute (i.e. *global deadlock*). Since there is no blocking instruction in the `main` goroutine in listing 1, the program terminates successfully regardless of other goroutines' statuses. However, this program suffers from a common bug in concurrent Go where one or more goroutines *leak* (i.e., *partial deadlock*) from the execution (i.e., never reach their end states).

The right side of the listing displays a successful run and a leak situation of the program. In the leak situation, first, the `Monitor` goroutine executes the `select` statement and, based on the available cases, picks the default case to execute. Right before the execution of mutex lock (line 21), the scheduler context-switches and the `StatusChange` goroutine starts its execution through which it holds the lock and blocks on sending to the channel (line 27) since there is no receiver on that channel. Upon blocking on send, the scheduler transfers back the control to the `Monitor` goroutine that tends to acquire the mutex, but because the mutex is already held by `StatusChange`, the `Monitor` goroutine also blocks. The circular wait between the container mutex and channel prevents both spawned goroutines from reaching their end states and leaves the program in an unnoticed deadlock situation.

4.2.2 Concurrency Bugs in Go

Based on a proposed bug taxonomy for Go [104], bugs are categorized separately based on their *causes* (shared-memory vs. message-passing) and *symptoms* (blocking vs. non-blocking). Blocking bugs historically refer to situations where one or more processing units (e.g., goroutines) are blocked, waiting for an external signal to resume (e.g., leak situation in listing 1. The observed causes of such blocking flaws in the context of Go are as follows:

- *Resource deadlocks*: Go inherits resource deadlocks from multithreaded languages like Java and C/Pthreads where goroutines are trapped in a circular wait for the resource (e.g., mutex) that is held by other goroutines.
- *Communication deadlocks*: Synchronized (unbuffered) channels transmit values from one goroutine to another in a rendezvous fashion. The sender (or receiver) blocks until the receiver (or sender) is ready to receive (send). Misuse of channel operations might result in one or more goroutines waiting for a sender/receiver to unblock them forever.

- *Mixed deadlocks*: The leak situation in listing 1 is the example of such deadlocks where one goroutine is blocked on acquiring a resource that is held by another goroutine which is blocked on communication.

Similar to other concurrent languages, Go has non-blocking bugs such as data races and atomicity violations while introducing new bug idioms due to its new concepts such as anonymous functions [104]. This work focuses on blocking bugs (i.e., partial and global deadlocks).

In addition to the non-deterministic nature of concurrent languages caused by the scheduler and interaction between concurrent components, Go introduces some level of non-determinism at the application level. The *select-case* statement (similar to switch-case) allows the goroutine to wait on multiple channel operations. The runtime picks one case pseudo-randomly among available cases (i.e., channel sends and receives that are ready to execute without blocking). If none of the cases are ready, the executing goroutine is blocked unless there is a *default* case. The default case makes the select non-blocking and prevents the goroutine from waiting for unavailable communications. Such random behavior expands the interleaving space, and it grows exponentially when nested selects are employed in conjunction with nested loops. As a result, tracing the cause of a program's misbehaved execution becomes increasingly tricky. Our observations (section 4.4) demonstrate that **select statements are involved at the center of many rare bugs**.

4.2.3 Accelerating Bug Exposure

Blocking bugs are primarily caused by the non-deterministic decisions that the scheduler makes. Such bugs may not manifest themselves in conventional testing and are difficult to reproduce. Figure 4.2 displays the histogram of 68 blocking bug kernels grouped by the number of trials that GOAT takes to detect them. Approximately 30% of bugs required more than one execution to happen and be detected by GOAT[2]. *stress testing* is a common way to detect such rare bugs by exercising the scheduler and examine the program's behavior in many executions. However, such testing is inefficient because some interleavings might get tested repeatedly while other feasible interleavings remain

[2]The figure and numbers are obtained from trials of GOAT on native execution of bug kernels without any randomization (i.e., $D = 0$).

untested. It has been empirically demonstrated that a small amount of randomness in each test execution can drastically reduce the number of iterations needed to find concurrency bugs [23,101]. For instance, forcing context-switches before synchronization/serialization operations in concurrent programs increases the probability of finding rare concurrent bugs [10]. In listing 1, a rare context-switch after the `select` statement in line 17 causes the lock operation on mutex *m* in line 21 of goroutine `Monitor` to *block* goroutine `StatusChange` on locking *m* in line 25 and causing a deadlock. Concurrency primitive usages (e.g., channel send/recv, mutex lock/unlock, select) are the *critical points* in the program because their behavior directly impacts the blocking behavior of Go programs. In GOAT, we statically identify such critical points and inject *yield* handlers before each concurrency primitive usage. During execution, the handlers randomly decide if the current goroutine should yield to other goroutines to execute. Results in section 4.4 show that such simple yields are effective in detecting rare bugs.

4.2.4 Testing Coverage Analysis

To demonstrate that testing has been thorough, *coverage metrics* are defined to measure the progress of tests and specify testing termination conditions. Coverage metric for the set of testing executions $\mathcal{T}$ is a set of *requirements* $\mathcal{R}$ that should get covered during testing iterations. We say requirement $R_i \in \mathcal{R}$ is covered during testing iteration $t_j \in \mathcal{T}$ if we can correlate an *action* during execution of t_j to R_i. For example, in *statement coverage*, which is a widely-used metric in testing sequential software, R is the set of source locations (file and line numbers) in the target program. R_i is covered by test execution t if the statement at location R_i is executed in t. The *coverage percentage* of a test $\mathcal{T}$ is the ratio of the requirements covered by at least one execution over the number of all requirements ($|R|$).

Concurrent software testing frameworks perform testing iterations to explore the schedule space and expose flaws. Depending on the class of target bug, different coverage metrics are proposed to quantify the quality of search space exploration. *Synchronization* coverage metrics such as *blocking-blocked* [21], *blocked-pair-follows* [103] and *synchronization-pair* [39] defined requirements to cover during testing for exposing blocking bugs (e.g. deadlocks). For example, the synchronization coverage model in [21] defines *blocking* and

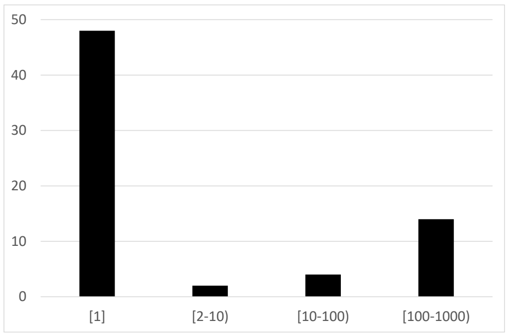


Figure 4.2: Distribution of number of trials for GOAT (D0) to detect 68 blocking bugs in GoKer [112]

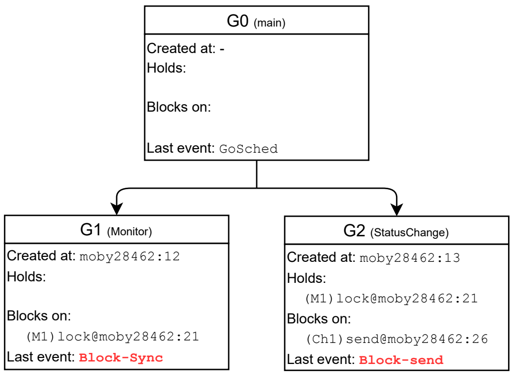


Figure 4.3: Goroutine tree of the leak situation in listing 1

blocked requirements per each synchronized block (i.e. mutually exclusive section of the code that is protected by a lock). The purpose of this requirement is to check if a test can report when there is a lock contention for two or more threads entering the synchronized block. That is, a thread is either *blocked* from entering the synchronization block or *blocking* other threads from entering by holding the lock.

The existing concurrency coverage metrics are primarily in the context of Java and C/Pthreads. They are not necessarily applicable to languages like Go as such languages have different concurrency primitives and semantics. Novel coverage metrics are required to enable the quantification of interleaving space exploration. Bron et al.,[8] enumerates four major characteristics for coverage metrics to gain acceptance:

1. **Static model:** A static model of requirements from the given program should be constructed by instrumenting the source code. The model should be well-understood by the developer or tester before execution. The model should maintain covered requirements during testing executions.

2. **Coverable and measurable requirements:** The absolute majority of requirements should be realistic enough to be *coverable* during testing. For a few that are not coverable (due to program semantics) or not *measurable* (because of technical limitations), the developer should be aware of the reason.

3. **Actions for uncovered requirements:** After testing terminates, every uncovered requirement should yield an action (e.g., extending testing iterations or removing dead code from the program, thus removing uncoverable requirements)

4. **Coverage satisfaction:** Some action should be taken upon reaching a threshold of coverage percentage (e.g., testing phase termination when reaching 100% statement coverage)

Defining a new coverage metric to satisfy the above characteristics requires an accurate and proper mental model of target bugs. Using the GOAT's infrastructure, we studied the underlying causes of many bugs, including GoKer benchmark [112] and propose a set of coverage requirements that enables extensive analysis of dynamic behavior of concurrency primitives under various scheduling scenarios. In section 4.3.3, we describe our proposed coverage metric for testing concurrent Go, which are extensible to all concurrent languages.

4.3 Design and Implementation

4.3.1 Overview

Figure 4.1 displays the overview of GOAT. Given a program **P** (i.e., a set of Go source files with a *main* function), GOAT automatically instruments **P** and constructs static and dynamic models of execution for thorough testing and analysis. Our main goal in GOAT is to facilitate the investigation of *non-deterministic interactions between concurrent components* (i.e., concurrent behavior) of **P** to achieve objectives acquainted in section 4.1.

Static Analysis: (section 4.3.2)— GOAT statically constructs a model **M** which is a table of source locations (files and line numbers) associated with concurrency primitive usages in **P** source files. The primary use of **M** is to identify locations in **P** as potential points for manipulating the schedule to explore possible scenarios and accelerate the discovery of rare bugs. Yield handlers are injected before each entry in **M** to decide if the following concurrency action (e.g., message send or mutex lock) should perform or yield to other goroutines. Such yields effectively perturb the scheduler and execute feasible but unconventional interleavings of **P**.

Coverage Requirements: (section 4.3.3)— Forcing the schedule perturbation is effective for exploring the feasible interleaving space until the bug is hit. However, a metric is required to evaluate the quality of interleaving space exploration and measure the progress until reaching a threshold. Following the factors of effective coverage metrics, we employ **M** to emulate the possible behavior of concurrent components of **P** and define a set of *coverage requirements* as indicators for quality and progress of schedule space exploration. The requirements are defined so that, during testing, a lack of requirement demands the user to fix the bug or remove the dead code.

Dynamic Analysis: (section 4.3.4)— To gain insight into the concurrent behavior of **P** and measure the covered requirements, we equipped GOAT with a dynamic tracing mechanism, which is an extension to the Go standard tracer package [33]. When tracing is enabled, an *execution concurrency trace* (ECT) file is generated once the execution of **P** terminates (e.g., successfully exits, fails, times out). ECT is a totally ordered *sequence* of events that contain information about the dynamic behavior concurrent components, enabling offline analysis of **P**'s execution.

Offline Analysis: (section 4.3.5)— In offline, GOAT first, separates the application-level events of ECT from the underlying runtime system of Go. Then, it constructs a goroutine tree from application-level goroutines to check if any goroutine has leaked/blocked (i.e., did not reach its final state) after the execution termination. Additionally, GOAT maintains a global goroutine tree for **P** and equivalences goroutines from run to run to accumulate the covered requirements from each execution of **P**. As soon as a bug is detected or the coverage exceeds a threshold, GOAT stops running and produces reports for manual analysis by the user.

4.3.2 Static Analysis

4.3.2.1 Concurrency Usage Model

GOAT statically constructs a model **M** from the usage of concurrency primitives in **P** files which enables uniform analysis during testing iterations. **M** is a table of source locations (files and line numbers) associated with *concurrency usages* (CU). We define CU as a triple of (f, l, k) where f is the file name, l is the line number, and k is the concurrency primitive used in the code location. $k \in \texttt{Channel} \cup \texttt{Sync} \cup \texttt{Go}$ where:

- $\texttt{Channel} = \{\texttt{send}, \texttt{receive}, \texttt{close}\}$
- $\texttt{Sync} = \{\texttt{lock}, \texttt{unlock}, \texttt{wait}, \texttt{add}, \texttt{done}, \texttt{signal}, \texttt{broadcast}\}$
- $\texttt{Go} = \{\texttt{go}, \texttt{select}, \texttt{range}\}$

GOAT constructs **M** by traversing the *abstract syntax tree* (AST) for each file in **P** using the Go AST package [47]. The first column of table 4.3 shows the CU locations extracted from program in listing 1.

4.3.2.2 Source Instrumentation

We employ **M** entries to instrument **P** with tracing and schedule perturbation mechanisms. First, we traverse the AST of **P** and inject three statements (i.e., AST nodes) to the beginning of **P**'s main function to enable end-to-end tracing:

- `goat_done := goat.Start()` initializes GOAT, enables tracing, and returns a channel as a conduit between application space and GOAT.
- `go goat.Watch(goat_done)` spawns a new goroutine as a watchdog for the liveness of the program (in case of global deadlock or infinite loop). The watchdog goroutine either receives from `goat_done` and sends back an ack signal or timeouts (default: 30 seconds). Then it stops tracing, flushing the trace buffer, and terminates.
- `defer goat.Stop(goat_done)` sends a value to the watcher goroutine after main returns and signals that the program is finished. Then it waits to receive the signal from the watcher, then stops tracing and terminates.

Moreover, we inject calls to `goat.handler()` before each CU in **M** to manipulate the native scheduler around concurrency primitve usages. `goat.handler()` is a function invocation that randomly calls `runtime.GoSched()` within a bound D to preempt the processing core from current goroutine and push the goroutine to the back of the global queue of runnable goroutines. When $D = 0$, GOAT does not perturb the scheduler and lets **P** to execute natively. For any $D > 0$, GOAT manipulates application-level goroutines from their regular execution D times. Our experiments (section 4.4) demonstrate that the optimum value for D is not larger than 3, showing that even a small number of yields is effective in exposing the bug (as also shown in [10]).

4.3.3 Coverage Requirements

Based on our investigations from the execution of Go applications and bug kernels, we emulate the possible behavior of concurrent components by defining a set of coverage requirements (summarized in table 4.1):

- **Req1 (Send/Recv):** {`blocked`, `unblocking`, `NOP`} – Goroutine G_1 is either *blocked* on a channel send (receive) if the receiver (sender) goroutine G_2 is not ready, or *unblocking* the waiting receiver (sender) goroutine G_2. A channel send or receive might also be neither blocked nor unblocking (NOP) for buffered channels.

- **Req2 (Select-Case):** {`blocked`, `unblocking`, `NOP`} × {`case`$_i$} – cases of select statements are channel sends and recives (or default case for non-blocking selects). For all select statements that has no default case, we obtain the cases of each select statement at runtime and maintain an instance of Req1 per case.

- **Req3 (Lock):** {`blocked`, `blocking`} – Goroutine G_i is either *blocked* when locking a mutex because another goroutine has locked the mutex or *blocking* other goroutines from acquiring the mutex lock.

- **Req4 (Unblocking):** {`unblocking`, `NOP`} – The goroutine that is performing channel close, mutex unlock, conditional variable signal and broadcast, waitGroup done, and non-blocking select case (send or receive) either *unblocks* one or more blocked goroutines or has no effect (NOP).

- **Req5-Go:** {`NOP`} – We emit a NOP action for each goroutine creation to indicate that it is covered during testing.

With the help of GOAT's infrastructure, the proposed requirements satisfy the characteristics of an "acceptable" coverage metric because:

Table 4.1: Coverge requirements defined for concurrent Go

Coverage Requirements	Concurrent Action	Coverage Requirement Types			
		Blocked	Unblocking	Blocking	NOP
Req. 1: Send/Recv	SEND	*	*		*
	RECV	*	*		*
Req. 2: Select-Case	CASE$_i$ (SEND)	*	*		*
	CASE$_i$ (RECV)	*	*		*
Req. 3: Lock	LOCK	*		*	
Req. 4: Unblocking	CLOSE		*		*
	UNLOCK		*		*
	SIGNAL		*		*
	BRDCST		*		*
	NB-SELECT		*		*
Req. 5: Go	Go				*

1. A *static model* **M** from program **P** is obtained by identifying its CU points. **M** is easy to understand by developers and reflects the concurrent behavior of **P**.

2. The defined requirements are *measurable* by analyzing the test's ECT. A global data structure maintains the covered requirements by each $t \in \mathcal{T}$.

3. Upon completion of $\mathcal{T}$ iterations, the *uncovered* requirements imply some *meaningful* information about the behavior of **P**. For example, if a send is always performing as *unblocking* and never as *blocked*, it means that the receiver always performs receive before the sender reaches its send instruction. In other words, the receive action *always happen-before* send action. This communication pattern might be part of **P**'s semantics and matches the developer's expectations (e.g., a set of goroutines are listening on a channel to perform non-frequent requests). Otherwise, the uncovered requirement "send-`blocked`" reflects a bug or flaw in the program design.

4. Since GOAT can detect occurred blocking bugs and maintain a global coverage model, $\mathcal{T}$ iterations terminate either by detecting a bug or reaching a coverage percentage threshold.

4.3.4 Dynamic Concurrency Tracing

The standard execution tracer package [33, 48] provides dynamic tracing for the construction of execution models from the interactions of processors, OS threads, goroutines, the scheduler, and the garbage collection mechanism. The tracing mechanism is compiled into all programs always through the runtime and is enabled on demand to study *perfromance bottlenecks* through visualizers like *pprof* [89]. The alphabet of trace events is total of 49 events [50], categorized and summarized in table 4.2. Although the event vocabulary is rich enough to model comprehensive goroutine latency and blocking behavior accurately, the vocabulary lacks concurrency primitive usage events for the construction of concurrency models. We enrich the standard tracing mechanism with 14 additional events to enable the production of dynamic models from the program's concurrency behavior:

- **Channel:** For each channel operation (make, send, receive, close), ECT includes an event with a unique id assigned to each channel.

Table 4.2: Event categories by the Go execution tracer

Category	Description
Process	Indication of process/thread start and stop
GC/Mem	Garbage collection and memory operation events
Goroutine	Goroutines events: create, block, start, stop, end, etc.
Syscall	Interactions with system calls
Users	User annotated regions and tasks (for bounded tracing)
Misc	System related events like futile wakeup or timers

- **(RW)Mutex, WaitGroup & Conditional Variables:** Similar to channels, we assign a unique id to each concurrency object and emit an event for each of their operations (lock, unlock, rlock, runlock, add, wait, signal, broadcast).
- **Select & Schedule:** The scheduler and the *select* structure introduce non-determinism to the execution. We keep track of the decisions made by the scheduler and select statements to obtain an accurate decision path during execution.

We call the output of enhanced tracer *execution concurrency trace* (ECT). ECT is a totally ordered *sequence* of events in which the order is approximated through a central clock with nanosecond precision. ECT also contains the call-stack for each event, enabling a direct mapping of events to source-line numbers. For each blocking operation (channels *sends/recvs*, mutex *locks*, waitGroup/conditional variable *wait* and *select* (when none of the cases are available)), ECT captures a pair of pre-operation and post-operation events to distinguish between the *request for action* and *completion of action*. Hence, ECT is especially effective for debugging because it enables modeling the blocking state of concurrent components at any given step of program execution. The enhanced dynamic tracing also enables the measurement of coverage requirements in offline (section 4.3.5).

4.3.5 Offline Analysis

In the lifetime of Go programs' executions, the runtime system creates new goroutines or pick from the pool of dead goroutines to perform various tasks such as bootstrapping the program, garbage marking and sweeping, and tracing. GOAT also adds an extra goroutine to *watch* the program execution in case of the main goroutine blockage. These goroutines are captured during tracing, but our focus is on the goroutines created from within the application. The distinguishment between runtime goroutines and application goroutines is essential to define the boundaries of the application and separate them from the underlying system. We say a goroutine is an *application-level* goroutine if it is the main goroutine (that executes the main function) or it has all of the following conditions: 1) its ancestor is the main goroutine, 2) it is not a Go runtime system goroutine, and 3) it is not a tracer goroutine. These conditions are assessed for every captured goroutine in ECT by checking the call stack of their corresponding *GoCreate* event.

GOAT constructs a *goroutine tree* (figure 4.3) of application-level goroutines from the generated ECT. Nodes in the goroutine tree represent a goroutine, and directed edges denote parent-child relationships in which the child is created from a `go` statement that the parent executes. Each node of the tree contains the entire sequence of events that the goroutine executed [3], information about the goroutine's creation site, the resources it holds at each execution point, and the final executed event right before the program termination. GOAT analyzes this collection of information to check whether any goroutine leaked after termination and whether the coverage requirements are covered.

4.3.5.1 Leak Detection

When tracing is enabled, every application goroutine invokes the tracer to capture *GoEnd* before finishing its execution. Before the main function returns, the main goroutine hands over the control to the root goroutine to finalize the program termination. This context-switch is done through invocation of `runtime.Gosched()` which emits the *GoSched* event. In GOAT, the main goroutine's final event in a successful execution is *GoSched* with `runtime.traceStop` on top of its stack.

We call an execution **successful**, if below conditions hold:

1. (1) all goroutines spawned in the main goroutine has *GoEnd* as their final event

2. (2) the final event of the main goroutine is *GoSched* with `runtime.traceStop` on top of its stack.

In the absence of any of these conditions, we conclude that the program suffers from a "deadlock" bug, because at least one goroutine did not reach its final state. Therefore,

[3]The sequence of events per node is not shown in figure 4.3 for simplicity.

GoAT executes procedure 2 which is a BFS traversal on the goroutine tree to check if the program suffers from partial or global deadlocks.

When a deadlock is detected, GoAT generates visualizations such as executed interleaving (listing 1) and goroutine tree (figure 4.3). The detailed report magnifies the scenario under which the bug has occurred and displays the final concurrent state of the program right before the failure. In addition, custom logs and reports such as the "happens-before" log of a set of goroutines and their associated resources are easily generated through a replay of ECT. Samples of such reports and visualizations are available in [appendix or online link]

4.3.5.2 Coverage Measurement

Once the program execution terminates, GoAT checks whether the extracted coverage requirements are covered during execution. A mapping between ECT dynamic concurrent events and statically obtained CU points is emitted by matching their respective call-stack and CU source location. Through a BFS traversal of the goroutine tree, we add a *coverage vector* to each goroutine node from the emitted mapping. Each element of the coverage vector is the respective covered value of the coverage requirement for the current goroutine node. During executions of tests $t \in \mathcal{T}$, we maintain and update a global goroutine tree

```
DeadlockCheck(G):
    if cur.lastEvent ≠ GoSched then
        return Global Deadlock
    end
    toVisit = [G.children]
    for |toVisit| ≠ 0 do
        cur = toVisit[0]
        if cur.lastEvent ≠ GoEnd then
            return Partial Deadlock (leak)
        end
        for n in cur.Children do
            append n to toVisit
        end
        toVisit = toVisit[1 :]
    end
    return Pass
```

Procedure 2: `DeadlockCheck` procedure with root node of goroutine tree (main goroutine) as input

Table 4.3: Concurrency Usages and coverage requirements of program in listing1

CU of list. 1		Coverage Requirements	Covered by		Overall
Line	Kind		run #1	run #2	Covered
12	go	covered	$\checkmark_{G0}$	$\checkmark_{G0}$	✓
13	go	covered	$\checkmark_{G0}$	$\checkmark_{G0}$	✓
17	select	c-recv-blocked	$\checkmark_{G1}$		✓
		c-recv-unblocking	$\checkmark_{G1}$		✓
21	lock	blocked		$\checkmark_{G1}$	✓
		blocking	$\checkmark_{G1}$		✓
22	unlock	unblocking	$\checkmark_{G1}$		✓
		no_op			
25	lock	blocked	$\checkmark_{G2}$		✓
		blocking		$\checkmark_{G2}$	✓
26	send	blocked	$\checkmark_{G2}$	$\checkmark_{G2}$	✓
		unblocking			
		no_op			
27	unlock	unblocking			
		no_op	$\checkmark_{G2}$		✓
		Coverage %	60%	33%	73%

after each t. It is crucial to maintain a global goroutine tree to measure the progress of coverage percentage over tests in $\mathcal{T}$. However, equivalencing between two goroutines and their respective coverage vectors from different executions is non-trivial. We say two goroutines G_m and G_n in the tests t_i and t_j are *equivalent* (i.e. falls into a identical node in the global goroutine tree) if their parents are equivalent and their creation source location (CU of kind go) are identical.

$$G_m \equiv G_n \text{if} \begin{cases} \text{parent}(G_m) \equiv \text{parent}(G_n) \wedge \\ \text{CU}(G_m).\text{file} = \text{CU}(G_n).\text{file} \wedge \\ \text{CU}(G_m).\text{line} = \text{CU}(G_n).\text{line} \end{cases} \tag{4.1}$$

4.4 Evaluation

4.4.1 Deadlock Detection

We assess the ability of GOAT and its variations in detecting bugs with minimum number of executions required to expose the bug. We have compared GOAT against three existing dynamic detectors:

- *Built-in* deadlock detector: It is an embeded mechanism in the standard Go runtime. The mechanism periodically makes sure that the queue of *runnable* goroutines is never empty until the main goroutine terminates. If the queue is empty and main has not terminated yet (i.e. main is blocked), it throws a runtime error.

Table 4.4: Output of each tool on GoKer [112] blocking bugs. Detected bug (minimum # of executions required) – **X (1000)**: the tool is not able to detect any bug after 1000 executions. **PDL**: Partial Deadlock, **GDL**: Global Deadlock, **PDL-k**: Partial Deadlock with *k* number of goroutines leaked. **DL**: A warning for potential deadlock is issued. **TO/GDL**: The global deadlock is detected because none of goroutines made any progress after 20 seconds, **CRASH**: The execution paniced because of a flaw in the execution (e.g., send on closed channel panic), **HANG**: The tool halt for more than 10 minutes.

Bug Description			Debugging Tools							
Cause	SubCause	Bug ID	BUILTINDL	GOLEAK	LOCKDL	GOAT				
						D0	D1	D2	D3	D4
Communication Deadlock	Channel	cockroach_2448	X (1000)	X (1000)	X (1000)	CRASH (1)	CRASH (1)	CRASH (1)	CRASH (1)	CRASH (1)
		cockroach_24808	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		cockroach_25456	GDL (1)	GDL(1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		cockroach_35073	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		cockroach_35931	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		etcd_6857	X (1000)	PDL (325)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (11)	PDL-1 (3)	PDL-1 (3)
		grpc_1275	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		grpc_1424	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		grpc_660	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		istio_17860	X (1000)	PDL (1)	X (1000)	PDL-1 (2)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		kubernetes_38669	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		kubernetes_5316	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-2 (1)	PDL-1 (1)	PDL-2 (1)	PDL-2 (1)
		kubernetes_70277	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		moby_21233	X (1000)	PDL (1)	X (1000)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)
		moby_33293	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (3)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		moby_4395	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		syncthing_5795	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
	Channel & Conditional Variable	kubernetes_11298	X (1000)	X (1000)	TO/GDL (179)	X (1000)	TO/GDL (352)	TO/GDL (158)	TO/GDL (179)	TO/GDL (179)
		moby_27782	X (1000)	PDL (741)	X (1000)	X (1000)	PDL-2 (1)	PDL-2 (1)	PDL-2 (4)	PDL-2 (4)
	Channel & Context	cockroach_10790	X (1000)	PDL (3)	X (1000)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)
		cockroach_13197	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		cockroach_13755	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		cockroach_18101	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		grpc_862	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		istio_18454	X (1000)	PDL (13)	X (1000)	PDL-2 (5)	PDL-1 (14)	PDL-1 (4)	PDL-1 (6)	PDL-1 (6)
		kubernetes_25331	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		moby_33781	X (1000)	PDL (1)	X (1000)	PDL-1 (221)	PDL-1 (10)	PDL-1 (8)	PDL-1 (10)	PDL-1 (10)
	Condition Variable	moby_29733	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		moby_30408	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
Mixed Deadlock	Channel & Lock	etcd_6873	X (1000)	PDL (371)	X (1000)	PDL-2 (1)	PDL-2 (2)	PDL-2 (7)	PDL-2 (6)	PDL-2 (6)
		etcd_7443	X (1000)	X (1000)	X (1000)	X (1000)	PDL-1 (9)	PDL-1 (15)	PDL-1 (14)	PDL-1 (14)
		etcd_7492	HANG (1)	HANG (1)	TO/GDL (4)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		etcd_7902	X (1000)	PDL (1)	X (1000)	PDL-4 (1)	PDL-4 (1)	PDL-4 (1)	PDL-4 (1)	PDL-4 (1)
		grpc_1353	X (1000)	PDL (1)	X (1000)	CRASH (1)	CRASH (1)	PDL-3 (1)	PDL-3 (1)	PDL-3 (1)
		grpc_1460	X (1000)	PDL (1)	X (1000)	PDL-2 (135)	PDL-2 (1)	PDL-2 (2)	PDL-2 (1)	PDL-2 (1)
		istio_16224	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		kubernetes_10182	X (1000)	PDL (44)	X (1000)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)
		kubernetes_1321	X (1000)	PDL (307)	X (1000)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		kubernetes_26980	GDL (375)	GDL (131)	X (1000)	TO/GDL (191)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		kubernetes_6632	X (1000)	X (1000)	X (1000)	PDL-2 (1)	PDL-2 (1)	PDL-2 (2)	PDL-2 (1)	PDL-2 (1)
		moby_28462	X (1000)	PDL (5)	X (1000)	PDL-2 (39)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)
		serving_2137	X (1000)	X (1000)	X (1000)	X (1000)	X (1000)	TO/GDL (88)	X (1000)	X (1000)
	Channel & WaitGroup	cockroach_1055	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		cockroach_1462	X (1000)	X (1000)	TO/GDL (1)	X (1000)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
	Misuse WaitGroup	moby_25384	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
Resource Deadlock	Double locking	cockroach_584	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		cockroach_9935	X (1000)	PDL (1)	DL (721)	PDL-1 (1)	PDL-1 (2)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		etcd_10492	GDL (1)	GDL (1)	DL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		etcd_5509	X (1000)	GDL (766)	TO/GDL (426)	X (1000)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		etcd_6708	GDL (1)	GDL (1)	DL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		grpc_3017	GDL (4)	GDL (4)	TO/GDL (3)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		grpc_795	GDL (1)	GDL (1)	DL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		hugo_5379	GDL (1)	GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		moby_17176	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		moby_36114	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		moby_7559	X (1000)	PDL (1)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		syncthing_4829	GDL (1)	GDL (1)	DL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
	RWR deadlock	cockroach_16167	X (1000)	X (1000)	DL (1)	X (1000)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		cockroach_3710	X (1000)	X (1000)	DL (123)	PDL-2 (28)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)
		cockroach_6181	X (1000)	PDL (1)	X (1000)	PDL-4 (1)	PDL-4 (1)	PDL-3 (1)	PDL-1 (1)	PDL-1 (1)
		hugo_3251	GDL (1)	GDL (1)	DL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)	TO/GDL (1)
		kubernetes_58107	X (1000)	X (1000)	X (1000)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)	PDL-1 (1)
		kubernetes_62464	X (1000)	X (1000)	DL (6)	PDL-2 (161)	PDL-2 (7)	PDL-2 (2)	PDL-2 (3)	PDL-2 (3)
	AB-BA deadlock	cockroach_10214	X (1000)	X (1000)	X (1000)	PDL-2 (368)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)	PDL-2 (1)
		cockroach_7504	X (1000)	X (1000)	X (1000)	PDL-2 (199)	PDL-2 (1)	PDL-2 (7)	PDL-2 (1)	PDL-2 (1)
		kubernetes_13135	X (1000)	PDL (1)	DL (4)	PDL-2 (1)	PDL-2 (5)	PDL-2 (1)	PDL-2 (22)	PDL-2 (22)
		kubernetes_30872	X (1000)	PDL (338)	X (1000)	PDL-3 (50)	PDL-3 (2)	PDL-3 (1)	PDL-3 (6)	PDL-3 (6)
		moby_4951	X (1000)	PDL (120)	X (1000)	PDL-2 (15)	PDL-2 (2)	PDL-2 (2)	PDL-2 (1)	PDL-2 (1)
Total Bugs:	68	Total Detected:	19	56	26	60	67	68	67	67

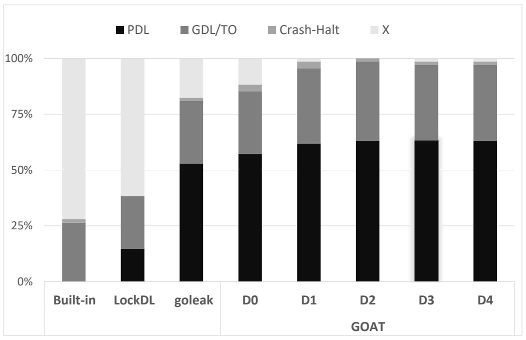


Figure 4.4: Histogram of detected bugs by each tool performed on 68 GoKer blocking bugs. PDL: partial deadlock, GDL/TO: global deadlock, Crash/Halt: causes the program to crash or halt during detection, X: nothing is detected

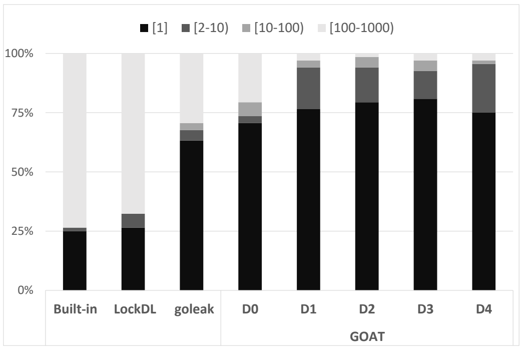


Figure 4.5: Histogram of required number of iterations by each tool to detect 68 GoKer blocking bugs

- *LockDL* [44]: This tool intercept with all mutex locks and unlocks of the target application to maintain a "lock-set" data structure. *LockDL* issues warning during runtime when it finds a circular wait in the lock-set or double-locking the same lock. It has a timeout mechanism for application that traps into global deadlocks (30 seconds by default).

- *goleak* [45]: This leak detector from Uber checks the program stack at the end of the main goroutine's execution to find the application-level goroutines that remained in the stack (i.e. leaked).

All experiments are performed on a server with two AMD Ryzen 5 3600 6-Core Processor (12 total cores with 2 threads per core and 6 cores per socket), 64 GB of RAM with generic Ubuntu 4.15.0 and Go version 1.15.6. Table 4.4 shows the details of results obtained from executing each tool per bug 1000 times. We show that the tool is unable to detect the bug after 1000 executions with **X (1000)**.

Figure 4.4 and table 4.4 show that variations of GOAT outperforms other detector by discovering the bug in 100% of the GoKer blocking benchmark. Figure 4.5 and highlighted cells of table 4.4 show that the idea of injecting random delays around concurrency usage points in the program drastically reduces the required number of testing iterations until the bug occur. D0 means GOAT did not delay the program at any point and D4 means that the target program has been delayed up to four times around its CU points. Figures

4.4 and 4.5 also state that the increase in the delay bound of GOAT does not necessarily increase the chance of exposing the bug. For example, the row of bug `serving_2137` in table 4.4 show that only GOAT D2 were able to detect the bug.

4.4.2 Coverage Analyis

We picked two representative bug kernels `etcd7443` and `kubernetes11298` to evaluate the coverage idea on them as they both have extensive use of channels, mutexes, conditional variables, nested selects within nested for loops and the buggy interleaving is proved to be rare to happen. figures 4.6 and 4.7 show the gradual increase in coverage percentage during execution runs for different values of D. Recall that D is the bound on the number of yields that we inject to the native execution of a given program to perturb the scheduler around concurrency usages. These figures show that the increase of number of delays grows the rate of coverage percentage. However, higher number of delays do not necessarily increase the coverage (D2 and D4 in figure 4.7). The drop in coverage for D1 in figure 4.7 is because of the new coverage requirements (e.g., a new goroutine is spawned and executing some concurrency primitives) that were encountered during testing execution.

4.5 Related Work

4.5.1 Go Correctness

Decades of research effort have been dedicated to the logical and performance correctness of concurrent and parallel programs. For CSP-based concurrent languages like Go, static (source-level) analysis methods [66, 67, 82] tend to assure bug freedom and verify safety properties through abstractions like session types and choreography synthesis. Ng and Yoshida [82] first proposed a static tool to detect global deadlock in Go programs

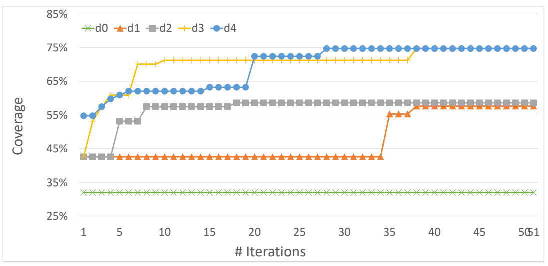


Figure 4.6: etcd7442 coverage

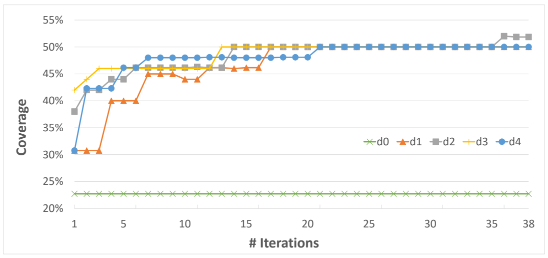


Figure 4.7: kuberenetes11298 coverage

using choreography synthesis. Later, Stadtmuller et al. [93] proposed a static trace-based global detection approach based on forkable regular expressions. Lange et al. proposed more static verification frameworks for checking channel safety, and liveness [66], and behavioral model checking [67]. Both methods approximate Go programs with session types and behavioral contracts extracted from their SSA intermediate representation. The mentioned work has limitations for handling dynamic (e.g., in-loop) goroutine or channel creation. They also do not scale and are impractical in real-world programs due to the state explosion problem and lack of proper front-end interface [112]. Besides, similar to other static analysis methods, they often suffer from false positives due to conservative constraints.

Dynamic (runtime-level) analysis approaches [20,96] rely on code instrumentation and program rewrites to obtain and analyze an *execution model*. Zhao et al. [115] introduced a runtime monitoring approach for deadlock detection for Occam programs based on wait-for graphs and some heuristics. Occam is a concurrent language based on CSP semantics, and similar to Go, it uses channels to establish communication between processes. Sulzmann and Stadtmuller proposed a dynamic verification approach for synchronous (unbuffered) channels [95], and a vector-clock-based approach for asynchronous channels [96]. Although they may support a larger subset of the Go language, they only focus on channels as the root cause of deadlocks and evaluated only on relatively small examples. Also, they usually do not scale for real-world Go applications with thousands of goroutines and LOC [19].

Standard Go comes with a few dynamic analysis tools. For example, the *race detector* [106] which is basically a wrapper around ThreadSanitizer [91], tracks memory accesses and detect races that happened during execution. A few other facilities for code coverage measurement, profiling, and tracing [48] are provided to deliver insight into the testing quality and performance behavior. The built-in race detector [106], despite its limitations (e.g. supporting up to 8192 goroutines), has proved to be effective in dynamically detecting data races in most cases quickly.

4.5.2 Systematic Testing

Systematic testing combines ideas from static and dynamic approaches to reduce the state space and reflect realistic behavior. Assuming the scheduler causes concurrency bugs (and not the program input), they may not manifest during conventional testing and difficult to reproduce, both due to non-deterministic decisions that the scheduler makes. Researchers have applied different methods [101] to reduce the interleaving space to explore effectively and efficiently. Delay-bounded [10, 23] and preemption-bounded [75] techniques systematically "fuzz" the scheduler to equally and fairly cover feasible interleaving. Other tools like Maple [111], CalFuzzer [56], and ConTest [21, 22] *actively* control the scheduler to maximise a pre-defined concurrency coverage criterion [39] or the probability of bug exposure [10].

4.6 Summary & Future Work

We presented GOAT, an analysis and testing framework for concurrent Go applications to assist concurrency debugging of real-world applications. GOAT combines static and dynamic methods to model and explore application execution. The scheduler behavior is pertubed with automatically injected random delays to accelerate the exposure of bug, if any. By dynamic measurment of a set of coverage requirements, we quantify the quality of schedule-space exploration of GOAT. GOAT detects all 68 blocking bugs of GoKer benchmark which are the bug kernels of top nine open-soruce projects written in Go. The schedule perturbation showed effectiveness in accelerating the bug exposure. Proposed coverage requirements accurately reflect the dynamic behavior of program executions and testing iterations.

Engineering of GOAT is flexible and extensible to more advanced components. For example, current minimal GOAT engine can be extended to take the full control over the Go scheduler and "guide" testing towards untested interleaving. We are dockerizing GOAT for easy and public use. We want to test on real-world programs. The data that ECT includes is rich enough for training accurate models and apply machine learning methods to learn and predict bug patterns.

CHAPTER 5

CONCLUSION

This dissertation presented a series of tools that combine concurrent debugging approaches to aid debugging using efficient data collection and automated analysis frameworks. We presented PARLOT, a portable low overhead dynamic binary instrumentation-based whole-program tracing approach that can support various dynamic program analyses, including debugging. Key properties of PARLOT include its on-the-fly trace collection and compression that reduces timing jitter, I/O bandwidth, and storage requirements to such a degree that whole-program call/return traces can be collected efficiently even at scale. We evaluated various versions of PARLOT created by disabling/enabling compression. Our metrics include the tracing overhead, required bandwidth, achieved compression ratio, initialization overhead, and the overall impact of compression. Detailed evaluations on the NAS parallel benchmarks running on up to 1024 cores establish the merit of our tool and our design decisions. PARLOT can collect more than 36 MB worth of data per core per second while only needing 56 kB/s of bandwidth and slowing down the application by 2.7x on average. These results are auspicious in supporting whole program tracing and debugging, particularly considering that most of the overhead is due to the DBI tool and not PARLOT.

We described the design of DiffTrace, the first tool we know of that situates debugging around *whole program* diffing, and (1) provides user-selectable front-end filters of function calls to keep; (2) summarizes loops based on state-of-the-art algorithms to detect loop-level behavioral differences; (3) condenses the loop-summarized traces into concept lattices that are built using incremental algorithms; (4) and clusters behaviors using hierarchical clustering and ranks them by similarity to detect and highlight the most salient differences.

Lastly, we presented GOAT, an end-to-end analysis and debugging framework for concurrent Go applications. GOAT combines static and dynamic methods to gather evidence about the dynamic behavior of concurrency primitives, model the program's concurrency behavior, and explore the interleaving space to reveal the flaws. GOAT detects all 68 blocking bugs of GoKer benchmark, which are the bug kernels adopted from the top nine open-source projects written in Go. Moreover, by defining a set of coverage requirements and dynamic measurement, we quantify the quality of the schedule-space exploration of GOAT. Proposed coverage requirements accurately reflect the dynamic behavior of program executions and testing iterations. The engineering of GOAT is flexible and extensible to more advanced components.

REFERENCES

[1] X. Aguilar, K. Fürlinger, and E. Laure, *Online mpi trace compression using event flow graphs and wavelets*, Procedia Computer Science, 80 (2016), pp. 1497 – 1506. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[2] ——, *Online mpi trace compression using event flow graphs and wavelets*, Procedia Computer Science, 80 (2016), pp. 1497 – 1506. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[3] D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz, *Scalable temporal order analysis for large scale debugging*, in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, Nov 2009, pp. 1–11.

[4] A. Allinea, *Allinea DDT*.

[5] V. Arora, R. K. Bhatia, and M. Singh, *A systematic review of approaches for testing concurrent programs*, Concurr. Comput. Pract. Exp., 28 (2016), pp. 1572–1611.

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, *The nas parallel benchmarks&mdash;summary and preliminary results*, in Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing '91, New York, NY, USA, 1991, ACM, pp. 158–165.

[7] I. Beschastnikh, J. Abrahamson, Y. Brun, and M. D. Ernst, *Synoptic: Studying logged behavior with inferred models*, in Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, New York, NY, USA, 2011, ACM, pp. 448–451.

[8] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur, *Applications of synchronization coverage*, in Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '05, New York, NY, USA, 2005, Association for Computing Machinery, p. 206–212.

[9] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, D. H. Ahn, and M. Schulz, *Automaded: Automata-based debugging for dissimilar parallel tasks*, in 2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN), June 2010, pp. 231–240.

[10] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, *A randomized scheduler with probabilistic guarantees of finding bugs*, in Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages

and Operating Systems, ASPLOS XV, New York, NY, USA, 2010, Association for Computing Machinery, p. 167–178.

[11] M. Burtscher, H. Mukka, A. Yang, and F. Hesaaraki, *Real-time synthesis of compression algorithms for scientific data*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, Piscataway, NJ, USA, 2016, IEEE Press, pp. 23:1–23:12.

[12] M. Burtscher and H. Rabeti, *A scalable heterogeneous parallelization framework for iterative local searches*, in 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, May 2013, pp. 1289–1298.

[13] M. Christakis, A. Gotovos, and K. Sagonas, *Systematic testing for detecting concurrency errors in erlang programs*, in 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, pp. 154–163.

[14] S. Claggett, S. Azimi, and M. Burtscher, *SPDP: An Automatically Synthesized Lossless Compression Algorithm for Floating-Point Data*, in 2018 Data Compression Conference, 2018.

[15] A. Cohen, X. Shen, J. Torrellas, J. Tuck, Y. Zhou, S. Adve, I. Akturk, S. Bagchi, R. Balasubramonian, R. Barik, M. Beck, R. Bodik, A. Butt, L. Ceze, H. Chen, Y. Chen, T. Chilimbi, M. Christodorescu, J. Criswell, C. Ding, Y. Ding, S. Dwarkadas, E. Elmroth, P. Gibbons, X. Guo, R. Gupta, G. Heiser, H. Hoffman, J. Huang, H. Hunter, J. Kim, S. King, J. Larus, C. Liu, S. Lu, B. Lucia, S. Maleki, S. Mazumdar, I. Neamtiu, K. Pingali, P. Rech, M. Scott, Y. Solihin, D. Song, J. Szefer, D. Tsafrir, B. Urgaonkar, M. Wolf, Y. Xie, J. Zhao, L. Zhong, and Y. Zhu, *Inter-Disciplinary Research Challenges in Computer Systems for the 2020s*, USA, 2018.

[16] J. Coplin, A. Yang, A. Poppe, and M. Burtscher, *Increasing Telemetry Throughput Using Customized and Adaptive Data Compression*, in AIAA SPACE and Astronautics Forum and Exposition, 2016.

[17] J. Davison de St. Germain, J. McCorquodale, S. Parker, and C. Johnson, *Uintah: a massively parallel problem solving environment*, in Proceedings the Ninth International Symposium on High-Performance Distributed Computing, 2000, pp. 33–41.

[18] D. de Oliveira, A. Humphrey, Q. Meng, Z. Rakamaric, M. Berzins, and G. Gopalakrishnan, *Systematic debugging of concurrent systems using coalesced stack trace graphs*, in Proceedings of the 27th International Workshop on Languages and Compilers for Parallel Computing (LCPC), September 2014.

[19] N. Dilley and J. Lange, *An empirical study of messaging passing concurrency in go projects*, in 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019, pp. 377–387.

[20] N. Dilley and J. Lange, *Bounded verification of message-passing concurrency in go using promela and spin*, Electronic Proceedings in Theoretical Computer Science, 314 (2020), p. 34–45.

[21] O. Edelstein, E. Farchi, E. Goldin, Y. Nir-Buchbinder, G. Ratsaby, and S. Ur, *Framework for testing multithreaded java programs*, Concurrency and Computation: Practice and Experience, 15 (2003).

[22] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, *Multithreaded java program test generation*, in Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande, JGI '01, New York, NY, USA, 2001, Association for Computing Machinery, p. 181.

[23] M. Emmi, S. Qadeer, and Z. Rakamarić, *Delay-bounded scheduling*, in Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11, New York, NY, USA, 2011, Association for Computing Machinery, p. 411–422.

[24] C. Flanagan and P. Godefroid, *Dynamic partial-order reduction for model checking software*, in Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '05, New York, NY, USA, 2005, Association for Computing Machinery, p. 110–121.

[25] E. B. Fowlkes and C. L. Mallows, *A method for comparing two hierarchical clusterings*, Journal of the American Statistical Association, 78 (1983), pp. 553–569.

[26] F. Freitag, J. Caubet, and J. Labarta, *On the Scalability of Tracing Mechanisms*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 97–104.

[27] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed, *Scalable load-balance measurement for spmd codes*, in SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Nov 2008, pp. 1–12.

[28] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st ed., 1997.

[29] Q. Gao, F. Qin, and D. K. Panda, *Dmtracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements*, in SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, Nov 2007, pp. 1–12.

[30] V. K. Garg, *Maximal antichain lattice algorithms for distributed computations*, in Distributed Computing and Networking, D. Frey, M. Raynal, S. Sarkar, R. K. Shyamasundar, and P. Sinha, eds., Berlin, Heidelberg, 2013, Springer Berlin Heidelberg, pp. 240–254.

[31] ——, *Introduction to lattice theory with computer science applications*, Wiley, 2015.

[32] R. Godin, R. Missaoui, and H. Alaoui, *Incremental concept formation algorithms based on galois (concept) lattices*, Computational Intelligence, 11, pp. 246–267.

[33] Golang, *Command Trace*.

[34] G. Gopalakrishnan, P. D. Hovland, C. Iancu, S. Krishnamoorthy, I. Laguna, R. A. Lethin, K. Sen, S. F. Siegel, and A. Solar-Lezama, *Report of the HPC correctness summit, jan 25-26, 2017, washington, DC*, CoRR, abs/1705.07478 (2017).

[35] C. Gottbrath and P. Thompson, *Totalview tips and tricks*, in Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06, New York, NY, USA, 2006, ACM.

[36] K. Hazelwood and A. Klauser, *A dynamic binary instrumentation engine for the arm architecture*, in Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES '06, New York, NY, USA, 2006, ACM, pp. 261–270.

[37] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, *Improving performance via mini-applications*, Sandia National Laboratories, Tech. Rep. SAND2009-5574, 3 (2009).

[38] C. A. R. Hoare, *Communicating sequential processes*, Commun. ACM, 21 (1978), p. 666–677.

[39] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold, *Testing concurrent programs to achieve high synchronization coverage*, in Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, New York, NY, USA, 2012, Association for Computing Machinery, p. 210–220.

[40] https://blog.golang.org/survey2019-results, *Go Developer Survey 2019 Results*.

[41] https://github.com/coreos/etcd, *ETCD: A distributed, reliable key-value store for the most critical data of a distributed system*.

[42] https://github.com/grpc/grpc-go, *A high performance, open source, general RPC framework that puts mobile and HTTP/2 first*.

[43] https://github.com/moby/moby/issues/28405, *moby-28462-commit*.

[44] https://github.com/sasha-s/go-deadlock, *Online deadlock detection in go (golang)*.

[45] https://github.com/uber-go/goleak, *Uber goleak*.

[46] https://golang.org/doc/effective_go.html, *Effective Go*.

[47] https://golang.org/pkg/go/ast/, *Package AST*.

[48] https://golang.org/pkg/runtime/trace/, *Package Trace*.

[49] https://golang.org/ref/mem, *The Go Memory Model*, 2014.

[50] https://golang.org/src/internal/trace/parser.go, *Package Parser*.

[51] https://kubernetes.io/docs/reference/, *Kubernetes Reference*.

[52] D. I. Ignatov, *Introduction to formal concept analysis and its applications in information retrieval and related fields*, CoRR, abs/1703.02819 (2017).

[53] Intel, *Pin, A Dynamic Binary Instrumentation*.

[54] D. Johnson and L. A. McGeoch, *The traveling salesman problem: A case study in local optimization*, Local Search in Combinatorial Optimization, 1 (1997).

[55] E. Jones, T. Oliphant, P. Peterson, et al., *SciPy: Open source scientific tools for Python*, 2001–. [Online; accessed ¡today¿].

[56] P. Joshi, M. Naik, C.-S. Park, and K. Sen, *Calfuzzer: An extensible active testing framework for concurrent programs*, in Computer Aided Verification, A. Bouajjani and O. Maler, eds., Berlin, Heidelberg, 2009, Springer Berlin Heidelberg, pp. 675–681.

[57] M. Jurenz, R. Brendel, A. Knupfer, M. Muller, and W. E. Nagel, *Memory allocation tracing with vampirtrace*, in Proceedings of the 7th International Conference on Computational Science Part II, ICCS '07, Berlin, Heidelberg, 2007, Springer-Verlag, pp. 839–846.

[58] I. Karlin, J. Keasler, and R. Neely, *LULESH 2.0 Updates and Changes*, August 2013.

[59] R. M. Karp, R. E. Miller, and A. L. Rosenberg, *Rapid identification of repeated patterns in strings, trees and arrays*, in Proceedings of the Fourth Annual ACM Symposium on Theory of Computing, STOC '72, New York, NY, USA, 1972, ACM, pp. 125–136.

[60] A. Ketterlin and P. Clauss, *Prediction and trace compression of data access addresses through nested loop recognition*, in Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '08, New York, NY, USA, 2008, ACM, pp. 94–103.

[61] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, *Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir*, in Tools for High Performance Computing 2011 - Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, ZIH, Dresden, September 2011, 2011, pp. 79–91.

[62] M. Kobayashi and M. MacDougall, *Dynamic characteristics of loops*, IEEE Transactions on Computers, C-33 (1984), pp. 125–132.

[63] S. O. Kuznetsov and S. A. Obiedkov, *Comparing performance of algorithms for generating concept lattices*, Journal of Experimental & Theoretical Artificial Intelligence, 14 (2002), pp. 189–216.

[64] I. Laguna, T. Gamblin, B. R. de Supinski, S. Bagchi, G. Bronevetsky, D. H. Anh, M. Schulz, and B. Rountree, *Large scale debugging of parallel tasks with automaded*, in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, New York, NY, USA, 2011, ACM, pp. 50:1–50:10.

[65] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, Commun. ACM, 21 (1978), pp. 558–565.

[66] J. Lange, N. Ng, B. Toninho, and N. Yoshida, *Fencing off go: Liveness and safety for channel-based programming*, in Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, New York, NY, USA, 2017, Association for Computing Machinery, p. 748–761.

[67] ——, *A static verification framework for message passing in go using behavioural types*, in Proceedings of the 40th International Conference on Software Engineering, ICSE '18, New York, NY, USA, 2018, Association for Computing Machinery, p. 1137–1148.

[68] B. Liu and J. Huang, *D4: Fast concurrency debugging with parallel differential analysis*, in Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, New York, NY, USA, 2018, ACM, pp. 359–373.

[69] S. Lu, S. Park, E. Seo, and Y. Zhou, *Learning from mistakes: A comprehensive study on real world concurrency bug characteristics*, in Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, New York, NY, USA, 2008, Association for Computing Machinery, p. 329–339.

[70] 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, PLDI '05, New York, NY, USA, 2005, ACM, pp. 190–200.

[71] D. Merkel, *Docker: lightweight linux containers for consistent development and deployment*, Linux journal, 2014 (2014), p. 2.

[72] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, *The paradyn parallel performance measurement tool*, IEEE Computer, 28 (1995), pp. 37–46.

[73] S. Mitra, I. Laguna, D. H. Ahn, S. Bagchi, M. Schulz, and T. Gamblin, *Accurate application progress analysis for large-scale parallel debugging*, in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, New York, NY, USA, 2014, ACM, pp. 193–203.

[74] K. Mohror and K. L. Karavanic, *Evaluating similarity-based trace reduction techniques for scalable performance analysis*, in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, New York, NY, USA, 2009, ACM, pp. 55:1–55:12.

[75] M. Musuvathi and S. Qadeer, *Iterative context bounding for systematic testing of multithreaded programs*, in Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07, New York, NY, USA, 2007, Association for Computing Machinery, p. 446–455.

[76] E. W. Myers, *An O(ND) difference algorithm and its variations*, Algorithmica, 1 (1986), pp. 251–266.

[77] A. Nakamura, T. Saito, I. Takigawa, M. Kudo, and H. Mamitsuka, *Fast algorithms for finding a minimum repetition representation of strings and trees*, Discrete Applied Mathematics, 161 (2013), pp. 1556 – 1575.

[78] A. Nataraj, A. Malony, A. Morris, D. C. Arnold, and B. Miller, *A framework for scalable, parallel performance monitoring*, 22 (2009), pp. 720–735.

[79] N. Nethercote and J. Seward, *Valgrind: A program supervision framework*, Electr. Notes Theor. Comput. Sci., 89 (2003), pp. 44–66.

[80] N. Nethercote and J. Seward, *How to shadow every byte of memory used by a program*, in Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE '07, New York, NY, USA, 2007, ACM, pp. 65–74.

[81] M. D. Network, *C Sequence Points*.

[82] N. Ng and N. Yoshida, *Static deadlock detection for concurrent go by global session graph synthesis*, in Proceedings of the 25th International Conference on Compiler Construction, CC 2016, New York, NY, USA, 2016, Association for Computing Machinery, p. 174–184.

[83] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski, *Scalatrace: Scalable compression and replay of communication traces for high-performance computing*, Journal of Parallel and Distributed Computing, 69 (2009), pp. 696 – 710. Best Paper Awards: 21st International Parallel and Distributed Processing Symposium (IPDPS 2007).

[84] R. Preissl, T. Köckerbauer, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan, *Detecting patterns in mpi communication traces*, 2008 37th International Conference on Parallel Processing, (2008), pp. 230–237.

[85] P. Ratanaworabhan and M. Burtscher, *Program phase detection based on critical basic block transitions*, in ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software, April 2008, pp. 11–21.

[86] L. D. Rose, A. Gontarek, A. Vose, R. Moench, D. Abramson, M. N. Dinh, and C. Jin, *Relative debugging for a highly parallel hybrid computer system*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015, 2015, pp. 63:1–63:12.

[87] P. C. Roth, D. C. Arnold, and B. P. Miller, *Mrnet: A software-based multicast/reduction network for scalable tools*, in Supercomputing, 2003 ACM/IEEE Conference, Nov 2003, pp. 21–21.

[88] P. C. Roth, J. S. Meredith, and J. S. Vetter, *Automated characterization of parallel application communication patterns*, in Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '15, New York, NY, USA, 2015, ACM, pp. 73–84.

[89] S. M. Russ Cox, *Profiling Go Programs*, 2013.

[90] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford, *Open | speedshop: An open source infrastructure for parallel performance analysis*, Scientific Programming, 16 (2008), pp. 105–121.

[91] K. Serebryany and T. Iskhodzhanov, *Threadsanitizer: Data race detection in practice*, in Proceedings of the Workshop on Binary Instrumentation and Applications, WBIA '09, New York, NY, USA, 2009, Association for Computing Machinery, p. 62–71.

[92] S. S. Shende and A. D. Malony, *The Tau parallel performance system*, International Journal on High Performance Computer Applications, 20 (2006), pp. 287–311.

[93] K. Stadtmüller, M. Sulzmann, and P. Thiemann, *Static trace-based deadlock analysis for synchronous mini-go*, in Programming Languages and Systems, A. Igarashi, ed., Cham, 2016, Springer International Publishing, pp. 116–136.

[94] S. M. Strande, H. Cai, T. Cooper, K. Flammer, C. Irving, G. von Laszewski, A. Majumdar, D. Mishin, P. Papadopoulos, W. Pfeiffer, R. S. Sinkovits, M. Tatineni, R. Wagner, F. Wang, N. Wilkins-Diehr, N. Wolter, and M. L. Norman, *Comet: Tales from the long tail: Two years in and 10,000 users later*, in Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact, PEARC17, New York, NY, USA, 2017, ACM, pp. 38:1–38:7.

[95] M. Sulzmann and K. Stadtmüller, *Trace-based run-time analysis of message-passing go programs*, CoRR, abs/1709.01588 (2017).

[96] M. Sulzmann and K. Stadtmüller, *Two-phase dynamic analysis of message-passing go programs based on vector clocks*, PPDP '18, New York, NY, USA, 2018, Association for Computing Machinery.

[97] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis, *Cockroachdb: The resilient geo-distributed sql database*, in Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, New York, NY, USA, 2020, Association for Computing Machinery, p. 1493–1509.

[98] S. Taheri, I. Briggs, M. Burtscher, and G. Gopalakrishnan, *Difftrace: Efficient whole-program trace analysis and diffing for debugging*, in 2019 IEEE International Conference on Cluster Computing (CLUSTER), 2019, pp. 1–12.

[99] S. Taheri, S. Devale, G. Gopalakrishnan, and M. Burtscher, *ParLOT: Efficient whole-program call tracing for HPC applications*, in Programming and Performance Visualization Tools - International Workshops, ESPT 2017 and VPA 2017, Denver, CO, USA, November 12 and 17, 2017, and ESPT 2018 and VPA 2018, Dallas, TX, USA, November 16 and 11, 2018, Revised Selected Papers, 2018, pp. 162–184.

[100] S. Taheri and G. Gopalakrishnan, *Automated Dynamic Concurrency Analysis for Go*, 2021.

[101] P. Thomson, A. F. Donaldson, and A. Betts, *Concurrency testing using schedule bounding: An empirical study*, in Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, New York, NY, USA, 2014, Association for Computing Machinery, p. 15–28.

[102] M. M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely, *A.: Pmac binary instrumentation library for powerpc/aix*, in In: Workshop on Binary Instrumentation and Applications, 2006.

[103] E. Trainin, Y. Nir-Buchbinder, R. Tzoref-Brill, A. Zlotnick, S. Ur, and E. Farchi, *Forcing small models of conditions on program interleaving for detection of*

concurrent bugs, in Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD '09, New York, NY, USA, 2009, Association for Computing Machinery.

[104] T. Tu, X. Liu, L. Song, and Y. Zhang, *Understanding real-world concurrency bugs in go*, in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, New York, NY, USA, 2019, Association for Computing Machinery, p. 865–878.

[105] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. Van Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. Peltz Jr., T. Peterka, M. Strout, and J. Wilke, *Extreme heterogeneity 2018 - productive computational science in the era of extreme heterogeneity: Report for doe ascr workshop on extreme heterogeneity*, (2018).

[106] D. Vyukov and A. Gerrand, *Introducing the Go Race Detector*, 2013.

[107] M. Weber, R. Brendel, T. Hilbrich, K. Mohror, M. Schulz, and H. Brunst, *Structural Clustering: A New Approach to Support Performance Analysis at Scale*, IEEE, May 2016, pp. 484–493.

[108] J. Weidendorfer, *Sequential performance analysis with callgrind and kcachegrind*, in Tools for High Performance Computing - Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart, 2008, pp. 93–113.

[109] A. Yang, H. Mukka, F. Hesaaraki, and M. Burtscher, *MPC: A massively parallel compression algorithm for scientific data*, in 2015 IEEE International Conference on Cluster Computing, Sept 2015, pp. 381–389.

[110] J. Yu and S. Narayanasamy, *A case for an interleaving constrained shared-memory multi-processor*, in Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA '09, New York, NY, USA, 2009, Association for Computing Machinery, p. 325–336.

[111] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, *Maple: A coverage-driven testing tool for multithreaded programs*, OOPSLA '12, New York, NY, USA, 2012, Association for Computing Machinery, p. 485–502.

[112] T. Yuan, G. Li, J. Lu, C. Liu, L. Li, and J. Xue, *Gobench: A benchmark suite of real-world go concurrency bugs*, in 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2021, pp. 187–199.

[113] X. Yuan and J. Yang, *Effective concurrency testing for distributed systems*, ASPLOS '20, New York, NY, USA, 2020, Association for Computing Machinery, p. 1141–1156.

[114] A. Zeller, *Yesterday, my program worked. today, it does not. why?*, in Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings, 1999, pp. 253–267.

[115] J. Zhao, H. Abe, Y. Nomura, J. Cheng, and K. Ushijima, *Runtime detection of communication deadlocks in occam 2 programs*, (1997), pp. 97–107.

[116] J. Ziv and A. Lempel, *A universal algorithm for sequential data compression*, IEEE Trans. Inf. Theor., 23 (2006), pp. 337–343.