ParLoT: Efficient Whole-Program Call Tracing for HPC Applications

Saeed Taheri[1], Sindhu Devale[2], Ganesh Gopalakrishnan[1], and Martin Burtscher[2]

[1] School of Computing, University of Utah, Salt Lake City, Utah, USA
{staheri,ganesh}@cs.utah.edu
[2] Department of Computer Science, Texas State University, San Marcos, Texas, USA
sindhu.devale@gmail.com
burtscher@cs.txstate.edu

Abstract. The complexity of HPC software and hardware is quickly increasing. As a consequence, the need for efficient execution tracing to gain insight into HPC application behavior is steadily growing. Unfortunately, available tools either do not produce traces with enough detail or incur large overheads. An efficient tracing method that overcomes the tradeoff between maximum information and minimum overhead is therefore urgently needed. This paper presents such a method and tool, called ParLoT, with the 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. This paper 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.

Keywords: Tracing, HPC, data compression, incremental compression

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 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 [13].

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 [13]. 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, 27]. 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 [27], 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 [28], 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 "Xray" 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 [25]. 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 [5], 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 [11] [12]. 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 introduces the basic ideas and infrastructure behind ParLoT and other tracing tools. Section 3 describes the design of ParLoT in detail. Sections 4 and 5 present our evaluation of different aspects of ParLoT and compare it with a similar tool. Section 6 concludes the paper with a summary and future work.

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.

2.1 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 [20] provides a dynamic instrumentation API that gives developers the ability to measure various performance aspects. It is used in tools like Open-SpeedShop [30] and TAU [31] as well as correctness debuggers like STAT [3]. Moreover, VampirTrace [17] uses it to provide a library for collecting program execution logs.

Valgrind [24] 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 [23] and call-graph generators like CALLGRIND [34] are built upon Valgrind.[3]

We implemented PARLOT on top of PIN [19], 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 [14]. 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 [33] is a DBI tool for the PowerPC/AIX architecture upon which PARLOT could also be based.

[3] Given the absence of tools similar to PARLOT, we employ CALLGRIND as a "close-enough" tool in our comparisons elaborated in §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 §5, we also present a self-assessment of PARLOT separately.

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 [31, 18, 1] already employ compression during the collection of performance analysis data. Tools such as ScalaTrace [26] also exploit the repetitive nature of time-step simulations [9]. Aguilar et al. [2] proposed a lossy compression mechanism using the Nami library [10] for online MPI tracing. Mohror and Karavanic [21] investigated similarity-based trace reduction techniques to store and analyze traces at scale.

Many performance and debugging tools for HPC applications [3, 22] rely on mechanisms such as MRNet [29] 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 §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.

3 Design of ParLoT

Our experimental results in §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

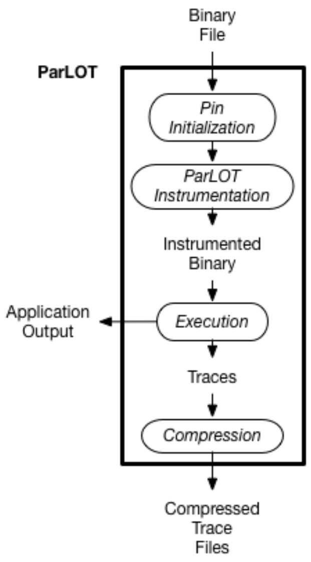

Fig. 1. Overview of ParLoT

without compression when running the Mantevo miniapps [15] 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 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.

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.

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 §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 §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.

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.

3.3 Compression Algorithm

We used the CRUSHER framework [35][8][6][7] 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 [15] 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 [36]. 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.

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 [16], 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.

4 Evaluation Methodology

4.1 Benchmarks and System

We performed our evaluations on the MPI-based NAS Parallel Benchmarks (NPB) [4]. 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 [32], 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.

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.

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 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.

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.

Table 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**

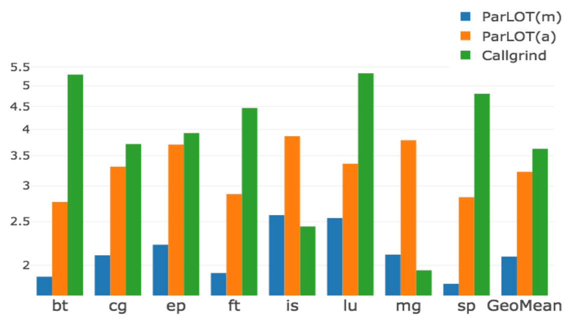

Fig. 2. Average tracing overhead on the NPB applications - Input B

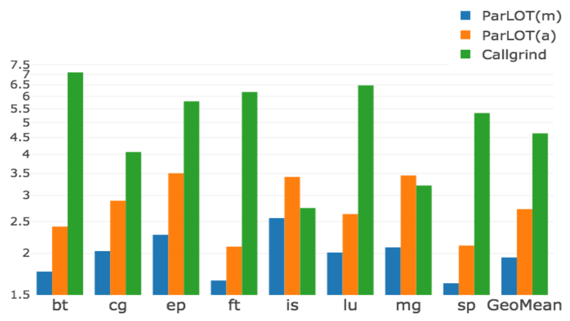

Fig. 3. Average tracing overhead on the NPB applications - Input C

5 Results

5.1 Tracing Overhead

Table 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 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 and 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 debug their applications. This is promising especially when considering how detailed the collected trace information is and that most of the

Table 2. Required bandwidth per core (kB/s)

Input	Tool	# Nodes	bt	cg	ep	ft	is	lu	mg	sp	GM
B	ParLoT(m)	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
		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**
	ParLoT(a)	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
		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**
	Callgrind	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
		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**
C	ParLoT(m)	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
		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**
	ParLoT(a)	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
		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**
	Callgrind	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
		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**

overhead is due to PIN (see §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.

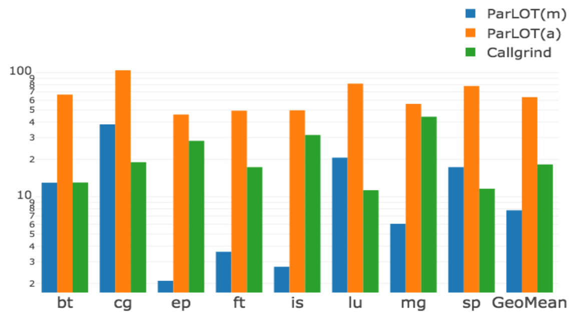

Fig. 4. Average required bandwidth per core (kB/s) on the NPB applications - Input B

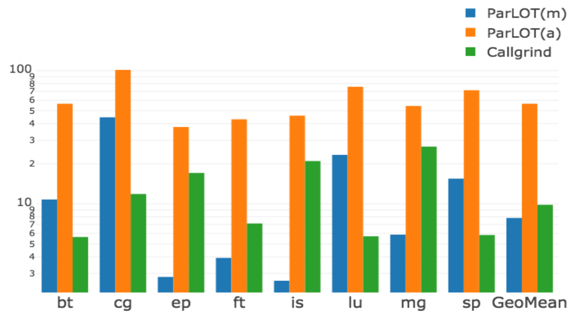

Fig. 5. Average required bandwidth per core (kB/s) on the NPB applications - Input C

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.

5.2 Required Bandwidth

Table 2, Fig. 4 and Fig. 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

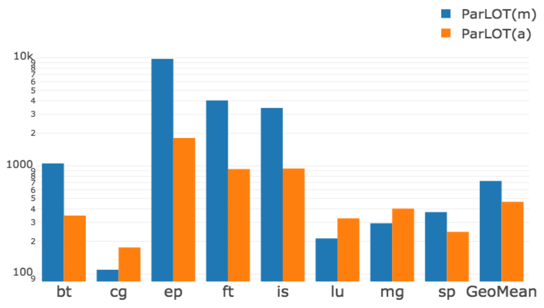

Fig. 6. Average compression ratio of ParLoT on the NPB applications - Input B

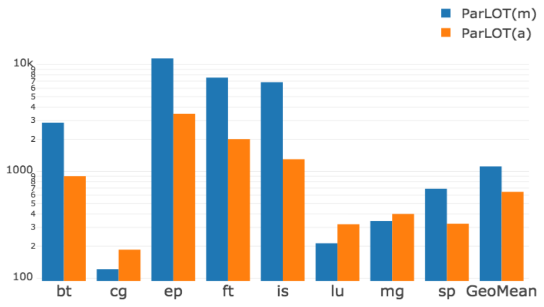

Fig. 7. Average compression ratio of ParLoT on the NPB applications - Input C

the required bandwidth of ParLoT is independent of the problem size, unlike Callgrind, where the input size has a linear impact on the results.

5.3 Compression Ratio

Table 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 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 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 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. 4, Fig. 5, Fig. 6 and Fig. 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).

Table 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**

5.4 Overheads

Table 4 presents the average overhead added to each application for different versions of ParLoT. Last rows of each section of this table present the geometric mean. This information captures how much each phase of ParLoT slows down the native execution.

Table 4. Tracing overhead of versions of ParLoT- Input B

Input: B	Nodes :	1			4			16			64		
	Detail Tools:	Pin-Init	ParLoT	ParLoT-NC	Pin-Init	ParLoT	ParLoT-NC	Pin-Init	ParLoT	ParLoT-NC	Pin-Init	ParLoT	ParLoT-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
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

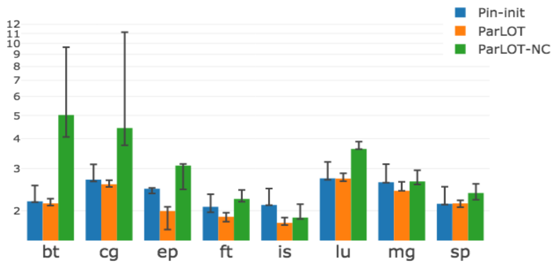

Fig. 8. Variability of ParLoT(m) overhead on 16 nodes - Input B

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. 8 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 Table 4.

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. 11 and 12. 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.

5.5 Compression Impact

Fig. 9 and Fig. 10 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 4 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.

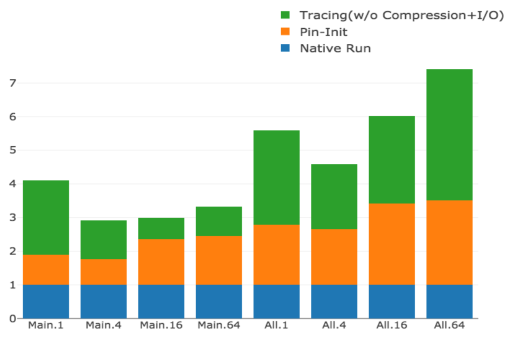

Fig. 9. ParLoT-NC tracing overhead breakdown - Input B

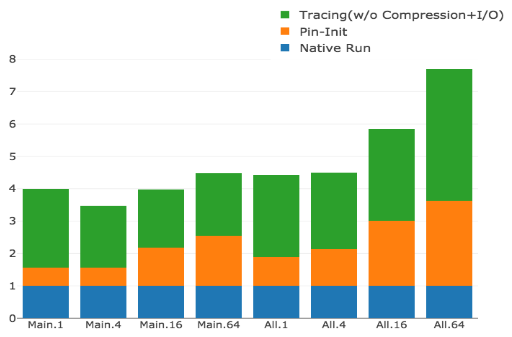

Fig. 10. ParLoT-NC tracing overhead breakdown - Input C

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.

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

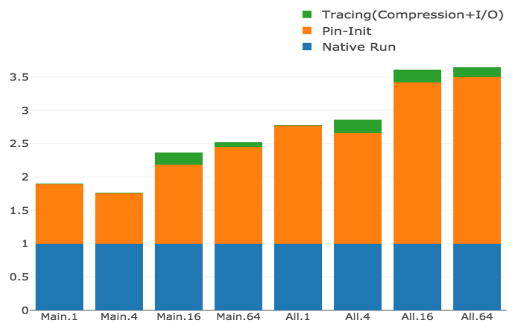

Fig. 11. Tracing overhead breakdown - Input B

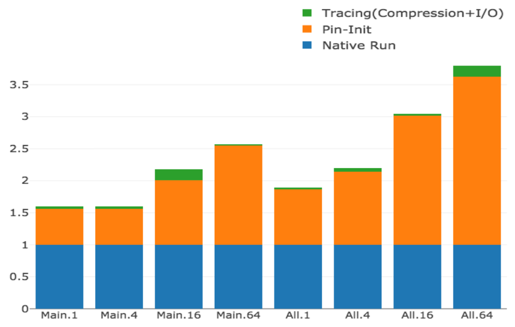

Fig. 12. Tracing overhead breakdown - Input C

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.

References

1. Aguilar, X., Frlinger, K., Laure, E.: Online mpi trace compression using event flow graphs and wavelets. Procedia Computer Science **80**(Supplement C), 1497 – 1506 (2016). https://doi.org/https://doi.org/10.1016/j.procs.2016.05.471, international Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA
2. Aguilar, X., Frlinger, K., Laure, E.: Online mpi trace compression using event flow graphs and wavelets. Procedia Computer Science **80**, 1497 – 1506 (2016). https://doi.org/https://doi.org/10.1016/j.procs.2016.05.471, http://www.sciencedirect.com/science/article/pii/S1877050916309565, international Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA
3. Arnold, D.C., Ahn, D.H., de Supinski, B.R., Lee, G.L., Miller, B.P., Schulz, M.: Stack trace analysis for large scale debugging. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS). pp. 1–10 (2007)
4. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The nas parallel benchmarks&mdash;summary and preliminary results. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing. pp. 158–165. Supercomputing '91, ACM, New York, NY, USA (1991). https://doi.org/10.1145/125826.125925
5. Burtscher, M., Rabeti, H.: A scalable heterogeneous parallelization framework for iterative local searches. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. pp. 1289–1298 (May 2013). https://doi.org/10.1109/IPDPS.2013.27
6. Burtscher, M., Mukka, H., Yang, A., Hesaaraki, F.: Real-time synthesis of compression algorithms for scientific data. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 23:1–23:12. SC '16, IEEE Press, Piscataway, NJ, USA (2016), http://dl.acm.org/citation.cfm?id=3014904.3014935
7. Claggett, S., Azimi, S., Burtscher, M.: SPDP: An Automatically Synthesized Lossless Compression Algorithm for Floating-Point Data. In: 2018 Data Compression Conference (2018)
8. Coplin, J., Yang, A., Poppe, A., Burtscher, M.: Increasing Telemetry Throughput Using Customized and Adaptive Data Compression. In: AIAA SPACE and Astronautics Forum and Exposition (2016)

9. Freitag, F., Caubet, J., Labarta, J.: On the Scalability of Tracing Mechanisms, pp. 97–104. Springer Berlin Heidelberg, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45706-2-10
10. Gamblin, T., de Supinski, B.R., Schulz, M., Fowler, R., Reed, D.A.: Scalable load-balance measurement for spmd codes. In: SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. pp. 1–12 (Nov 2008). https://doi.org/10.1109/SC.2008.5222553
11. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edn. (1997)
12. Godin, R., Missaoui, R., Alaoui, H.: Incremental concept formation algorithms based on galois (concept) lattices. Computational Intelligence **11**(2), 246–267
13. Gopalakrishnan, G., Hovland, P.D., Iancu, C., Krishnamoorthy, S., Laguna, I., Lethin, R.A., Sen, K., Siegel, S.F., Solar-Lezama, A.: Report of the HPC correctness summit, jan 25-26, 2017, washington, DC. CoRR **abs/1705.07478** (2017), http://arxiv.org/abs/1705.07478
14. Hazelwood, K., Klauser, A.: A dynamic binary instrumentation engine for the arm architecture. In: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems. pp. 261–270. CASES '06, ACM, New York, NY, USA (2006). https://doi.org/10.1145/1176760.1176793
15. Heroux, M.A., Doerfler, D.W., Crozier, P.S., Willenbring, J.M., Edwards, H.C., Williams, A., Rajan, M., Keiter, E.R., Thornquist, H.K., Numrich, R.W.: Improving performance via mini-applications. Sandia National Laboratories, Tech. Rep. SAND2009-5574 **3** (2009)
16. Intel: Pin, a dynamic binary instrumentation, https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
17. Jurenz, M., Brendel, R., Knupfer, A., Muller, M., Nagel, W.E.: Memory allocation tracing with vampirtrace. In: Proceedings of the 7th International Conference on Computational Science Part II. pp. 839–846. ICCS '07, Springer-Verlag, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72586-2-118
18. Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A.D., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: 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. pp. 79–91 (2011)
19. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 190–200. PLDI '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1065010.1065034
20. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The paradyn parallel performance measurement tool. IEEE Computer **28**(11), 37–46 (1995). https://doi.org/10.1109/2.471178, https://doi.org/10.1109/2.471178
21. Mohror, K., Karavanic, K.L.: Evaluating similarity-based trace reduction techniques for scalable performance analysis. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. pp. 55:1–55:12. SC '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1654059.1654115, http://doi.acm.org/10.1145/1654059.1654115

22. Nataraj, A., Malony, A., Morris, A., C. Arnold, D., Miller, B.: A framework for scalable, parallel performance monitoring **22**, 720–735 (01 2009)
23. Nethercote, N., Seward, J.: How to shadow every byte of memory used by a program. In: Proceedings of the 3rd International Conference on Virtual Execution Environments. pp. 65–74. VEE '07, ACM, New York, NY, USA (2007)
24. Nethercote, N., Seward, J.: Valgrind: A program supervision framework. Electr. Notes Theor. Comput. Sci. **89**(2), 44–66 (2003). https://doi.org/10.1016/S1571-0661(04)81042-9, https://doi.org/10.1016/S1571-0661(04)81042-9
25. Network, M.D.: C sequence points, https://msdn.microsoft.com/en-us/library/azk8zbxd.aspx
26. Noeth, M., Ratn, P., Mueller, F., Schulz, M., de Supinski, B.R.: Scalatrace: Scalable compression and replay of communication traces for high-performance computing. Journal of Parallel and Distributed Computing **69**(8), 696 – 710 (2009). https://doi.org/https://doi.org/10.1016/j.jpdc.2008.09.001, best Paper Awards: 21st International Parallel and Distributed Processing Symposium (IPDPS 2007)
27. de Oliveira, D., Humphrey, A., Meng, Q., Rakamaric, Z., Berzins, M., Gopalakrishnan, G.: 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), http://www.sci.utah.edu/publications/Oli2014a/OliveiraLCPC2014.pdf
28. Ratanaworabhan, P., Burtscher, M.: Program phase detection based on critical basic block transitions. In: ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software. pp. 11–21 (April 2008). https://doi.org/10.1109/ISPASS.2008.4510734
29. Roth, P.C., Arnold, D.C., Miller, B.P.: Mrnet: A software-based multicast/reduction network for scalable tools. In: Supercomputing, 2003 ACM/IEEE Conference. pp. 21–21 (Nov 2003). https://doi.org/10.1145/1048935.1050172
30. Schulz, M., Galarowicz, J., Maghrak, D., Hachfeld, W., Montoya, D., Cranford, S.: Open | speedshop: An open source infrastructure for parallel performance analysis. Scientific Programming **16**(2-3), 105–121 (2008). https://doi.org/10.3233/SPR-2008-0256, https://doi.org/10.3233/SPR-2008-0256
31. Shende, S.S., Malony, A.D.: The Tau parallel performance system. International Journal on High Performance Computer Applications **20**, 287–311 (May 2006). https://doi.org/10.1177/1094342006064482, http://portal.acm.org/citation.cfm?id=1125980.1125982
32. Strande, S.M., Cai, H., Cooper, T., Flammer, K., Irving, C., von Laszewski, G., Majumdar, A., Mishin, D., Papadopoulos, P., Pfeiffer, W., Sinkovits, R.S., Tatineni, M., Wagner, R., Wang, F., Wilkins-Diehr, N., Wolter, N., Norman, M.L.: 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. pp. 38:1–38:7. PEARC17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3093338.3093383, http://doi.acm.org/10.1145/3093338.3093383
33. Tikir, M.M., Laurenzano, M., Carrington, L., Snavely, A.: A.: Pmac binary instrumentation library for powerpc/aix. In: In: Workshop on Binary Instrumentation and Applications (2006)
34. Weidendorfer, J.: 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. pp. 93–113 (2008). https://doi.org/10.1007/978-3-540-68564-7-7

35. Yang, A., Mukka, H., Hesaaraki, F., Burtscher, M.: MPC: A massively parallel compression algorithm for scientific data. In: 2015 IEEE International Conference on Cluster Computing. pp. 381–389 (Sept 2015). https://doi.org/10.1109/CLUSTER.2015.59
36. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theor. **23**(3), 337–343 (Sep 2006). https://doi.org/10.1109/TIT.1977.1055714