LOW-OVERHEAD TRACING OF LARGE-SCALE PARALLEL PROGRAMS

by

Sindhu Devale

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
May 2016

Committee Members:

Martin Burtscher, Chair

Apan Qasem

Ziliang Zong

DEDICATION

To my mentor, I couldn’t have done this without you, thank you for all of your support and guidance along the way. To my husband, who has been so supportive and has been my backbone throughout the journey, I cannot thank him enough. To my parents, thank you for always giving me the moral support.

ACKNOWLEDGEMENTS

This project is supported by the National Science Foundation.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Some parallelization bugs only manifest themselves when a program is executed at scale. Such bugs are notoriously difficult to find, and tracing parallel programs at scale tends to be very expensive both in terms of execution overhead and in terms of the amount of trace data generated. To make light-weight debugging possible on large-scale systems, I present and evaluate a scalable profiling tool called RTC-Tracer that incrementally compresses the gathered information before it is written to memory or disk. For example, RTC-Tracer can track every function call and return of the Mantevo miniapps running on Stampede with a 1.73 to 2.31x overhead in execution time on average while compressing the collected information by a factor of 100, resulting in only a few kilobytes per second of trace data being emitted by each processor.

CHAPTER 1

INTRODUCTION

Tracing is a very useful and powerful method for analyzing the performance of programs and for identifying bugs. In fact, software engineers already invest nearly 70% of the development time of a project in debugging mostly serial code [1]. With the rapidly increasing amounts of data being generated, there is a growing need to scale the number of cores to process them. Parallel frameworks have thus become central to advancements in various fields of science and engineering. Unfortunately, the more cores a program uses, the harder, more expensive, and more time consuming it becomes to debug the code, and buggy code can drastically reduce the productivity. It is estimated that the US loses $60 billion every year due to software glitches [1].

1.1. Large traces

Most of the existing debugging techniques perform poorly when scaled. One of the reasons is the overhead in collecting large amounts of runtime information, which increases in proportion with the number of running processes and/or threads. Large trace sizes often constrain the scalability on large-scale systems and complicate the analysis and visualization of the trace data. What information to extract from the programs, how to best extract it, and how to keep the overhead of run-time and memory low are additional challenges.

To record traces, the application is usually instrumented, i.e., extra code is added at various points to intercept the desired events. The trace records are kept in a memory buffer and written to a file after program termination or upon buffer overflow.

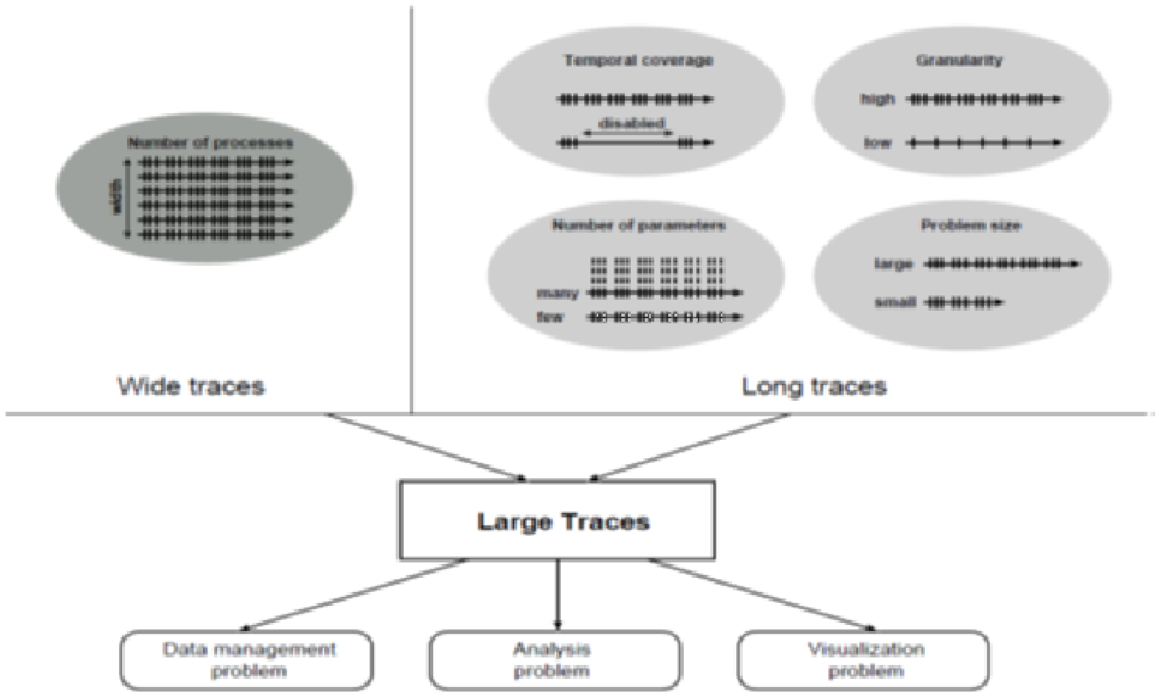

Figure 1: Reasons for large trace sizes and the problems they create

The reasons for large trace sizes [2] as shown in Figure 1 are explained in detail below.

- **Number of processes or threads:** Since this number is equal to the number of time-lines in a time-line diagram, it is often referred to as the *width* of an event trace (as opposed to the *length*, which represents the number of events per process or thread). Because the total number of gathered events usually grows with the number of processes or threads, the width influences both the total amount of data as well as the total number of local trace files that need to be handled.

- **Temporal coverage:** The intervals to be traced need not cover the entire execution. It is obvious that restricting tracing to smaller intervals can substantially decrease the amount of trace data, but may also result in loss of fidelity.

- **Granularity:** How many events are recorded during a given interval depends on the frequency at which events are generated. This is typically related to the granularity of measurements, that is, the level of detail (e.g., function, block, or statement level) captured through tracing.

- **Number of event parameters:** The number of parameters recorded directly affects the trace size. Hence, the number of parameters rarely exceeds a few.

- **Problem size:** This factor considers the number of performance-relevant events as a result of the input applied to a certain algorithm. A typical example is the number of iterations performed to arrive at a solution, which can prolong execution and increase the number of events traced.

As a result, there are data management problems to store huge traces. Moreover, analyzing huge traces is a problem and cannot be done in a time efficient manner. Due to these problems, we need a low-overhead tracing mechanism that gathers as much information as possible at as low of a cost as possible.

For my research, I have chosen a set of MPI and OpenMP applications to evaluate my approach. MPI is widely used for distributed-memory programming (e.g., clusters) and OpenMP is used for shared-memory programming (e.g., multicore systems).

1.2. Contributions

This thesis makes the following contributions:

1. A user-friendly, portable tool called RTC-Tracer to extract function call and return information from parallel programs.
2. The tool is efficient both in terms of runtime and in terms of storage utilization.
3. The tool incorporates a custom algorithm to incrementally compress the generated traces at runtime.

1.3. Results

RTC-Tracer works well for large-scale parallel programs. Its overhead is a factor of about 1.73 to 2.31. The compression algorithm it incorporates often compresses better than standard compression algorithms while, at the same time, compressing the data much more quickly. The resulting bandwidth requirement per core is only a few kilobytes of data emitted per second compared to megabytes of data without compression.

1.4. Outline

The rest of this thesis is organized as follows: Chapter 2 presents the background. Chapter 3 summarizes related work. Chapter 4 describes the design of the RTC-Tracer tool. Chapter 5 presents the evaluation methodology and the testbed. Chapter 6 studies and analyzes the performance results. Chapter 7 concludes with a summary and future work.

CHAPTER 2

BACKGROUND

2.1. Tracing

In software engineering, tracing [3] is a specialized use of logging to record information about a program's execution. Since software tracing is often low-level, the possible volume of trace information is high.

Phases of instrumentation tracing [4]:

a. Instrumentation – adding trace code to the application.
b. Tracing – the tracing code executes together with the application code and writes information to the disk.
c. Analysis – evaluate the traces generated to identify problems.

2.2. Instrumentation

Instrumentation refers to the ability to measure or extract certain features of the program [5]. Programmers implement instrumentation by adding extra code to the application code. The added code may output logging information to the screen or to a file [6].

1. Source code instrumentation – instrument source programs
2. Binary instrumentation – instrument binary executables directly
    a. Static binary instrumentation – inserts additional code and data before execution and generates a persistent modified executable
    b. Dynamic binary instrumentation – inserts additional code and data during execution without making any permanent modifications to the executable

Advantages

Binary instrumentation:

- Language independent
- Machine-level view
- Instrument legacy/proprietary software
- No access to the source code required

Dynamic instrumentation:

- No need to recompile or relink
- Discover code at runtime
- Handle dynamically-generated code
- Attach to running processes

Original Binary
Binary with Injected Instrumentation Code

Figure 2: Difference between original and instrumented executable

Figure 2 pictorially shows how an executable is modified when instrumented. Instrumentation incurs overhead in terms to memory due to the extra data that needs to be stored and also in terms of runtime due to the extra code that needs to be executed.

2.3. Debugging

One of the uses of trace generation is identifying errors in a program. In computer systems, debugging is the process of locating bugs and fixing them. Debugging a program starts by identifying the problem, isolating the source of the problem, and then fixing it. Debugging is a necessary process in almost any software (or hardware) development process [7]. For complex products, debugging is done at multiple levels, for example in unit tests for the smallest units of a system as well as in system tests when the product is used with other existing products. Thorough debugging is a necessary step to ensure software quality. Bugs plague software projects, and today's complicated software stacks make debugging more difficult than ever.

2.4. Compression

When dealing with large-scale parallel programs, any attempt to generate traces will likely result in a huge amount of data. Moreover, such tracing will also incur significant overhead due to the need to transfer and store these vast amounts of data. Hence, we need a way to both decrease the space and the overhead. I do this by applying lossless compression to the traces before storing the trace data on the disk. Compression involves encoding information using fewer bits than the original representation. Compression is useful because it helps reduce resource usage, such as data storage space, and boosts transmission bandwidth. Because compressed data must typically be decompressed before it can be used, this extra processing imposes computational costs during decompression.

CHAPTER 3

RELATED WORK

3.1. Tracing tools

Tracing of parallel programs has always attracted interest. However, as these systems have multiple cores and each core may be able to run multiple threads, we need to generate traces for each thread of each core. Many different types of information can be extracted from a program like the memory addresses, the registers used, etc. One popular approach is to record the function call information, i.e., the function enter and leave events along with the name and possibly the source-code location of each function.

One tool that records function call information is VampirTrace [8]. VampirTrace consists of a tool set and a runtime library for instrumentation and tracing of software applications. It is particularly tailored to parallel and distributed High-Performance Computing (HPC) applications. Its instrumentation component modifies a given application by injecting additional measurement calls during runtime. The tracing component provides the actual measurement functionality used by the instrumentation calls. Through this mechanism, a variety of detailed performance properties can be collected and recorded at runtime. This includes function enter and leave events, MPI communication, OpenMP events, and performance counters. After a successful tracing run, VampirTrace writes all collected data to a trace file in the Open Trace Format (OTF). As a result, the information is available for post-mortem analysis and visualization by various tools. Most notably, VampirTrace provides the input data for the Vampir analysis and visualization tool. Trace files can quickly become very large, especially with automatic instrumentation.

Tracing applications for just a few seconds can result in trace files of several hundred megabytes per core. To protect users from creating trace files of several gigabytes, the default behavior of VampirTrace is to limit the internal buffer to 32 MB per process. Thus, even for large-scale runs, the total trace file size will be moderate. Of course, this means that some important information may not be included in the trace file.

Dyninst [9] is another tool that allows insertion of code into a computer application that is either running or on disk. The API for inserting code into a running application, called dynamic instrumentation, shares much of the same structure as the API for inserting code into an executable file or library, known as static instrumentation. The API also permits changing or removing subroutine calls from the application program. Binary code changes are useful to support a variety of applications, including debugging, performance monitoring, and composing applications out of existing packages. The goal of this API is to provide a machine independent interface to permit the creation of tools and applications that use runtime and static code patching.

3.2. Pin

Pin [10] is a tool for the instrumentation of programs. It supports the insertion of arbitrary code (written in C or C++) in arbitrary places in the executable. The code is added dynamically while the executable is running. It is also possible to attach Pin to an already running process.

Pin provides a rich API that abstracts away the underlying instruction set idiosyncrasies and allows context information such as register contents to be passed to the injected code as parameters. Pin automatically saves and restores the registers that are over-written by the injected code so the application continues to work normally. Limited access to symbol and debug information is available as well.

Advantages of Pin [6]:

- Easy-to-use instrumentation: Uses dynamic instrumentation, does not need source code, recompilation, or post-linking
- Programmable instrumentation: Provides rich APIs to write user-defined instrumentation tools (called Pintools) in C/C++
- Multiplatform: Supports x86, x86-64, Itanium, Xscale

  OS's: Windows, Linux, OSX, Android
- Robust: Instruments real-life applications: Databases, web browsers, multi-threaded applications, and supports signals
- Efficient: Applies compiler optimizations on instrumentation code

3.2.1. Tracing compression techniques

Traces are widely used in industry and academia to study the behavior of programs and processors [11]. The problem is that traces from interesting applications tend to be very large. For example, collecting just one byte of information per executed instruction generates on the order of a gigabyte of data per second of CPU time on a high-end microprocessor. Moreover, traces from many different programs are typically collected to capture a wide variety of workloads. Storing the resulting multi-gigabyte traces can be a challenge, even on today's large hard disks.

Hamou-Lhadj and Lethbridge [12] use common subexpression elimination to compress procedure-call traces. The traces are represented as trees and subtrees that occur repeatedly are eliminated by noting the number of times each subtree appears in the place of the first occurrence. A directed acyclic graph is used to represent the order in which the calls occur. This method requires a preprocessing pass to remove simple loops and calls generated by recursive functions. One benefit of this type of compression is that it can highlight important information contained in the trace, thus making it easier to analyze.

3.2.2 Compression algorithms

3.2.2.1. Bzip2

Bzip2 is a general-purpose compressor that operates at byte granularity [12]. It implements a variant of the block sorting algorithm described by Burrows and Wheeler [13]. Bzip2 [14] applies a reversible transformation to a block of inputs, uses sorting to group bytes with similar contexts together, and then compresses them with a Huffman coder. The block size is adjustable. I use the "--fast" and "--best" option.

3.2.2.2. Gzip

The deflation algorithm used by gzip [15] is a variation of LZ77. It finds duplicated strings in the input data. The second occurrence of a string is replaced by a pointer to the previous string, in the form of a pair (distance, length). Distances are limited to 32 kilobytes, and lengths are limited to 256 bytes. When a string does not occur anywhere in the previous 32 kilobytes, it is emitted as a sequence of literal bytes. I use the “--fast” and “--best” option.

CHAPTER 4

DESIGN AND IMPLEMENTATION

This chapter describes the design of the RTC-Tracer I developed for the efficient function-call tracing of large-scale parallel programs.

4.1. Information Recorded and Extracted

RTC-Tracer extracts and records, for each program thread, the enter and leave events of each executed function, including functions in library code, from which it derives the following information:

1. Function call frequency: The number of times a particular function was invoked.

2. Call edge frequency: The number of a times one function called another.

3. The approximate call stack at every point in the program execution.

4. The full call and return trace with corresponding function and image names.

The first step is to read the symbol table. The symbol table records information such as the name and starting address of each function of a program. A symbol table may only exist during compilation, or it may be embedded in the executable for later exploitation, for example by a debugger.

RTC-Tracer assigns every function in the application a unique ID and instruments every location in the program where a function is entered or left. Moreover, for each running thread, it creates a stack data structure in which every element has two fields: one for the function ID and the other for holding the current stack-pointer register value.

Since each thread has its own runtime stack, each thread is also assigned its own ‘function’ stack. Whenever a thread enters a function, the corresponding function ID is recorded in the trace and pushed onto the function stack along with the value of the stack pointer. Similarly, after a thread leaves a function, a special ID is recorded in the trace and the function stack is popped.

4.2. Stack Correction

For certain types of debugging, a consistent call stack is desired. However, due to function inlining combined with compiler optimizations such as code scheduling, it is not always possible for Pin to determine when an application leaves a function as there is no corresponding return instruction. As a consequence, some function leaves are not recorded, resulting in inconsistent call stacks. To correct this problem as much as possible, RTC-Tracer compares the application’s stack pointer (SP) value to the SP value on the function stack whenever a function is entered or left. If the SP values are out of sync, Pin must have missed one or more leave events, which are then successively added to the trace until the function stack is consistent again. This way, the resulting trace is consistent with a possible runtime stack at every execution point, though it may not be completely precise, which is why I refer to it as an “approximate” call stack.

.

4.3. Compression

Since the symbol information is the same for all running threads of an application, the functions names and corresponding image names are only emitted once to save space. The actual traces tend to be many orders of magnitude larger than the symbol information and pose some major problems. First, writing them to secondary storage can severely slow down the system due to the large bandwidth requirement. Second, the huge amount of required storage space makes the traces slow to access and difficult to handle. Hence, I decided to compress them.

My first idea was to generate the full trace and then compress it. However, that would still have required huge amounts of (temporary) storage and resulted in high bandwidth requirements. The better alternative, which I implemented, is to compress the traces on-the-fly as they are being generated. Hence, I needed to find a compression algorithm that not only compresses function call traces well but also does so very quickly so as not to slow down the execution of the instrumented code.

I used a tool called CRUSHER to determine a good compression algorithm based on several (uncompressed) training traces I had recorded. CRUSHER reported that an LZ component followed by a ZE component would work well. Since my tool supports up to 65535 unique function IDs, the trace entries are two-byte words, which are fed into the LZ component. Its output is interpreted as a sequence of bytes, which is fed into the ZE component for further compression. The output of the ZE component is stored to disk.

The LZ component implements a variant of the LZ77 algorithm [18]. It uses a 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 the component advances to the next entry. If the three values match, the component counts how many values following the current value match the values following that location. The length of the matching substring is emitted and the component advances by that many values.

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

Typically, compression is used in the following way. A buffer is filled with trace data, and whenever the buffer is full, the data is compressed and written out. Unfortunately, this approach imposes long pauses upon application threads whenever compression is invoked, which can be a problem in parallel programs where threads synchronize with each other. To alleviate this problem, I had to implement both of these compression components in an incremental way, i.e., to compress the just generated trace entry without knowing the next trace entries yet. That means the current matching counter in the LZ component may have to either be incremented or emitted and a new count started. Similarly, the current bitmap in the ZE component may have to be updated or a new bitmap started.

The result section of this thesis shows that my implementation does not only compress the traces well but also quickly. In fact, the extra runtime to perform the compression is significantly lower than the overhead saved by not having to emitting the uncompressed trace. In other words, RTC-Tracer tends to run faster with compression than without.

4.4. Trace Reader

To read and process the compressed traces, I wrote a corresponding decompressor. However, the decompressor does not have to work incrementally, so it was easier to implement. It simply performs the inverse operations to recreate the original trace. From this trace, it then computes the four types of information listed in Section 4.1. The user can select which type of information the trace reader should output.

CHAPTER 5

EVALUATION METHOD

5.1. Pin

Pin is a tool for the binary instrumentation of programs. My RTC-Tracer tool is based on Pin. Pin makes it possible to insert arbitrary code (written in C or C++) in arbitrary places in the executable. The code is added dynamically while the executable is running.

The best way to think about Pin is as a "just-in-time" (JIT) compiler [10]. The input to this compiler is a normal executable. Pin intercepts the program execution and generates ("compiles") new code for each straight-line code sequence it encounters and transfers control to the generated sequence. In JIT mode, the only code ever executed is the generated code. The original code is only used for reference. When generating code, Pin gives the user the opportunity to inject his or her own code, which is called program instrumentation.

Conceptually, instrumentation consists of the following two components.

1. A mechanism that decides where and what code to insert
2. The code to execute at the insertion points

These two components are referred to as instrumentation code and analysis code. Both components live in a single executable called a Pintool. Pintools can be thought of as plugins that can modify the code-generation process inside Pin. RTC-Tracer is such a Pintool. I am using Pin 2.14 in this thesis.

5.2. Benchmarks

In this thesis, I am using the Mantevo (version 3.0) [16] miniapps created by a team at Sandia National Laboratories to generate traces and evaluate RTC-Tracer. Mantevo is a multi-faceted application performance project, with the goal to provide open source-software to promote informed algorithm, application, and architecture decisions in the HPC community. It provides application performance proxies known as miniapps. Miniapps combine some or all of the dominant numerical kernels contained in an actual stand-alone application. They include libraries wrapped in a test driver providing representative inputs. They may also be hard-coded to solve a particular test case so as to simplify the need for parsing input files and mesh descriptions. Miniapps range in scale from partial, performance-coupled components of the application to a simplified representation of a complete execution path through the application. The following subsections describe each miniapp in more detail.

5.2.1. *MiniFE*

This is a miniapp that mimics the finite element generation, assembly, and solution for an unstructured grid problem. The physical domain is a 3D box with configurable dimensions and a structured discretization (which is treated as unstructured). The domain is decomposed using a recursive coordinate bisection (RCB) approach and the elements are simple hexahedra. The problem is linear and the resulting matrix symmetric, so a standard conjugate gradient algorithm is used as solver with a general sparse matrix data format and no preconditioning.

5.2.2. MiniGhost

MiniGhost is a finite difference mini-application that implements a difference stencil across a homogenous three-dimensional domain.

5.2.3. MiniMD

This is a parallel molecular dynamics (MD) simulation package written in C++ and intended for use on parallel supercomputers and new architectures for testing purposes. MiniMD uses spatial decomposition MD, where individual processors in a cluster own subsets of the simulation box.

5.2.4. MiniXyce

This is a circuit simulation application. Circuit simulation is the cornerstone of the electrical design automation industry and is a crucial part of commercial electrical design. Like most circuit simulation tools, MiniXyce is based on a modified nodal analysis formulation, resulting in Kirchoff Current Laws being enforced across a potentially arbitrary network. The resulting system of differential-algebraic equations is solved implicitly using Newton-based methods. Traditional circuit codes have almost exclusively relied upon direct matrix solvers, but preconditioned GMRES is the method of choice for parallel simulation.

5.2.5. PathFinder

PathFinder searches for “signatures” within graphs. The graphs being searched are directed and cyclic. Many but not all nodes within the graph have labels. Any given node may have more than one label, and any label may be applied to more than one node. A signature is an ordered list of labels. PathFinder searches for paths between labels within the signature. PathFinder returns success if there is a path from a node with the first label in the signature that passes through nodes with each label in order, ultimately reaching a node with the last label in the signature. Labeled nodes need not be contiguous on any given path. PathFinder simply searches until a signature is satisfied or all path-ways have been exhausted.

5.2.6. TeaLeaf

TeaLeaf is a mini-app that solves the linear heat conduction equation on a spatially decomposed regular grid using a five-point stencil with implicit solvers. In TeaLeaf, temperatures are stored at the cell centers. A conduction coefficient is calculated that is equal to the cell centered density or the reciprocal of the density. This is then averaged to each face of the cell for use in the solution. Solving is carried out using an implicit method due to the severe time-step limitations imposed by the stability criteria of an explicit solution for a parabolic partial differential equation. The implicit method requires the solution of a system of linear equations, which form a regular sparse matrix with a well-defined structure.

5.2.7. HPCCG

HPCCG is similar to MiniFE but generates a synthetic linear system. The focus is entirely on the sparse iterative solver.

5.2.8. MiniSMAC2D

This mini-application solves the finite-difference 2D incompressible Navier-Stokes equations with the Spalart-Allmaras one-equation turbulence model on a structured body-conforming grid. The grid is partitioned into subgrids that are load balanced for the number of MPI ranks requested by the user. Subgrids overlap by one grid point for point-to-point boundary communication. MiniSMAC2D currently features implicit line and symmetric Gauss-Seidel relaxation algorithms. As a test case, input files are included for a C-grid around a NACA 4412 airfoil at various angles of attack.

5.2.9. CoMD

This is an extensible molecular dynamics proxy applications suite featuring the Lennard-Jones potential and the Embedded Atom Method potential.

5.3. Configuration

For all tested MPI applications, traces are generated for 2, 4, 8, 16, 32 and 64 nodes, i.e., 32, 64, 128, 256, 512 and 1024 cores, respectively. For the OpenMP applications, traces are generated on one node with 16 cores and 2, 4, 8, 16 and 32 threads. The traces are compressed using the algorithm described in Chapter 4 before they are stored to disk. For comparison purposes, the resulting traces are decompressed and the gzip --

fast, gzip --best, bzip --fast, and bzip --best algorithms are applied to them. All the experiments were conducted on the Stampede system at TACC [17]. Stampede has two 8-core Xeon E5 processors per node running at 2.7 GHz clock frequency. There are a total of 6400 nodes, each with a 32 GB of main memory. The compilers used for the experiments are mvapich2 2.1 for MPI codes and gcc 4.9.1 for OpenMP applications. In both cases, I used the –O3 compiler flag.

CHAPTER 6

RESULTS

The following subsections show the main results of the experiments I conducted. First, I investigate the overhead introduced by my RTC-Tracer tool. Second, I study the overhead of employing on-the-fly compression. Third, I evaluate the resulting compression ratio and compare it to the standard compression algorithms gzip and bzip2. Fourth, I study the compression speed. Finally, I examine the required trace-data bandwidth.

6.1. Relative runtime

This subsection investigates the overhead incurred by RTC-Tracer when tracing function calls and returns and performing on-the-fly trace compression. The overhead is computed by dividing the runtime of the Pin instrumented application by the runtime of normal execution of the application with the same input. Figure 3 shows the results for different core counts. It lists the various applications along the x-axis as well as the average overhead. The y-axis represents the overhead. Values above 1.0 indicate a slowdown due to the tracing. I conducted experiments with 32, 64, 128, 256, 512 and 1024 cores for the MPI applications and with 2, 4, 8, 16, 32, and 64 threads for the OpenMP applications.

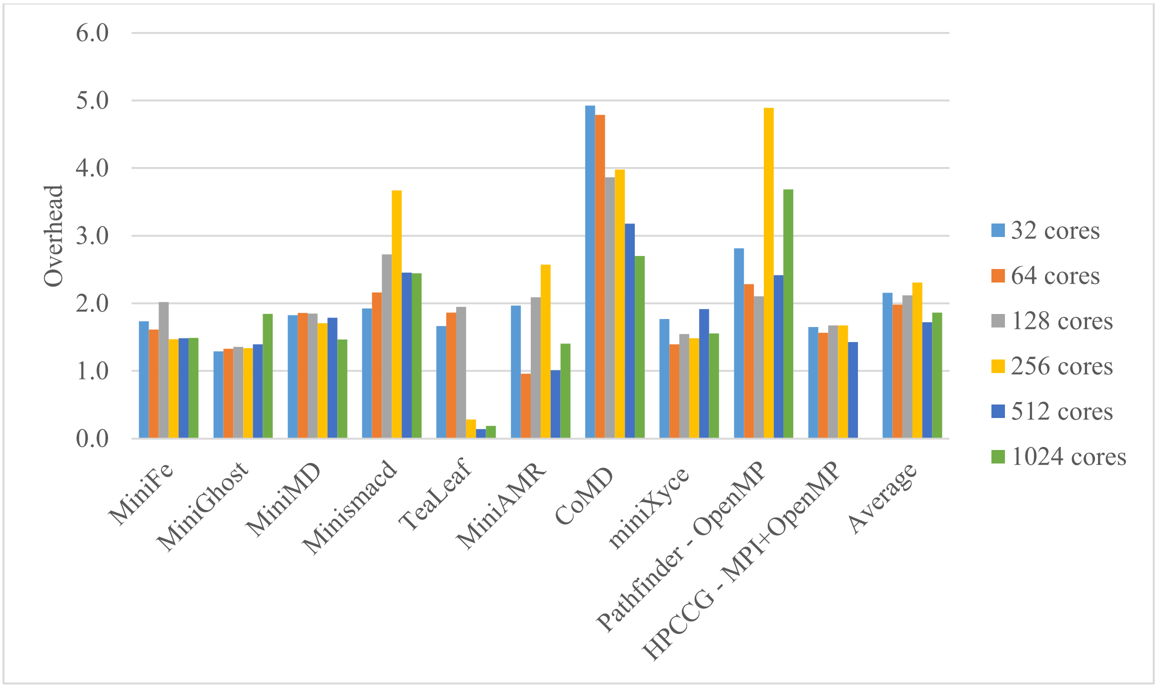

Figure 3: Runtime overhead of RTC-Tracer (pathfinder and HPCCG uses lesser cores than indicated)

These results highlight two key aspects of RTC-Tracer. First, the overhead is quite low. I found no application or core count for which it exceeded a five-fold slow-down. On average, the overhead is only about a factor of two, meaning that the applications run twice as long with full tracing than they do without any tracing. Second, and equally importantly, the overhead stays roughly the same as the applications are scaled to larger core/thread counts. In fact, the overhead seems to decrease slightly at larger core/thread counts. This demonstrates that my approach is scalable and will probably also work at even larger scales than what I was able to test.

HPCCG is missing the green bar as this application does not run with 64 threads. The overhead of the TeaLeaf application is below one, indicating that the run with the tracing turned on was faster than the run without tracing. I can only surmise that TeaLeaf is probably a nondeterministic application whose runtime depends on the timing of its threads, which the tracing modifies, as it is unrealistic to achieve a speedup by tracing an application.

6.2. Relative runtime of RTC-Tracer with and without compression

This subsection investigates the effect of the on-the-fly compression implemented in the RTC-Tracer by comparing the runtime of the RTC-Tracer with and without compression. Figures 4 and 5 shows the runtime without compression divided by the runtime with compression turned on. Numbers above 1.0 mean that the version without compression is slower.

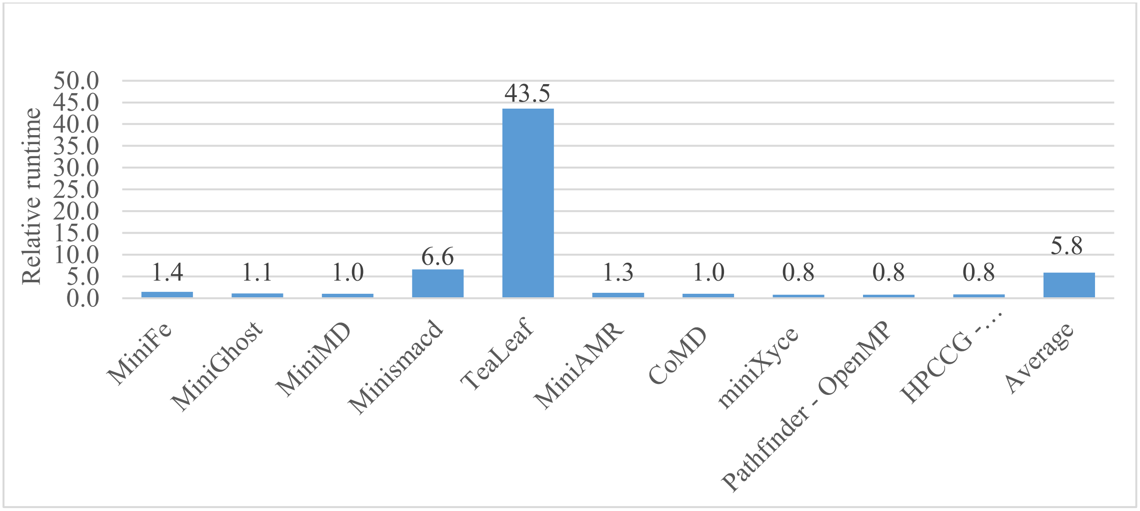

Figure 4: Relative runtimes of RTC-Tracer without compression for 32 cores (pathfinder and HPCCG uses lesser cores than indicated)

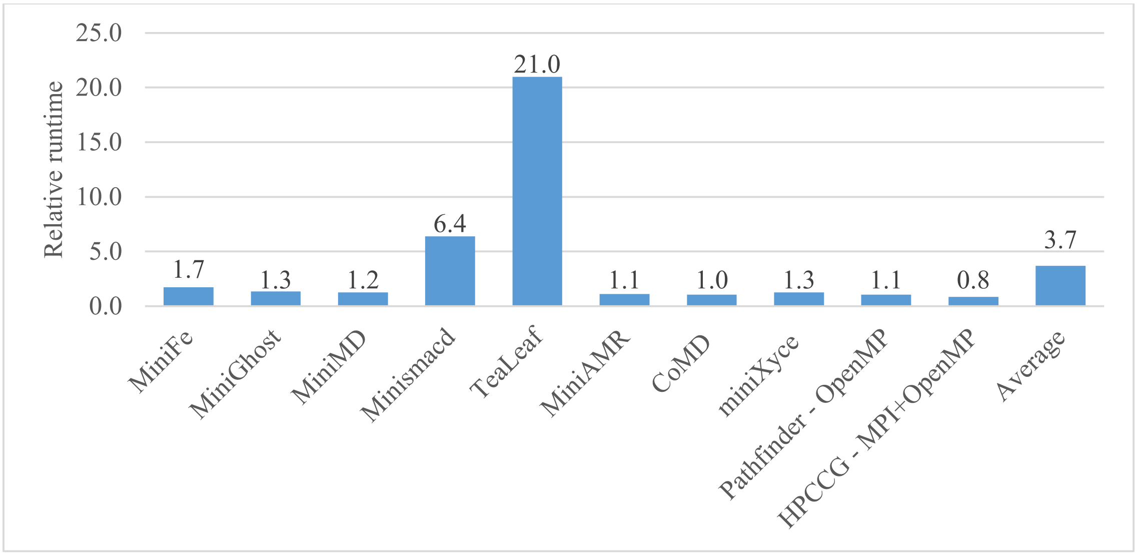

Figure 5: Relative runtimes of RTC-Tracer without compression for 128 cores (pathfinder and HPCCG uses lesser cores than indicated)

Interestingly, for most of the tested applications, RTC-Tracer incurs a lower overhead when compression is turned on. This result seems counterintuitive as the compression takes time to perform. However, the resulting data footprint is so much smaller that fewer calls to write out trace data need to be executed and the required memory and disk bandwidths are much lower, resulting in a net benefit. In other words, RTC-Tracer is often faster with compression than without.

With 32 cores, there are three applications that can be traced faster without compression. However, with 128 cores, only one of those application programs is still faster without compression. All other applications run faster with trace compression enabled. This again indicates that RTC-Tracer scales well and will likely also perform well on larger core counts than I tested.

6.3. Compression ratio

This subsection investigates the compression ratio achieved by the customized compression algorithm used in RTC-Tracer and compares it with the compression ratios of standard compression algorithms applied to the decompressed traces. The compression ratios in Figures 6 and 7 are the compressed trace size divided by the uncompressed trace size.

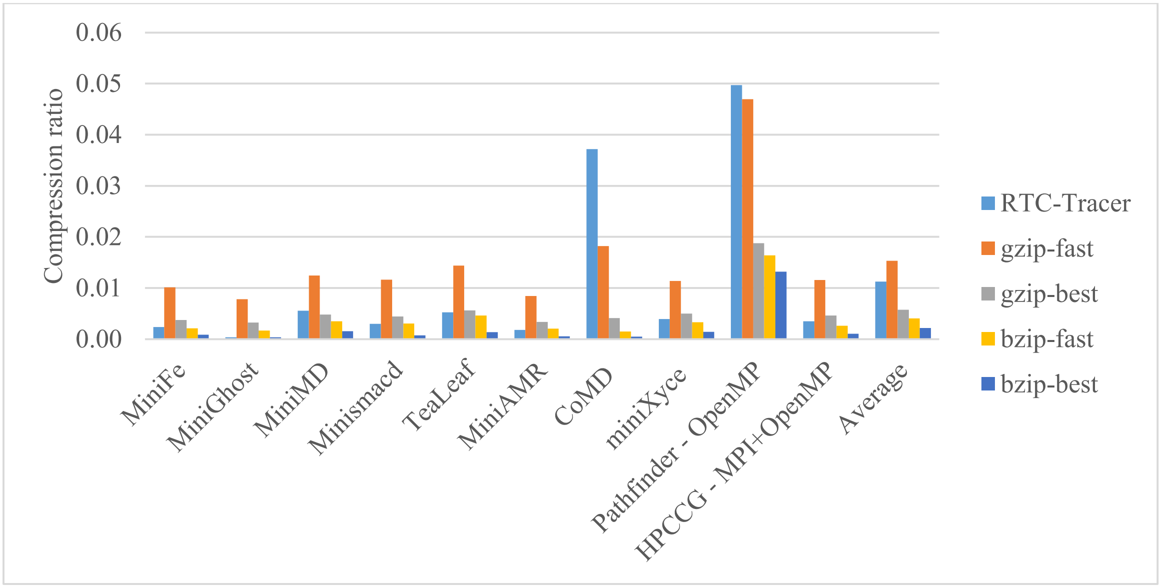

Figure 6: Compression ratio for traces generated with 32 cores (pathfinder and HPCCG uses lesser cores than indicated)

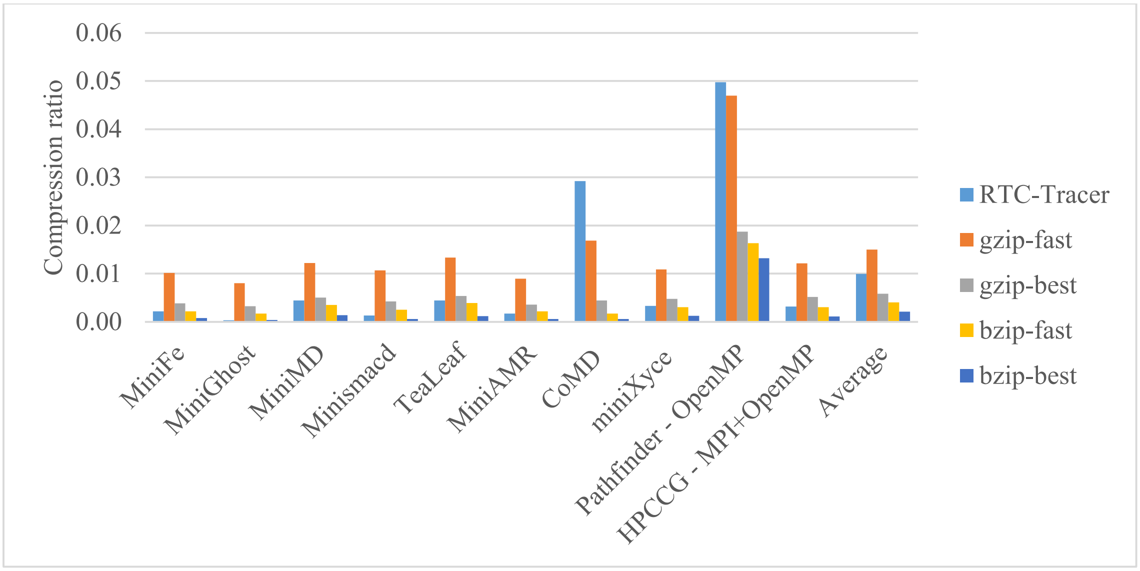

Figure 7: Compression ratio for traces generated with 128 cores (pathfinder and HPCCG uses lesser cores than indicated)

Figures 6 and 7 show that, in many cases, the algorithm in the RTC-Tracer compresses better than gzip --fast. Only bzip2 --best consistently outperforms it. The compression ratios are very stable between 32 cores and 128 cores. On average, the RTC-Tracer compresses the traces by over a factor of 100. Gzip --best, bzip2 --fast and bzip2 --best have a better average compression ratio than RTC-Tracer. However, RTC-Tracer outperforms gzip --best and even bzip2 --fast on quite a few programs. It just compresses two programs' traces relatively poorly, which is why its average is not better. The compressed trace sizes for gzip --fast, gzip --best, bzip --fast and bzip --best are listed in the appendix. Overall, the utilized compression algorithm performs well, especially given that it performs incremental on-the-fly compression that needs to be very fast.

6.4. Compression speed comparison

In this subsection, the runtime of gzip --fast, gzip --best, bzip2 --fast and bzip2 --best are compared with the runtime of RTC-Tracer. In the following four figures, the y-axes represent the runtime of these compression algorithms relative to the runtime of RTC-Tracer, which includes the runtime of the actual application. Results above 1.0 indicate that the RTC-Tracer runs faster than the compression tools.

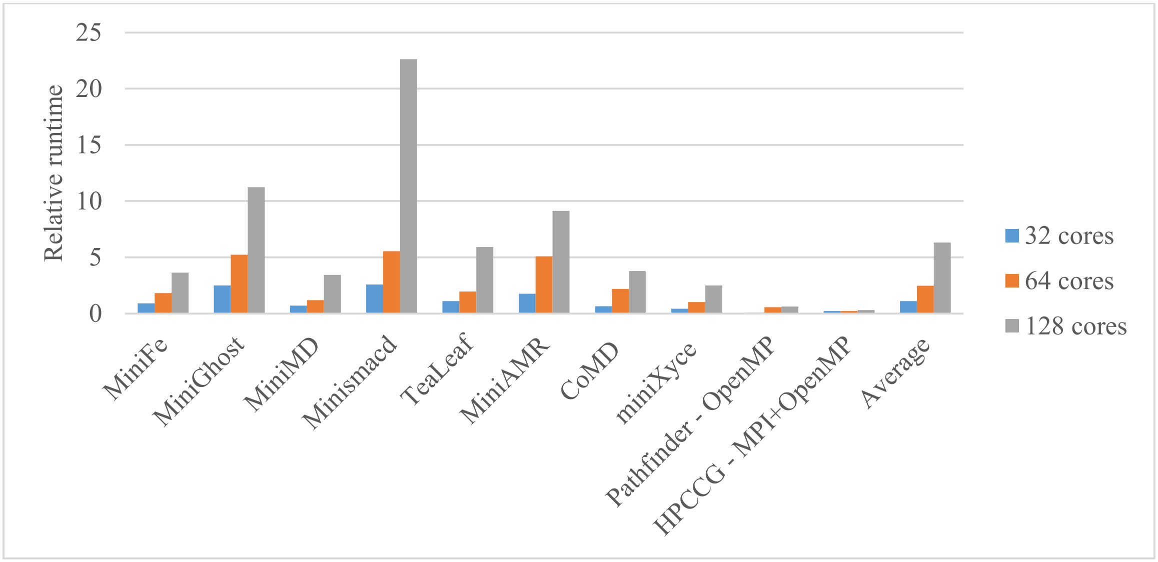

Figure 8: Execution time comparison of gzip --fast with respect to RTC-Tracer (pathfinder and HPCCG uses lesser cores than indicated)

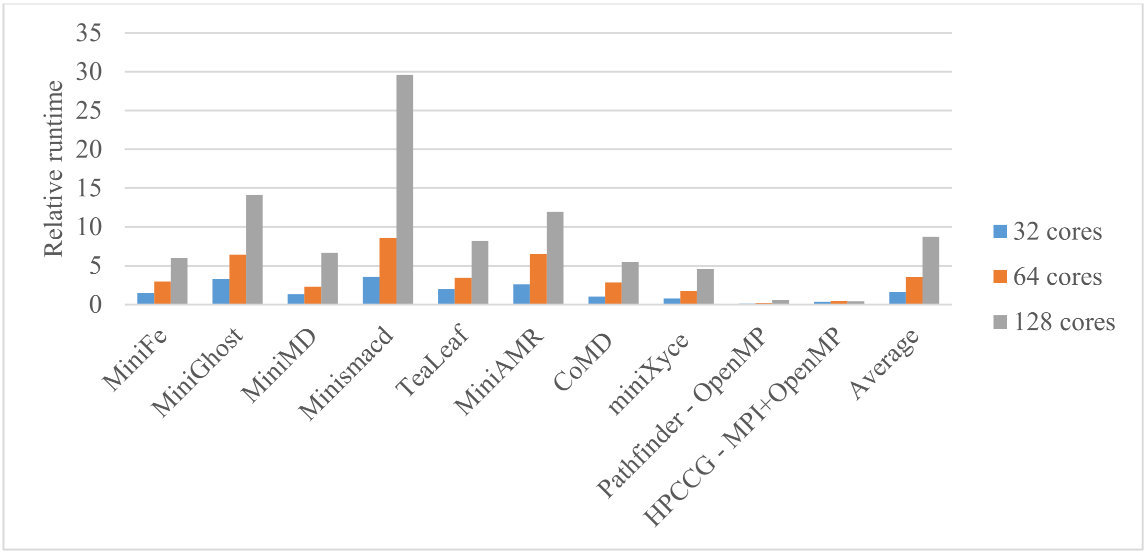

Figure 9: Execution time comparison of gzip --best with respect to RTC-Tracer (pathfinder and HPCCG uses lesser cores than indicated)

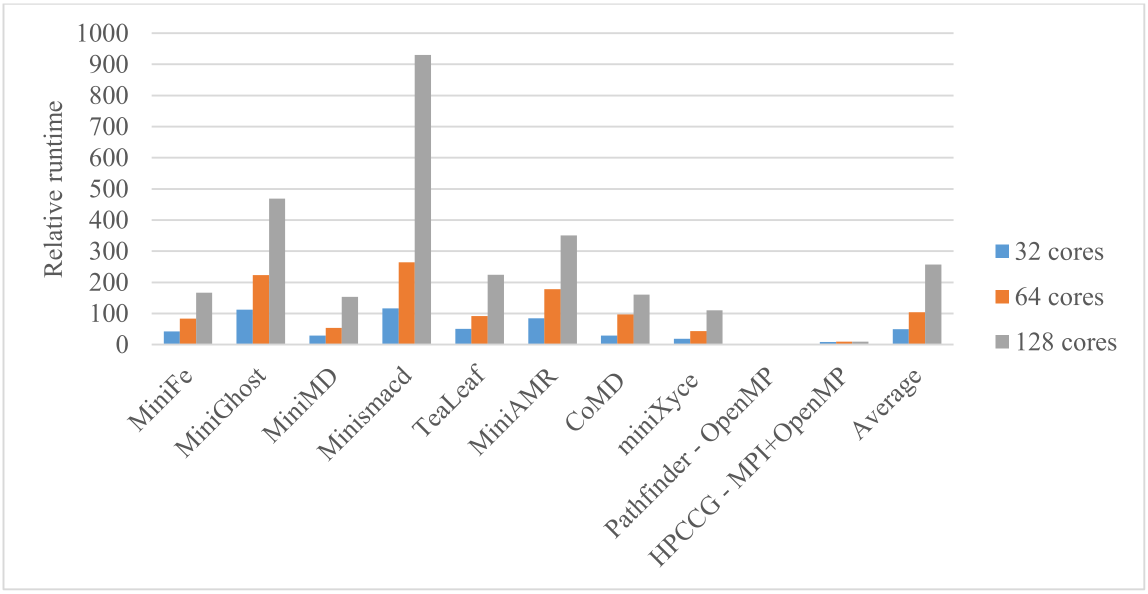

Figure 10: Execution time comparison of bzip2 --fast with respect to RTC-Tracer (pathfinder and HPCCG uses lesser cores than indicated)

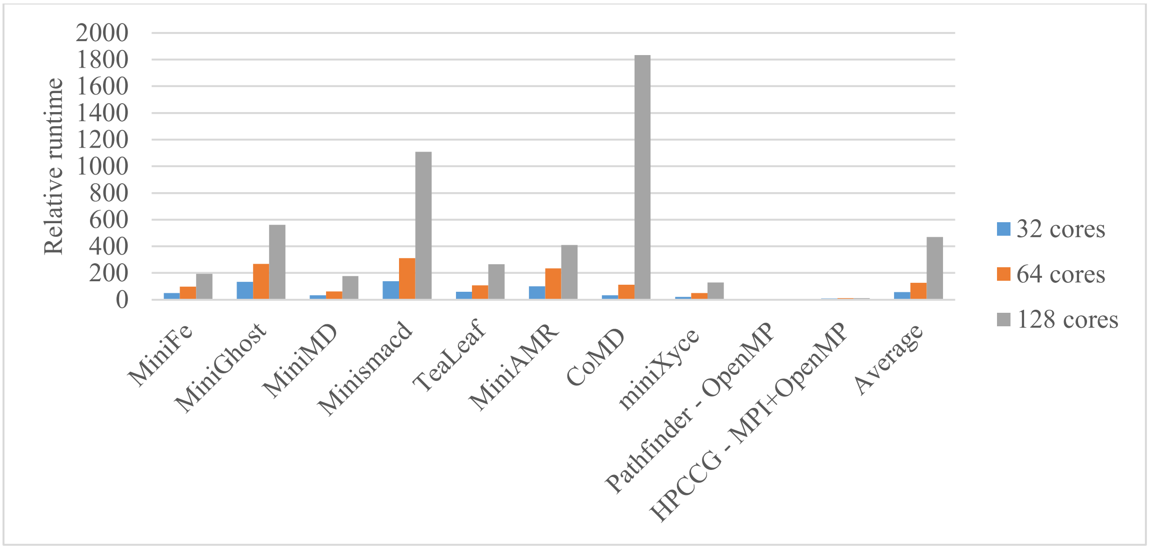

Figure 11: Execution time comparison of bzip2 --best with respect to RTC-Tracer (pathfinder and HPCCG uses lesser cores than indicated)

The two general-purpose compressors running in either mode, but especially in the --best mode, are much slower than RTC-Tracer, which compresses the traces nearly as well. This is especially surprising since the RTC-Tracer running time includes the application execution time. Moreover, the runtime advantage of RTC-Tracer clearly increases for larger core counts. The reason for this is that bzip2 and gzip are serial implementations whereas RTC-Tracer naturally runs in parallel as each application thread compresses its own trace concurrently with the work performed by the other threads.

6.5. Bandwidth requirements

The decompressed trace sizes shown in the appendix reveal that the amount of data collected from just a few seconds of runtime is huge and increases as the number of cores/threads increases. Writing that much data to secondary storage throughout the execution of an application is unrealistic for large-scale programs and programs running for more than a few seconds. Moreover, the data would flood the network and/or disk system, which potentially slows down the application and places a great burden on the communication subsystem.

This subsection shows the bandwidth required with RTC-Tracer, which compresses the data before they are even written to memory or disk or sent over the network. Table 1 lists the bandwidth per core for RTC-Tracer. The values are in kilobytes per second (per core).

Miniapps	32 cores	64 cores	128 cores
miniFE	9.2	9.2	8.2
miniGhost	3.6	3.5	3.7
miniMD	15.1	13.9	15.5
miniSmacd	31.5	39.7	27.9
Tealeaf	24.1	18.1	22.7
miniAMR	13.6	15.5	12.7
CoMD	94.9	87.7	104.0
miniXyce	6.6	7.3	8.2
Pathfinder	1.4	0.9	0.5
HPCCG	2.6	1.4	0.7
Average	20.26	19.72	20.41

Table 1: Bandwidth required for compressed traces per core for RTC-Tracer

As the table shows, the required bandwidth is quite low, ranging from less than a kilobyte per second to just over 100 kilobytes per second, with an average of about 20 kilobytes per second. These bandwidths are just a small fraction of what a modern disk or network interface can handle, meaning that tracing with my tool should not significantly impact the communication subsystem. Note also that the bandwidth per core remains nearly constant as we scale the application to more cores and compute nodes, which again indicates that my approach scales well.

CHAPTER 7

SUMMARY

7.1. Summary

This thesis presents a new function-call tracing tool called RTC-Tracer for large-scale parallel systems that records the function enter and function leave events of every thread of every core. The new tool scales well to more than 1000 cores and has an average overhead in the range of 1.73 to 2.31, which remains approximately constant when scaling to larger core counts. It incorporates a customized compression algorithm that compresses the trace data by over a factor of 100 on average. In fact, it not only compresses about as well as standard compression algorithms but also compresses the data much more quickly, making on-the-fly compression possible. The resulting amount of bandwidth required per core is in the range of less than a kilobyte per second to a hundred kilobytes per second, which is low compared to today's disk and network throughputs. To the best of my knowledge, RTC-Tracer is the first tracing tool that combines a low overhead in terms of runtime with a low overhead in terms of emitted data and exhibits excellent scaling.

7.2. Future Work

Based on the results of this work, future work aims at using RTC-Tracer on higher core counts to check its true scalability, testing it on different systems, and, of course, using the resulting traces for various purposes like debugging or performance analysis of large-scale parallel programs.

APPENDIX SECTION

This section lists all the raw performance data for all tested configurations.

Miniapps	RTC-Tracer	Decom-pressed file	gzipfast	gzipbest	bzipfast	bzipbest
miniFE	4060452	1702179508	17211201	6381210	3605361	1420826
miniGhost	4617202	14105059920	110231840	45395407	23524998	4528823
miniMD	7487896	1342433816	16693130	6464096	4675068	2077442
miniSmacd	17570100	5916002724	68771205	26224771	18153318	4191933
Tealeaf	11613895	2222591456	32018729	12425011	10237816	2997136
miniAMR	7684711	4288048804	36228099	14389242	8689694	2293252
CoMD	2101428618	56545850496	1029625461	232393485	82701014	27184562
miniXyce	3793692	963204800	10965942	4787497	3150104	1353497
Pathfinder	390096	7845440	368352	147040	128304	103568
HPCCG	2615736	753430804	8730953	3480891	1982573	777832

Table A.1 File sizes (in bytes) of the traces generated with 32 cores

Miniapps	RTC-Tracer	Decompressed file	gzipfast	gzipbest	bzipfast	bzipbest
miniFE	8358821	3467578944	35499232	13179877	7584649	2927496
miniGhost	9568519	29316227444	230384205	95039180	49182783	9532335
miniMD	15140005	2700342976	33940324	13270296	9652504	4184308
miniSmacd	38998749	16010404908	191742728	76534248	50798774	12252671
Tealeaf	19341202	4474861812	59476525	23396842	17289159	5135695
miniAMR	18339586	9785843192	86730321	34323459	21844614	5802774
CoMD	2094958941	100501551340	1409089619	388440433	140285220	33332916
miniXyce	7782203	2052089784	23346313	10164924	6674089	2803875
Pathfinder	395952	5797364	309041	134091	118252	104135
HPCCG	2662072	811550556	9662076	4037472	2295976	856195

Table A.2 File sizes (in bytes) of the traces generated with 64 cores

Miniapps	RTC-Tracer	Decompressed file	gzipfast	gzipbest	bzipfast	bzipbest
miniFE	17719457	8220005312	83203337	31571744	17920258	6684864
miniGhost	20312355	62054408644	499806741	202844746	107700620	21780294
miniMD	30009160	6816874740	83364532	34247521	23955113	9489038
miniSmacd	88117610	66553655080	711388720	281290931	165305981	36911657
Tealeaf	44958065	10201349540	136010658	54564918	39769487	11734322
miniAMR	43314004	25746905440	231124441	92318920	55548638	15127955
CoMD	259977767	89013890004	1501012314	396956398	152909753	52801774
miniXyce	15918271	4783886664	52061240	22859222	14456730	5934702
Pathfinder	390096	7845440	368352	147040	128304	103568
HPCCG	2708323	849278788	10322036	4359806	2599995	937213

Table A.3 File sizes (in bytes) of the traces generated with 128 cores

LITERATURE CITED

[1] G. Tassey, "The Economic Impacts of Inadequate Infrastructure for Software Testing." May 2002. [Online]. Available: http://www.rti.org/pubs/software_testing.pdf

[2] http://people.ac.upc.edu/felix/pasa06.pdf

[3] https://en.wikipedia.org/wiki/Tracing (software)

[4] https://msdn.microsoft.com/en-us/library/zs6s4h68(v=vs.110).aspx

[5] https://en.wikipedia.org/wiki/Instrumentation_(computer_programming)

[6] https://cs.gmu.edu/~astavrou/courses/ISA_673_S13/PIN_lecture.pdf

[7] http://searchsoftwarequality.techtarget.com/definition/debugging

[8] https://tudresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/projekte/vampirtrace/dateien/VT-UserManual-5.14.4.pdf

[9] http://www.dyninst.org/sites/default/files/manuals/dyninst/DyninstAPI.pdf

[10] https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/

[11] http://cs.txstate.edu/~mb92/papers/tc05.pdf

[12] A. Hamou-Lhadj and T.C. Lethbridge, “Compression Techniques to Simplify the Analysis of Large Execution Traces,” Proc. 10th Int’l Workshop Program Comprehension, pp. 159-168, June 2002.

[13] M. Burrows and D.J. Wheeler, “A Block-Sorting Lossless Data Compression Algorithm,” Digital SRC Research Report 124, May 1994.

[14] http://www.bzip.org/

[15] http://www.gzip.org/algorithm.txt

[16] https://mantevo.org/

[17] https://portal.tacc.utexas.edu/user-guides/stampede#running

[18] https://en.wikipedia.org/wiki/LZ77_and_LZ78