ABSTRACT

ANANTHAKRISHNAN, SRINATH KRISHNA. Customized Scalable Tracing with in-situ Data Analysis. (Under the direction of Frank Mueller.)

The next few years have been projected to usher in the wake of the exascale era where systems are expected to be comprised of several million cores. Applications that are scaled to run on these systems can generate extensive amounts of data, and experience with current petascale systems shows that developers are struggling to keep pace with this increase in scale. A large number of problems surface at high scale. Root cause diagnosis of such problems often fails because tools, specifically trace-based ones, cannot afford to record the entire set of metrics they measure owing to the prohibitive cost of instrumentation.

We propose to address these tool scalability problems by combining customized tracing and providing support for in-situ data analysis. To this end, we have developed ScalaJack, a framework that supports dynamic customizable instrumentation and pluggable extension capabilities through which a user can instrument the interfaces that are pertinent to the problem at hand and also perform in-situ data analysis at the specific points of execution thus achieving scalable trace sizes. The framework also allows users to eliminate the presence of cross cutting concerns by factoring code into modular aspects thus achieving better maintainability. We evaluate the viability of ScalaJack by demonstrating its ability with several case studies of traditional HPC applications.

© Copyright 2013 by Srinath Krishna Ananthakrishnan

All Rights Reserved

Customized Scalable Tracing with in-situ Data Analysis

by
Srinath Krishna Ananthakrishnan

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Networking

Raleigh, North Carolina

2013

APPROVED BY:

Xiaosong Ma	George Rouskas

Frank Mueller
Chair of Advisory Committee

DEDICATION

To *Amma*, *Appa* and *Matty*.

BIOGRAPHY

Srinath Krishna Ananthakrishnan was born to Mr. K. V. Ananthakrishnan and Mrs. A. Jayanthi in the town of Thanjavur, a small town in South India in 1988. He did most of his schooling in the beautiful city of Chennai after moving there in the summer of '94. He did his undergraduation at Madras Institute of Technology in Electronics and Communication from 2005-09. He then worked at Cisco Systems (formerly Starent Networks Corp.), Bangalore for two years on their Packet Data technologies before embarking on his first overseas journey to the United States for his Masters at North Carolina State University. During the course of his Masters, he did his internship at Riverbed Technology on the TCP/IP stack of their WAN optimization product. Motivated by his interest in Parallel systems, he started working under Dr. Frank Mueller on ScalaJack. After his Masters, he would be continuing from where he left off at Riverbed. In addition to being interested in computers, he likes to sing, play the piano and hopes to learn a bunch of other instruments.

ACKNOWLEDGEMENTS

It is hard to believe that two wonderful years have whizzed past so quickly. It has been a roller-coaster ride with several caffeine-influenced nights, several days of running in the biting cold to attend classes and meetings, hours of despair trying to solve the simplest of bugs, horrors of having our home robbed, the joys of getting your code working, the moments of inspiration fuelled by last minute anxiety, the wonderful hours spent by playing volleyball, those eventful birthday bashes etc. As Steve Jobs would put it, by connecting the dots backward, I realise that things turned out to be perfect. This journey would not have been possible without the help and guidance of many people.

First, I would like to thank my family. I thank my parents for their immense support and faith in me. I am forever indebted to my uncle for his support and motivation for my higher studies. I also thank my mentor from Cisco Systems for all the the UNIX *gyaan* he imparted. But for his suggestions, it would have been a hard time combing through the several lines of ScalaTrace.

Next, I would like to thank my advisor Dr. Frank Mueller for his encouragement and patience during the course of my work. I am forever indebted to Dr. Mueller for his amazing guidance and the confidence he showed in me. Our discussions have always been of help to me in seeing problems in a new light and also paved the way for new ideas to tackle interesting problems. In addition to the technical exchanges, Dr. Mueller's professionalism and dedication are definitely an inspiration for me and I am sure these would be of immense benefit in the years to come. This was an enriching experience for me, both technically and professionally. I would also like to thank Dr. Xiaosong Ma and Dr. George Rouskas for agreeing to serve on my advisory committee.

Last but not least, I would like to thank Xing Wu, for helping me ramp up to ScalaTrace and answering my frequent annoyances. Also, I would like to thank Karthik, David, Arash and James for making 3226 my second home. I would also like to thank my friends Gokul, Lakshman, Raaj and Raghavendran for taking up my cooking schedule from time to time and providing me with support, friendly banter during hard times and most importantly, good food.

TABLE OF CONTENTS

LIST OF TABLES . . . . **vii**

LIST OF FIGURES . . . . **viii**

Chapter 1 Introduction **1**
- 1.1 The exascale era 1
- 1.2 Tool deficiencies at exascale 2
  - 1.2.1 Scalability of tracing tools 2
  - 1.2.2 Scalability of instrumentation data 2
  - 1.2.3 Customizable instrumentation 3
  - 1.2.4 Maintainability of codes 3
- 1.3 Hypothesis 3
- 1.4 Contributions 4
- 1.5 Organization 5

Chapter 2 Background **6**
- 2.1 ScalaTrace 6
  - 2.1.1 Intra-node compression 7
  - 2.1.2 Inter-node compression 8
- 2.2 Aspect-Oriented Programming 9

Chapter 3 Design and Implementation **11**
- 3.1 High-level design 11
- 3.2 Custom Event Framework 13
  - 3.2.1 Registering custom events 13
  - 3.2.2 Invoking custom events 14
- 3.3 User Callback Framework 16
  - 3.3.1 Registering callbacks 17
  - 3.3.2 Compression of user data in callbacks 17
- 3.4 Preloader 18
  - 3.4.1 Specification file 19
  - 3.4.2 Statically linked routines 20
  - 3.4.3 Dynamically linked routines 20

Chapter 4 Evaluation **23**
- 4.1 Evaluation Setup 23
  - 4.1.1 Aspect-oriented metrics 23
- 4.2 Performance analysis 24
- 4.3 Visualization and Load balancing 27
- 4.4 Data analysis in-situ with trace analysis 32

Chapter 5 Related Work **36**

Chapter 6 Future Work **38**

Chapter 7 Conclusion **40**

REFERENCES **42**

LIST OF TABLES

Table 4.1 Aspect metrics for IS 29
Table 4.2 Aspect metrics for CLAMR 32
Table 4.3 Aspect metrics for TF-IDF 35

LIST OF FIGURES

Figure 1.1 Projected performance based on the Nov '12 Top500 list 2

Figure 2.1 Nested MPI events 7
Figure 2.2 ScalaTrace's Intra-node compression 8
Figure 2.3 Inter-node radix tree compression 9
Figure 2.4 Cross-cutting concerns in a traditional application 10
Figure 2.5 Application after aspects refactoring 10

Figure 3.1 ScalaJack's High-level Design 12
Figure 3.2 Typical Application workflow with ScalaJack 12
Figure 3.3 Communication and Computation phases in an MPI program 13
Figure 3.4 MPI events in a loop 14
Figure 3.5 Successive Custom events 15
Figure 3.6 Custom events with MPI events 16
Figure 3.7 Nested custom events 16
Figure 3.8 The design of the preloader 19
Figure 3.9 A sample specifiction for the preloader 20
Figure 3.10 Generated code for statically compiled routines 21
Figure 3.11 Generated code for dynamically linked routines 22

Figure 4.1 Outline of the IS benchmark 25
Figure 4.2 Outline of the IS benchmark with ScalaJack 26
Figure 4.3 Trace file sizes with IS 27
Figure 4.4 Execution times with IS 28
Figure 4.5 Percentage overhead with ScalaJack for IS 28
Figure 4.6 Outline of CLAMR 29
Figure 4.7 Outline of CLAMR with ScalaJack 30
Figure 4.8 Execution times with CLAMR 31
Figure 4.9 Percentage overhead with ScalaJack for CLAMR 31
Figure 4.10 Outline of TF-IDF 33
Figure 4.11 Outline of TF-IDF with ScalaJack 34
Figure 4.12 Execution times with TF-IDF 34
Figure 4.13 Percentage overhead with ScalaJack for TF-IDF 35

Chapter 1

Introduction

1.1 The exascale era

Recent years have seen an exponential upsurge in the concurrency levels of large-scale supercomputers. The Top500 list is a biannual list of supercomputers ranked on the basis of their computing power [6]. The recent edition of the Top500 list released in November 2012 has Oak Ridge National Laboratory's Titan machine at the top, boasting 560,640 cores with Nvidia Kepler GPUs capable of performing about 17.6 PFLOP/s. As can be seen from Figure 1.1, this decade has been projected to usher in the wake of the exascale era, where systems are expected to be comprised of several millions of components with up to $O(10^{11})$ threads [2].

Even current petascale systems, e.g. Titan, have several thousands of active tasks. Experience with such systems suggests that application developers are struggling to cope with the immense parallelism offered. Scaling applications to this level of parallelism still remains an immense challenge. With such an increase in scale, applications are more likely to fall short relative to expected performance due to bottlenecks. Experience suggests that typical application codes suffer from scalability issues when the concurrency levels increase by a factor of 10. Such issues can be manifestations of problems ranging from inefficient communication between tasks to inefficient usage of memory hierarchies. Analysis of such problems typically requires the knowledge of an application's global as well as local behavior.

It is also interesting to note that, more than 10% of the supercomputers on the Top500 list sport hybrid architectures with accelerators. It is evident that with the race to exascale, supercomputers are beginning to be comprised of hybrid architectures with several thousand cores teaming up with accelerators to churn out immense computing capabilities.

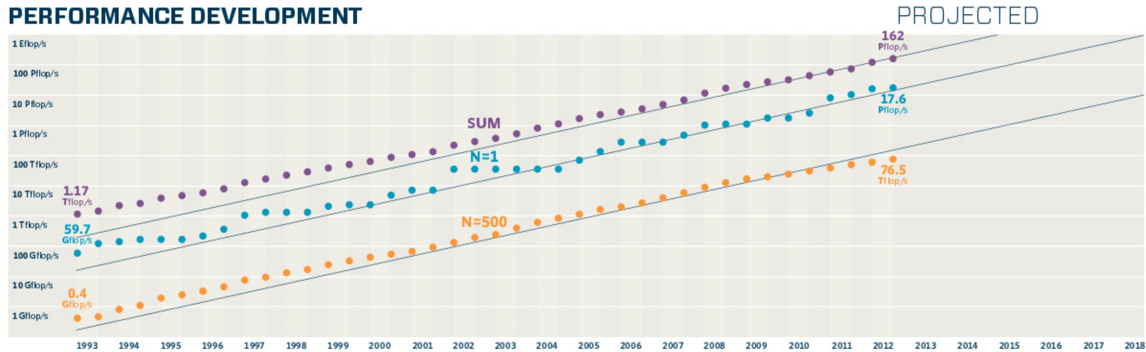

Figure 1.1: Projected performance based on the Nov '12 Top500 list

1.2 Tool deficiencies at exascale

1.2.1 Scalability of tracing tools

One of the most frequently used techniques for root cause diagnosis is tracing, where specific events in the application are identified and traced as part of an execution of the application. Tracing differs from profiling in that it tries to preserve more data about the execution, including the chronology of events that took place, while profiling is inherently lossy and focuses on overall performance metrics related to specific event types. With trace-based tools, it becomes increasingly difficult to isolate problems since their causes are often subtle to identify and the instrumentation cost can be prohibitive. Tools generally try to instrument interfaces exhaustively because developers cannot predict the subset of interfaces that might be pertinent to an application's problem. Using these tools might be prohibitive and expensive for two reasons. First, instrumenting more interfaces than necessary might introduce perturbations that can mask the problem at hand. Second, instrumenting more interfaces results in more data, thus impairing scalability.

1.2.2 Scalability of instrumentation data

Traditional approaches to reduce the footprint of diagnostic data involve timely probing [5], instead of instrumenting all APIs. In addition, tools often employ reduction in data volume through compression. However, this merely postpones the problem of analyzing the data by writing compressed yet complete traces to the disk. This approach would not be better for long running applications which is the norm for HPC. Hence, it is imperative to reduce data volume through in-situ analysis rather than mere compression. In order to be more performant, applications using online techniques need to be capable of analyzing data in realtime, i.e. at a rate comparable to that at which data is generated. This can be realized by leveraging the

knowledge of the user for instrumenting interfaces that are problem-specific.

1.2.3 Customizable instrumentation

To aid the instrumentation of MPI programs, the MPI Standard [7] defines the PMPI profiling layer. Tracing or profiling tools that are developed for MPI programs can make use of the PMPI layer to seamlessly instrument the communication and computation routines of the application. By intercepting MPI APIs at the profiling layer, tools can perform tracing or profiling of functions and can ensure scalability by compression of data. However, for non-MPI APIs, contemporary tools provide little to no support. As a result, developers have to either rely on different tools to instrument such APIs or implement their own home-grown tools that cater to both MPI and non-MPI APIs alike. It is thus necessary that tools provide these customized tracing capabilities so that users can leverage them to instrument non-MPI APIs (in addition to MPI APIs through the PMPI layer) so as to localize inefficiencies.

1.2.4 Maintainability of codes

In recent years, HPC application codes are becoming increasingly large and multi-scalar as more intelligent modules are embedded into applications like end-to-end IO pipelines [26]. Also, with hybrid architectures, applications are increasingly comprised of several logically separate modules that need to co-exist seamlessly. It is often desired to segregate code into well-defined modules so as to eliminate cross-cutting concerns. Common strategies, e.g. Object-Oriented Programming, are not adequate enough because certain concerns like logging and tracing cannot be engineered efficiently with reduced scattering of code across modules. Aspect-Oriented Programming (AOP) [11] aims to solve this problem of "code tangling" by refactoring cross-cutting concerns into aspects that are executed at certain points in the code. A tool framework that supports the instrumentation of APIs at defined interfaces can help realize the AOP paradigm by permitting the execution of code at the prologue and epilogue of interfaces, thus reducing cross-cutting concerns.

1.3 Hypothesis

With the advent of the exascale era, debugging and performance analysis of massively parallel programs is becoming an increasingly arduous task as tools suffer due to scalability issues. This is predominantly because tools instrument interfaces exhaustively, which are possibly large enough to mask the anomalies. We aim to address this limitation in this thesis. Hence, the hypothesis of this thesis is:

Tool scalability can be vastly improved by providing users the ability to instrument a subset of interfaces of their choice and also allow for performing in-situ analysis of the collected performance data. Such a framework can be utilized for innovative performance analysis and can accelerate the debugging of HPC codes. As an added benefit, such a framework will help realize the AOP paradigm thus improving code readability.

1.4 Contributions

We propose to combat this problem of trace scalability via ScalaJack, a novel tool that supports problem-specific extraction and on-the-fly reduction of data through analysis. ScalaJack is realized above ScalaTrace [17], a tracing tool for MPI programs that ensures scalablity by exploiting the repetitive nature of timestep simulations of SPMD programs. The ScalaJack framework makes the following contributions.

1. ScalaJack supports dynamic customizable instrumentation by exposing APIs to the user for defining custom events to be part of the trace. Such custom events will now co-exist with MPI events and will be compressed using the intra-node and inter-node compression routines of ScalaTrace. This results in near-constant trace file sizes that preserve the underlying communication structure of the program.

2. ScalaJack supports in-situ analysis for diagnostic data reduction. Users can again utilize APIs to register these analysis entities with the respective reduction routines. These entities are associated with the events in a program and are subject to the intra-node and inter-node compression routines of ScalaTrace, during which the appropriate reduction routines are invoked. With ScalaJack, users can thus realize what we term as *active analysis tracing.*

3. ScalaJack supports manual instrumentation of custom events by exporting APIs that the user can use to mark sections of code as prologues and epilogues to events. On the other hand, ScalaJack also supports automatic instrumentation of such custom events through a preloader tool that auto-generates code at compile time from a specification file.

4. Extensions developed for ScalaJack are aspect-oriented and, hence, can reduce cross-cutting concerns in a program's source code thereby improving code readability and maintainability.

Evaluation of ScalaJack with typical use cases from HPC applications shows that ScalaJack results in scalable trace file sizes with increasing number of tasks with minimal additional overhead. Aspect-oriented analysis of the resulting codes also suggests a significant decrease in the scattering of cross-referenced data structures across different modules.

1.5 Organization

This dissertation is organized as follows. Chapter 2 outlines ScalaTrace, a tracing tool for MPI programs on top of which ScalaJack is realized. Also, since ScalaJack helps segregate code into aspects, a brief introduction into aspect-oriented programming and typical usages in HPC applications is discussed. Chapter 3 describes a high level design of ScalaJack with an example of how a typical application can use it. Following this, we discuss the implementation of ScalaJack, a framework supporting custom events and user callbacks. We then discuss the evaluation of ScalaJack in Chapter 4 and the related work in this realm in Chapter 5. We discuss how ScalaJack can be improved in the future in Chapter 6 and we summarize our work in Chapter 7.

Chapter 2

Background

This chapter summarizes the capabilities of ScalaTrace, a scalable and novel tracing tool for MPI environments on top which ScalaJack's customizable instrumentation and user callback frameworks are realized. We summarize the compression mechanisms of ScalaTrace that are reused by ScalaJack to ensure scalability. Also, a brief background on aspect-oriented programming and design is provided which is realized through the user callback framework within ScalaJack.

2.1 ScalaTrace

ScalaTrace [17] is a state-of-the-art scalable parallel communication tracing library for MPI programs. It utilizes the PMPI profiling layer to intercept MPI events and trace them as part of a program's execution. ScalaTrace achieves near constant trace sizes by employing novel techniques for intra-node and inter-node compression. Intra-node compression is achieved by identifying loops within a program and compressing events that form the loop, while inter-node compression is performed on events across nodes as part of *MPI_Finalize*. ScalaTrace employs structures known as RSDs and PRSDs to represent events in a loop as constant size trace logs. An RSD is represented as a tuple $< length, event_1 ... event_n >$ and a PRSD can represent multiple RSDs in nested loops. For instance, the MPI program shown in Figure 2.1 will correspond to the PRSD $< 10, RSD_1, MPI_Barrier >$ where RSD_1 corresponds to $< 10, MPI_Send >$.

ScalaTrace achieves lossless tracing by storing highly scalable traces through an elastic data representation [23] where data elements can start as scalars, morph into vectors in time and finally transform into histograms. The fact that parameters to MPI events are also recorded through this elastic data representation enables ScalaTrace to produce scalable yet lossless trace files. ScalaTrace also includes a replay engine with support for non-deterministic replays when

```
... // preceding events
for(i=0; i<10; i++) {
  for (j=0; j<10; j++) {
    MPI_Send(...);
  }
  MPI_Barrier();
}
... // following events
```

Figure 2.1: Nested MPI events

histograms are employed allowing events to be replayed without original program code.

ScalaJack reuses the compression algorithms of ScalaTrace but augments and extends it by introducing APIs accessible to the user. Users can thus define their own custom events specific to a program and also register specific callbacks for performing in-situ analysis of live data.

2.1.1 Intra-node compression

ScalaTrace performs intra-node compression by identifying loops in a program and storing them as PRSDs. This is done by maintaining the events in a queue. When an event is added to the trace, it is identified as the *target_tail* and a search is performed from the back of the queue to identify the presence of the same event. This identified event becomes the *match_tail*. The event immediately following the *match_tail* becomes the *target_head* and a search is performed again to identify the *match_head*. Once the head and tail of the *match* and *target* iterators have been identified, ScalaTrace performs an element-by-element comparison to identify if two ranges are indeed the same, which is true in case of a loop. This is shown in Figure 2.2. Here, *Send* is identitified as the head of the two iterators and *Barrier* is identified as the tail. The next version of ScalaTrace, ScalaTrace II [23] takes this approach further by allowing loops to be slightly different. This is done by introducing dummy events in the *match* and *target* queues and compressing them accordingly. This gives better scalability as iteration-specific events are properly identified as part of the loop and compressed.

In order to perform efficient intra-node compression, ScalaTrace employs different techniques, which are discussed below.

1. **Calling Sequence Identification:** To distinguish between different execution of events, ScalaTrace employs a signature associated with each event. This signature is computed by walking the stack backward.

2. **Recursion-Folding Signatures:** To prevent an explosion in size of the stack signature

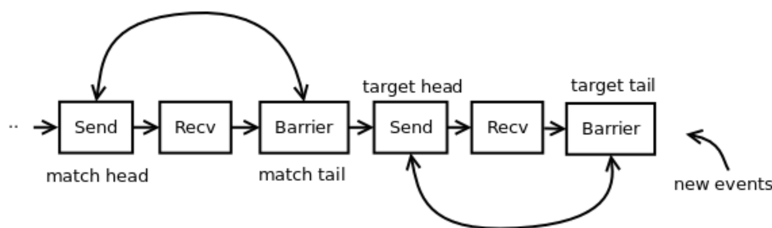

Figure 2.2: ScalaTrace's Intra-node compression

for recursive invocations, ScalaTrace identifies recursive calls and folds them to their first occurence.

3. **Location-independent Encodings:** ScalaTrace keeps track of the parameters to a MPI function and stores them as part of the trace. To ensure scalability, parameters like the *source* and *destination* are stored relative to the task's rank instead of the absolute value. For stencil codes, this helps in achieving highly scalable traces.

4. **Request Handles:** ScalaTrace compresses arguments like the request handles, which are opaque pointers to the MPI implementation's internal data structures, by accumulating them in a buffer and storing the indices to the buffer rather than the absolute values. This helps in indentifying repetitive patterns that are otherwise non-obvious.

2.1.2 Inter-node compression

In order to ensure scalability, ScalaTrace employs inter-node compression in a bottom-up fashion over a radix tree. In contrast to traditional approaches, this eliminates the need for separate trace files for each task, which would increase the IO bandwidth and thus compromise performance. In addition, separate trace files result in linearly increasing disk space. The compression algorithm has two queues to work on - the master and the slave queues. RSDs and PRSDs are merged with each other when events match. As parameters to events can morph into vectors, the same events with different parameters on two nodes are merged as one, retaining a vector that holds the two parameters. The compression algorithm works by identifying a subsequence match between the master and slave queues. The two subsequences are then merged together to arrive at the new sequence. Events from the slave queue that are not merged are finally added to the master queue.

For inter-node compression, ScalaTrace employs the following techniques.

1. **Task ID Compression:** ScalaTrace stores each of the task IDs in a PRSD-style representation known as the *ranklist* representing a set of ranks. As part of the global trace,

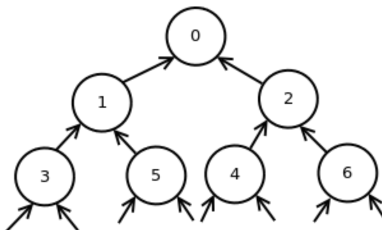

Figure 2.3: Inter-node radix tree compression

events are annotated with *ranklists* containing IDs of the task that executed the event. With such a recursive structure, large numbers of task IDs can be stored without compromising scalability.

2. **Radix Tree:** To ensure scalability, RSDs are merged over a binary tree as shown in Figure 2.3. Each node in the tree receives trace files from its two children, merges them with its own event queue and then sends the merged trace to its parent. Finally, *rank 0* retrieves the traces and performs IO to write the trace files.

In addition to tracing capabilities, ScalaTrace is equipped with benchmark generating and replaying capabilities with ScalaBenchgen [24] and ScalaReplay [23]. Also, trace files generated by ScalaTrace for lower numbers of nodes can be extrapolated to larger numbers of nodes with ScalaExtrap [25], thereby helping to analyse the scalability of the program and also helping in procurement of supercomputers.

2.2 Aspect-Oriented Programming

Aspect-oriented programming is a software engineering technique that aims to reduce the effect of cross-cutting concerns in a program by separating ”tangled” code segments. An *aspect* is defined as a piece of code that cannot be cleanly encapsulated in a generalized procedure. These are typically cross-cutting concerns that cut at specific *pointcuts* in a given program's code. Programs written in the aspect-oriented paradigm aim to cleanly separate components and aspects from each other by providing mechanisms that make it possible to abstract and compose them to produce the overall system [11]. Some typical aspects identified in applications are logging, timing etc. With HPC applications, this list can be extended to include performance evaluation, visualization, load-balancing, fault-tolerance provision, memory (re)allocation, synchronization primitives and other optimizations [11].

With aspects, better code is realized by moving the cross-cutting concerns from the original *component* of the application to the aspects themselves. An *advice* is a functionality that is

```
...
record_start_time();
compute_fd();
record_stop_time();
...
```

Figure 2.4: Cross-cutting concerns in a traditional application

```
...
compute_fd();
...

aspect_compute_fd() {
  before:
    record_start_time();
  after:
    record_stop_time();
}
```

Figure 2.5: Application after aspects refactoring

executed at the region of a *pointcut* in the program. Each aspect can be executed before or after or around the *advice* by suitably wrapping the *advice*. Traditional aspect-oriented frameworks like AspectJ [12] and AspectC [3] rely on providing run-time or compile-time support for aspects in Java and C. In ScalaJack, we rely on supporting aspects through run-time constructs like PMPI for MPI calls and dynamic pre-loading otherwise (or manual tagging of the prologue and epilogue). More so, ScalaJack is aimed to be an aspect-oriented framework to perform in-situ data analysis with trace analysis specifically geared for HPC codes. To the best of our knowledge, such a tool has no precedence.

To illustrate, Figure 2.4 shows a code snippet with a computation routine that is "peppered" with timing code before and after with timing code. With respect to the main objective of the component that performs the computation, the timing routines are cross-cutting concerns. Hence, the region before and after the call to the *compute_fd()* advice are pointcuts and their respective code can be refactored into aspects. Figure 2.5 shows the same program where the timing routines are refactored into aspects for the advice *compute_fd()*. With code being refactored across several sections of the code, it can be seen that aspect-oriented programming can yield codes with better maintainability, readability and reliability.

Chapter 3

Design and Implementation

This chapter outlines a high-level overview of the design of ScalaJack and also how developers are envisioned to make use of ScalaJack in their applications. This is followed by a discussion on the implementation of the custom event framework and the custom user handler framework in ScalaJack.

3.1 High-level design

The design of ScalaJack is shown in Figure 3.1. MPI events from each task are traced through the PMPI wrapper while users use APIs from ScalaJack to register and trace arbitrary functions. Each event traced by ScalaJack is wrapped with a prologue and epilogue for performing tracing in addition to invoking embedded aspects registered through the callback framework. Events within a task are compressed on-the-fly by exploiting the loop constructs in the program while a further phase of compression is performed via inter-node compression over all the tasks. This highly compressed trace is thus scalable with the number of processes.

A typical application workflow using ScalaJack would resemble Figure 3.2. A parallel application uses the customizable instrumentation capabilities of ScalaJack to trace and instrument MPI routines or arbitrary functions in their code, in addition to performing in-situ reduction (through analysis) of data generated through instrumentation. This data is co-located with the appropriate event blocks and stored as RSDs and PRSDs in a scalable fashion, preserving the structure of the program. Correlating data to the events in the trace provides better insight into diagnosing problems, thus helping in identifying even the subtlest of performance anomalies. Additionally, the application can use the framework to perform other tertiary tasks that are identified as cross-cutting concerns, e.g., visualization, so that better code modularity is achieved.

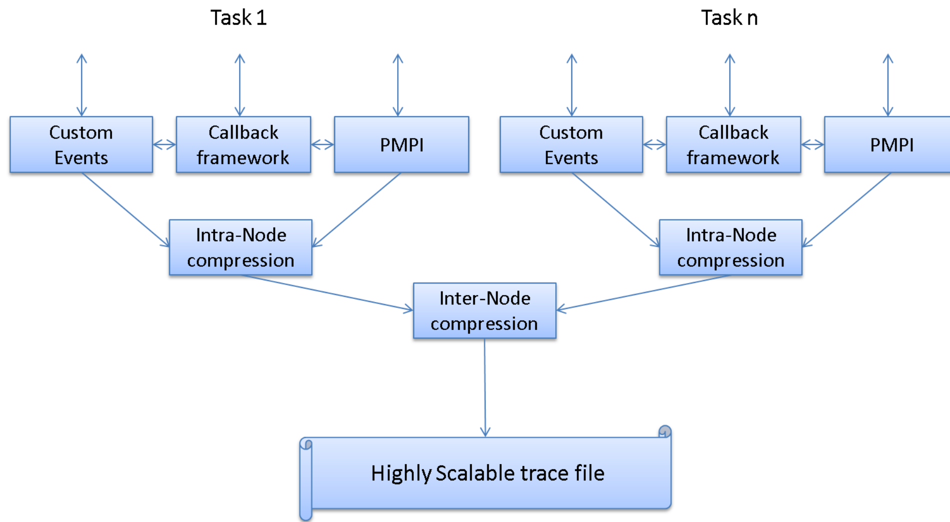

Figure 3.1: ScalaJack's High-level Design

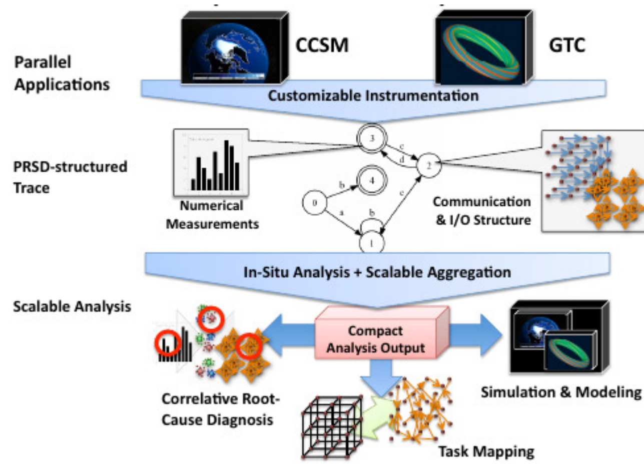

Figure 3.2: Typical Application workflow with ScalaJack

```
...                 // preceding events
MPI_Send(...);      // communication
...                 // computation
MPI_Recv(...);      // communication
...                 // computation
MPI_Allreduce(...); // communication
...                 // following events
```

Figure 3.3: Communication and Computation phases in an MPI program

3.2 Custom Event Framework

ScalaTrace originally supports three levels of tracing, viz. tracing of events with no compression, intra-node compression of events and intra-node with inter-node compression. The events are MPI routines, characterized as communication points and instrumented via the PMPI layer. Phases between successive MPI events are considered computation points (see Figure 3.3). ScalaJack implements a custom-event registration framework that a user can use to extend ScalaTrace's scalable compression algorithms to trace arbitrary events in their program. As part of this work, a new level of tracing is also introduced where only the *MPI_Init* and *MPI_Finalize* calls are instrumented to signal the start and end of tracing while no other MPI events need to be instrumented. This, along with the custom event registration APIs, enables the user to localize problems in their applications by instrumenting functions at will with trace and analysis capabilities.

In theory, any two arbitrary functions can be tagged as the Init and Finalize routines for marking the start and end of tracing. This even supports tracing in non-MPI environments. However, in such a system, it would be the user's responsibility to provide alternatives for *MPI_Comm_rank*, *MPI_Comm_size* and *MPI_Barrier* because these APIs are critical in ScalaJack's algorithms for identifying processes and identifying the world for cross-node data reduction. In this paper, to facilitate discussion, it is assumed that ScalaJack is utilized in either a pure MPI-only environment or a hybrid MPI environment involving GPUs or CPU threads.

ScalaJack's custom event framework introduces two functionally distinct APIs - one for registering the event with the framework and the other for marking the prologue and epilogue of an event. These are explained further.

3.2.1 Registering custom events

Users can register events with ScalaJack's framework with the *SJ_register_custom_event* API to obtain an event code unique to the event. The enumeration of these event codes are mapped

```
... // preceding events
for (i=0; i<5; i++) {
    MPI_Send(...);
    ...
    MPI_Recv(...);
}
MPI_Send(...);
... // following events
```

Figure 3.4: MPI events in a loop

onto a different but related namespace from those of MPI events that were originally traced. ScalaTrace internally maintains a control block for every custom event registered, indexed with the respective event code. While registering custom events, users can also supply optional *flags* that control ScalaJack's handling of the event.

One such flag is *SJ_CE_FLAG_NO_SIG*. This flag instructs ScalaJack to refrain from generating signatures, which identify a specific invocation of a function. ScalaJack, by default, uniquely identifies each invocation of an event with a stack signature computed by walking the stack backtrace. This signature is used to identify events that form a loop and to compress them as part of the intra-node compression step. Such a scenario is shown in Figure 3.4, where the two invocations of *MPI_Send* are considered unique. While the first is part of the RSD that defines the loop, the second is part of the PRSD that includes the loop RSD and PRSDs from preceding and following events. However, with the flag present, no stack signatures are generated for custom events to identify unique invocations. This may be useful for the user in an application that resembles Figure 3.5, where the three invocations of the *user_function* are identified to closely resemble each other will be compressed together. This may be preferable in collecting timing or performance analysis for events that are scattered about the program. Scalable compression can be achieved by aggregating such events irrespective of their origin of invocation. Since walking the stack is expensive, developers can switch this feature off, if they deem it unnecessary for the application.

3.2.2 Invoking custom events

Custom events registered by the user are mapped to user-supplied arbitrary functions. This is done by prefixing and suffixing the function call with the APIs *SJ_custom_event_pre* and *SJ_custom_event_post*, respectively. These prologue and epilogue functions for custom events are synonymous to those for MPI events instrumented via the PMPI layer. They are discussed in more detail in Section 3.3.

```
... // preceding events
user_function(...);
...
user_function(...);
...
user_function(...);
... // following events
```

Figure 3.5: Successive Custom events

With the original design, developers have to manually pepper each of their custom event with a call to the prologue and epilogue. This constraint can be lifted by utilizing the dynamic preloader bundled with ScalaJack. Users can optionally provide a specification of the functions to be instrumented and ScalaJack can auto-generate code that can wrap the original function call.

It should be noted that when custom events are used with MPI events, registered custom events will co-exist with traditional MPI events, i.e., they will be written as part of the same trace file. In such a scenario, the provision of the flag *SJ_CE_FLAG_NO_SIG* might not guarantee the compression of custom events outside of loops as shown in Figure 3.6. Since signatures for the two *MPI_Send* events will be different, the match and target queues will not unify and, hence, ScalaTrace's compression algorithms will refrain from matching the custom events even though no signature is associated with them. In such a scenario, the user can optionally utilize the level of tracing where only *MPI_Init* and *MPI_Finalize* are traced.

Another challenge with allowing the user to provide custom events is the phenomenon of nested custom events. As shown in the Figure 3.7, functions that are registered as custom events can internally call other custom events. Thus, before the epilogue of *custom_event_1* is reached, the prologue of *custom_event_2* is encountered. One way to tackle this would be to buffer nested events till all the epilogues are encountered, e.g., by storing them in a list. However, this does not reflect the actual execution of the program and the accretion of such nested events might hamper scalability in applications that exhibit highly recursive or nested behavior. ScalaJack tackles this problem by identifying the prologue and epilogue as custom events themselves when it detects the presence of nested custom events. Thus, the trace contains the prologues and epilogues as communication events while custom events are recorded as computation phases. In any case, the event sequence thus recorded resembles the execution behavior of the program.

```
... // preceding events
MPI_Send(...);
...
user_function(...);
...
MPI_Send(...);
...
user_function(...);
... // following events
```

Figure 3.6: Custom events with MPI events

```
...
SJ_custom_event_pre(event1);
custom_event_1(...);
SJ_custom_event_post(event1);
...

void custom_event_1(...)
{
  ...
  SJ_custom_event_pre(event2);
  custom_event_2(...);
  SJ_custom_event_post(event2);
}
```

Figure 3.7: Nested custom events

3.3 User Callback Framework

User callbacks provide hooks at destined points in the call graph of a program. These hooks can be utilized by users to perform in-situ data analysis on the data collected as part of the custom events. In addition, these callbacks realize an aspect-oriented framework which developers can utilize to implement efficient and cleaner code by separating cross-cutting concerns as aspects from the main algorithmic concern at hand.

As mentioned before, ScalaJack utilizes the PMPI layer to instrument MPI events. Each event is prefixed with a prologue and postfixed with an epilogue. The prologue is responsible for creating the event control blocks in ScalaJack. Since MPI events are marked as communication points in ScalaJack, a prologue generally marks the end of a computation event and the beginning of communication. In contrast, the epilogue consists of routines that append the

events into the trace and perform intra-node compression on-the-fly, if enabled. On the same lines, the epilogue serves as the end of the communication and the beginning of computation.

The user callback framework extends this connotation by allowing developers to register callbacks at the prologue and epilogue, which serve as *pointcuts* at their code. In addition, developers can optionally feed data into the trace from the callbacks pertaining to the computation or communication phase of the event. As part of the epilogue, the analysis of data is effected during on-the-fly compression of the trace.

Like the custom event framework, ScalaJack's user callback framework also comprises of two APIs - one for registering callbacks and the other for user-provided data analysis routines.

3.3.1 Registering callbacks

ScalaJack implements a *Stat* (short for Statistics) class, which is instantiated twice per event. The two *Stat* objects associated with the MPI events govern the statistics for the computation phase (before the event) and the communication phase (of the event itself). The framework allows users to register these callbacks in two ways. Users can either extend the *Stat* class by providing alternatives for the *start* and *end* methods, or they can register two methods that mark the start and end of the collection of statistics via the *SJ_register_user_jack* API. In the latter scenario, ScalaJack internally associates a *Stat* object with the registered *start* and *end*. As part of the prologue, the end of the computation phase is identified by calling the *end* method of the *Stat* object for computation; in addition, the start of the communication phase is identified by calling the *start* method of the *Stat* object for communication. The epilogue invokes the *end* of the computation phase and the *start* of the computation phase.

The registration of callbacks can optionally be augmented with flags similar to the custom events. One such flag is *SJ_UJ_FLAG_CALLBACK*, which can be used to only perform auxiliary tasks (as with aspects) as part of the callbacks without any addition into the trace. In this mode, users can override the *callback* method of the *Stat* class. Such user callbacks are invoked only once for every prologue and epilogue. With the callback mode, there is no distinction between communication and computation events as with the default mode; nonetheless users can distinguish between the prologue and epilogue.

3.3.2 Compression of user data in callbacks

ScalaJack augments ScalaTrace by providing data analysis capabilities within the user callback framework, thus allowing users to perform in-situ data analysis within trace analysis. While ScalaJack tags all data originating from the *end* method as numeric and compresses them as a *Histogram*, users can provide their own compression routines by extending the *ValueSet* class and implementing the *jadd*, *jmerge*, *pack* and *unpack* routines. While the *jadd* and *jmerge*

routines are used for addition to the trace and reduction, respectively, the *pack* and *unpack* routines serialize and de-serialize the user-specified data for transfer between tasks for inter-node compression. As with the registration of callbacks, users can register callback functions with ScalaJack's framework, instead of extending the class and ScalaJack will internally instantiate a *ValueSet* object with the appropriate merge routines. With this capability, users can identify appropriate *pointcuts* in their program, associate them with specific data and optionally feed the data to the trace file ensuring scalability through online compression.

While most aspect-oriented frameworks deal with mapping aspects to specific events, the framework realized through ScalaJack does not make this distinction. Aspects are applicable to every event that is traced with ScalaJack. The distinction between event-specific aspects can be made by light-weight filter predicates with-in an aspect. To enable this, developers have access to the event objects of the *pointcuts*. This enables them to execute aspects for specific events or specific conditions, e.g. to access the send count or destination of MPI events. Another advantage of tying aspects to all events is that users also have access to the entire trace queue consisting of all the events traced thus far (but in PRSD-structurally compressed form). This is beneficial in scenarios where users need to perform analysis on the trace as a whole. For instance, tasks can compute their similarity with others in the system in k-clustering, thereby grouping them based on a similarity metric.

3.4 Preloader

To aid automatic instrumentation, ScalaJack features a built-in preloader that autogenerates code at compile time to intercept function calls in a program. A program written in a language like C can invoke routines that are either statically compiled within the binary (through a statically linked library) or can be dynamically looked up by the dynamic linker. The preloader is designed to support the interception of both such routines.

In contrast to other tools that implement interception capabilities, ScalaJack's preloader does not rely on run-time binary interception through tools like Pin [14] or Dyninst [21]. This is because such tools introduce significant run-time overhead on the system, which could pose a problem at exascale and can potentially introduce perturbations masking the original problem. Instead, the preloader relies on compile-time strategies to generate two different libraries, one of which is statically linked with the application and the other dynamically preloaded using the *LD_PRELOAD* facility of the dynamic linker. The preloader relies on this dual approach to intercept statically compiled and dynamically linked routines because of the lack of support for dynamically preloading weak symbols that are statically linked with the application.

The design of the preloader is shown in Figure 3.8. It consists of the compile time generator tool, *sjmake*, which generates code for intercepting functions that can then be used to generate

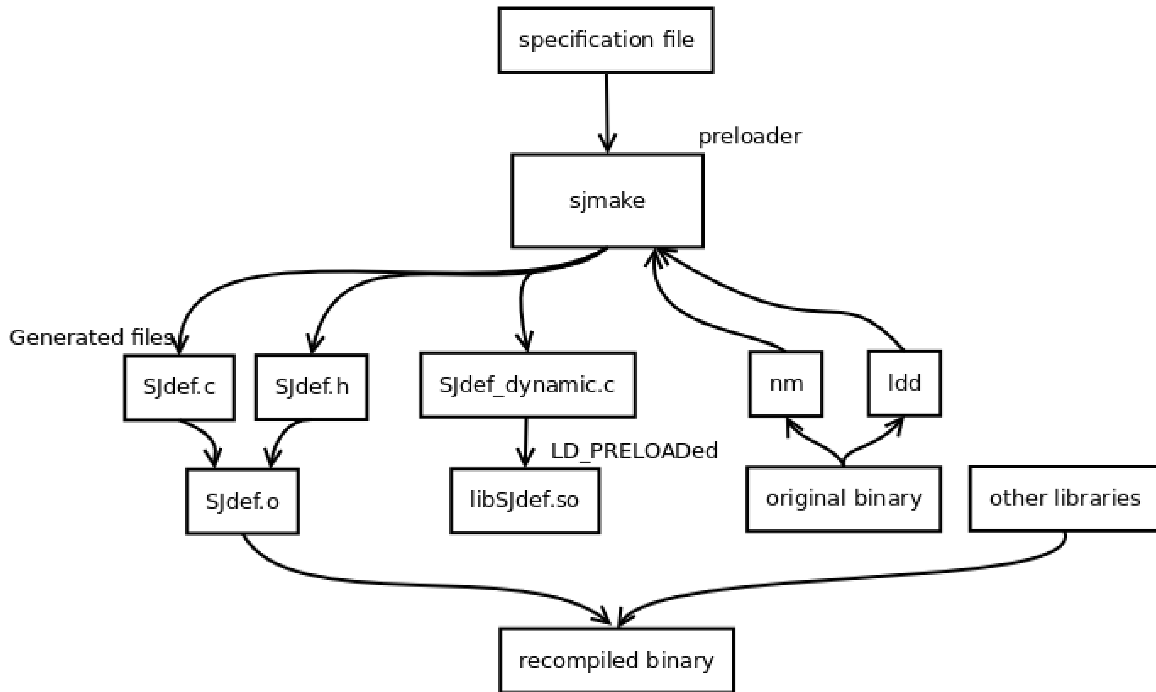

Figure 3.8: The design of the preloader

libraries for the program to use. This tool generates code based on a user-provided specification file enumerating the API calls to be instrumented.

3.4.1 Specification file

A sample specification file for the preloader is shown in Figure 3.9. The specification file has two distinct sections, the *sj-include* and the *sj-definitions*. Code added to the *sj-include* section is duplicated in the generated file. This can be utilized by the user to include header files or define data types that are part of the functions to be intercepted. On the other hand, the *sj-definitions* section consists of a list of function prototypes that are to be intercepted. The specification file does not need to identify a function as statically linked or dynamically linked as the preloader automatically ascertains this from the compiled binary.

As shown in Figure 3.9, by specifying definitions of an MPI function, users can instrument a subset of MPI routines in conjunction with the custom tracing level of ScalaJack, thereby tracing only the MPI events that are appropriate to the problem at hand.

```
# a sample specification file for sjmake
sj-include:
#include <mpi.h>
#include <stdlib.h>

sj-definitions:
int myfoo();
int mybar(int x, ...);
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm);
..
```

Figure 3.9: A sample specifiction for the preloader

3.4.2 Statically linked routines

The preloader, by default identifies all the routines defined in the specification file as statically linked ones. Statically linked routines are intercepted by defining the function as a weak symbol in the application and then generating a non-weak version as part of the static library. When the generated library is linked with the application, the weak symbol gets resolved to the overloaded function. To ascertain the original address of the weak symbol which needs to be invoked, the preloader requires the *nm*, *ldd* tools along with the original version of the binary. Since the preloader assumes that functions are by default statically linked, it tries to lookup the symbol address from the executable using the *nm* utility. Symbols that are undefined are assumed to be dynamically linked and are handled differently. As part of the code generation, the preloader invokes the original definition of the function by invoking the address looked up by *nm*. These addresses are statically encoded in the generated code during compile time and thus pose no overhead at run-time.

The generated code for static functions resembles Figure 3.10. As part of the static generation process, two files, *SJdef.c* and *SJdef.h* are generated. While the former contains the generated code for performing interception, the latter contains definitions of the functions as weak symbols for inclusion in the application. The library *SJdef.o* generated from the *SJdef.c* file is then used to generate the recompiled application.

3.4.3 Dynamically linked routines

Those definitions that are undefined in the binary are considered as dynamically linked and are handled differently. The preloader utilizes the run-time dynamic linker's capabilites through *libdl.so*. The search is performed over all libraries that are dynamically linked with the applica-

```
int myfoo()
{
  int ret;

  SJ_custom_event_pre(1300); // 1300 is the event code

  // nm_table has addresses of the original routines
  ret = (int()(*)())(nm_table[0])();

  SJ_custom_event_post(1300); // 1300 is the event code

  return ret;
}
```

Figure 3.10: Generated code for statically compiled routines

tion which are looked up by invoking the *ldd* utility. This dynamically looked up value is cached as part of the function routine so that subsequent calls to the functions utilize the cached value instead of looking up the symbol again.

The generated code for dynamically linked routines is shown in Figure 3.11. For dynamic routines, the preloader generates the *SJdef_dynamic.c* file which can be compiled to the library, *libSJdef.so*. This library is dynamically preloaded using the *LD_PRELOAD* functionality of the linker. As a result, calls made to preloaded functions are intercepted at the generated library which internally calls the original definition.

In addition to these, the preloader also generates bookkeeping routines for registration of custom events with ScalaJack. This registration routine is defined as a weak symbol and ScalaJack automatically invokes this registration function as part of the epilogue of *MPI_Init*.

```
int myfoo()
{
  int ret;
  static void *fn = NULL;

  SJ_custom_event_pre(1300); // 1300 is the event code

  if (fn) {
    ret = (int()(*)())(fn)();
  } else {
    for (item: ldd_table && !fn) {
      fn = lookup_dlsym(item);
    }

    ret = (int()(*)())(fn)();
  }

  SJ_custom_event_post(1300); // 1300 is the event code

  return ret;
}
```

Figure 3.11: Generated code for dynamically linked routines

Chapter 4

Evaluation

This chapter evaluates the viability of ScalaJack in traditional HPC applications. We assess the scalability of ScalaJack via the traces generated by utilizing ScalaJack's custom event framework. In addition, the overhead incurred in using ScalaJack over a naive implementation is studied. We evaluate ScalaJack by refactoring several case studies of typical HPC applications to utilize ScalaJack's aspect-oriented callback framework. Tasks that are tangential to the program are refactored as part of these callbacks. As a result, cross-cutting concerns are removed from the main component of the program, thus improving readability and maintainability.

4.1 Evaluation Setup

All experiments were conducted on our ARC cluster with each node boasting two AMD Opteron 6128 processors with 8 cores each (16 per node) and InfiniBand interconnects between them. The number of nodes chosen for experimentation was a maximum of 16, with each node handling up to a maximum of 16 tasks. Experiments involving execution times and trace file sizes were averaged over a maximum of 10 runs.

4.1.1 Aspect-oriented metrics

Since ScalaJack helps remove cross-cutting concerns in the code, the amount of code related to a concern that is scattered is reduced. To quantify the improvement of using ScalaJack over a naive implementation, with respect to the code footprint, we utilize the degree of scattering *(DOS)* and degree of focus *(DOF)* metrics from [8]. Concentration *(CONC)* measures how many of the source lines related to a concern s are contained within a component t (e.g., file, class, method intending to a specific task), i.e.,

$$CONC(s,t) = \frac{SLOC_{t,s}}{SLOC_s}$$

where $SLOC_{t,s}$ is the number of source lines of code *(SLOC)* in component t related to concern s, and $SLOC_s$ is the *SlOC* in all of concern s. It should be noted that *SLOC* excludes comments, blank lines and annotations for concern assignment. However, the drawback of *CONC* is that it does not reflect the amount of scattering of a concern's code and does not allow for different concerns to be compared. This is covered by the degree of scattering *(DOS)* metric defined by

$$DOS(s) = 1 - \frac{|T| \sum_t^T \left(CONC(s,t) - \frac{1}{|T|}\right)^2}{|T| - 1}$$

where T is the set of components and $|T| > 1$. *DOS* is a normalized factor between 0 (completely localized) and 1 (completely delocalized). Thus, a reduction in *DOS* is an indication of less scattering of code across components.

Degree of Focus *(DOF)* is a dual to the DOS metric and captures how focused a component is. Dedication *(DEDI)* is defined as

$$DEDI(t,s) = \frac{SLOC_{t,s}}{SLOC_t}$$

where $SLOC_{t,s}$ is the number of source lines of code *(SLOC)* in component t related to concern s, and $SLOC_t$ is the *SLOC* in all of component t. Again, a better metric would be the normalized degree of focus *(DOF)*

$$DOF(s) = \frac{|S| \sum_s^S \left(DEDI(t,s) - \frac{1}{|S|}\right)^2}{|S| - 1}$$

where S is the set of concerns and $|S| > 1$. *DOF* is also a normalized factor between 0 (completely unfocused) and 1 (completely focused). Thus, an increase in *DOF* is desired as it is indicative of reduction in scattering and increase in focus.

4.2 Performance analysis

One of the most frequently identified aspects in any program is performance analysis. Developers typically want to identify the performance characteristics of specific regions of their code. In most HPC applications, distinct regions of computation and communication can be identified, and it is often desired to collect performance metrics related to the phases. We evaluate ScalaJack's viability with the IS benchmark of the NAS Parallel Benchmark suite. IS sorts

```
MPI_Init();

PAPI_library_init();

initialization();

PAPI_start();
create_random_seq();
add_to_trace(PAPI_read());
..
  MPI_Reduce();
  add_to_trace(MPI_REDUCE);
..
PAPI_shutdown();

MPI_Finalize();
```

Figure 4.1: Outline of the IS benchmark

integers through a parallel implementation of bucket sort. As part of the benchmark, each task generates a random number sequence from a seed based on the rank.

We illustrate ScalaJack's capabilities to support performance analysis aspects by choosing PAPI [15] to instrument the L1 data cache misses during the random sequence generation in addition to performing trace analysis on every MPI event in the program. We compare an implementation of the IS benchmark that uses ScalaJack with a naive implementation with tracing concerns around all MPI functions and performance analysis concerns around the random sequence generation step. We utilize the tracing level of ScalaJack where all MPI events are traced with custom events where both intra-node and inter-node compression is performed. Figure 4.1 shows the outline of the naive implementation of the IS benchmark. The code initializes the PAPI library, followed by an instrumentation of the random sequence generation routine of IS with PAPI APIs. The return value of this instrumentation routine is then added to the trace. To indicate the changes to perform tracing, a sample MPI routine, *MPI_Reduce* is shown with an API call following it, to add data to the trace. The ScalaJack version shown in Figure 4.2 differs from the naive implementation by utilizing PMPI wrappers to trace events (and compress them) while the PAPI APIs are invoked as part of the callback *StatPAPI* registered. These callbacks are invoked as part of the prologue and epilogue of the custom event associated with random number generation. This allows for separation of concerns and reusability of the PAPI statistics collection *Stat* framework.

Figure 4.3 compares the trace files generated with ScalaJack and that of the naive imple-

```
    MPI_Init();

    initialization();
    SJ_register_event(&ec);
    SJ_register_user_jack(StatPAPI);

    SJ_ce_pre(rand);
    create_random_seq();
    SJ_ce_post(rand);
    ..
      MPI_Reduce();
    ..
    MPI_Finalize();

  StatPAPI: rand
    start:
      PAPI_start();
    end:
      return PAPI_read();
```

Figure 4.2: Outline of the IS benchmark with ScalaJack

mentation. The trace file sizes shown are relative to the ones generated with $n = 4$ tasks. As can be seen from the graph, traces generated with ScalaJack are highly scalable with an increasing number of processors compared to the traces generated by the naive implementation. This is owing to the fact that ScalaJack employs intra-node (to compress loops) and inter-node compression to generate a single trace file, while the naive implementation performs no compression and generates traces for each of the tasks. We compare relative trace file sizes because, on an absolute scale, trace files generated with ScalaJack are larger by a factor of a few hundred bytes for lower values of n due to timestamp data that is also added to the trace. ScalaJack internally times every communication and computation phase of the program and stores them as histograms. This is utilized later by the replay engine and other tools like benchmark generators to create instances of the original program. [17][24].

To highlight the overhead incurred in using ScalaJack, we compare the running times of the two implementations of the IS benchmark. As shown in Figure 4.4, ScalaJack introduces very little overhead to the naive implementation's execution. To put it in a different perspective, Figure 4.5 shows the percentage overhead times of ScalaJack over the naive implementation. As it can be seen, ScalaJack introduces a performance overhead of about 0.07% for $n = 4$. There is substantial variability in the overhead of ScalaJack over the naive implementation since each

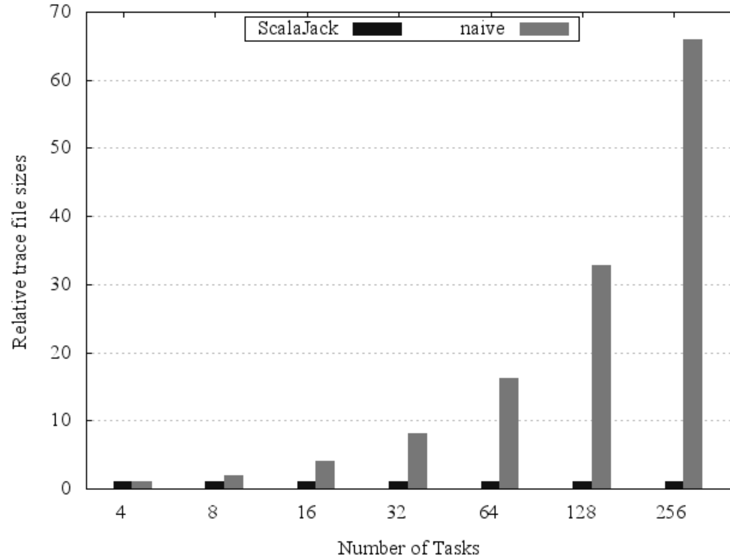

Figure 4.3: Trace file sizes with IS

task of the naive implementation performs IO to the parallel file system at *MPI_Finalize* to write n traces files for n nodes back to disk, each of which may be rather large (in the order of GBs depending on the number loop iterations). However, with ScalaJack, only *rank 0* performs IO to the file system after aggregating the traces from all its peers, i.e., a single file of rather moderate size (in the MBs) suffices.

Table 4.1 shows the improvement of using ScalaJack for separation of concerns over the naive implementation. For IS, the identifiable components are main and PAPI, where the main component implements the benchmark while the PAPI component implements the performance metrics collection routines. The concerns here are identified as perf and sort, where perf is the actual performance metrics collection API invoked at the pointcuts and sort is the rest of the main component that performs the sorting. The goal is to reduce the tangling of code between the two concerns and ScalaJack achieves this. This is reflected by the lower DOS score and a correspondingly higher DOF score for ScalaJack compared to the naive implementation.

4.3 Visualization and Load balancing

We evaluate the effectiveness of the ScalaJack framework on CLAMR [13] an adaptive mesh refinement solver developed at Los Alamos National Laboratory. CLAMR implements a cell-based shallow water code on MPI by computing the finite difference on AMR. As with AMR codes, CLAMR periodically refines the mesh and also performs load balancing across the nodes

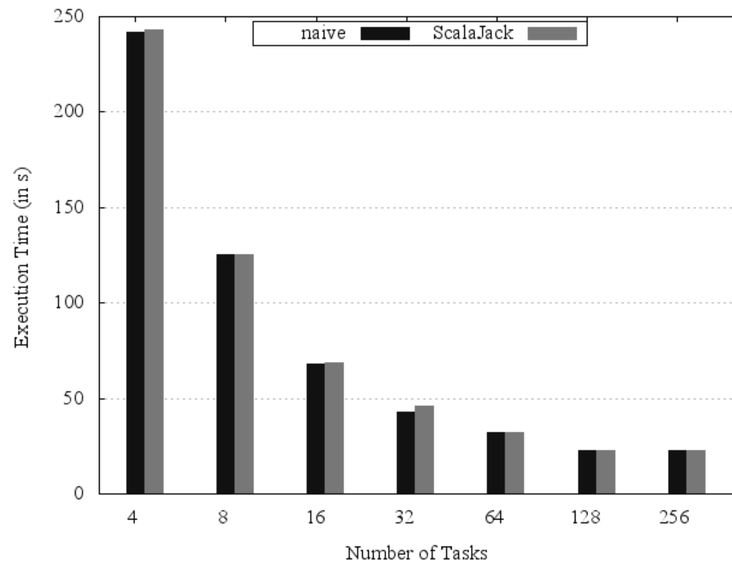

Figure 4.4: Execution times with IS

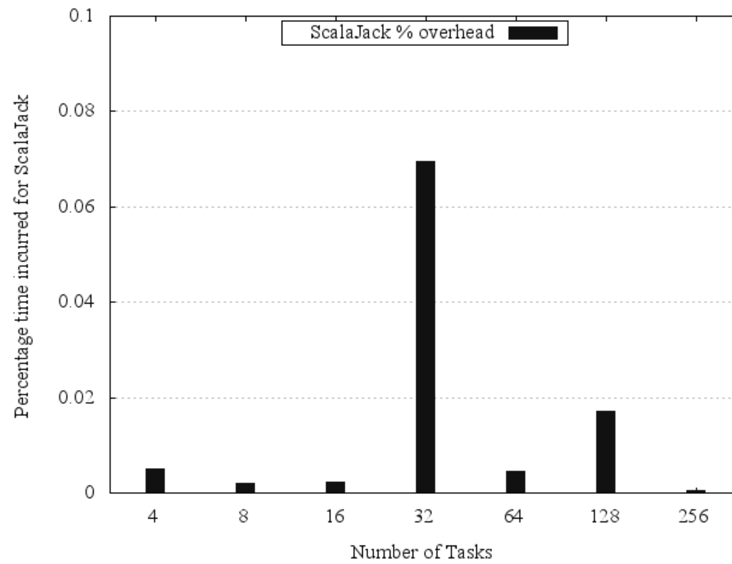

Figure 4.5: Percentage overhead with ScalaJack for IS

to redistribute the meshes. In addition, CLAMR performs OpenGL or MPE-based visualization to display the current state of the mesh.

Table 4.1: Aspect metrics for IS

	naive		ScalaJack	
	PAPI	main	PAPI	main
CONC(perf, t)	1	0.4777	1	0.0444
DOS(perf)	0.4992		0.0850	
	perf	sort	perf	sort
DEDI(main, s)	0.0588	0.9411	0.0057	0.9942
DOF(main)	0.7782		0.9770	

```
MPI_Init();

initial_setup();

visualize();
for (loop) {
  for (burst_loop) {
    calc_fd();
    refine_mesh();
    load_balance();
  }
  visualize();
}
stats_print();

MPI_Finalize();
```

Figure 4.6: Outline of CLAMR

Application codes like CLAMR have numerous conflicting concerns that can be effectively addressed using ScalaJack. A rough outline of the naive implementation of CLAMR is shown in Figure 4.6. As can be seen, tasks like visualization, mesh refinement, load balancing and printing of statistics are not part of the main concern at hand, i.e., computing the finite difference. An implementation of CLAMR with ScalaJack is shown in Figure 4.7. In this implementation, the various concerns that are tangential to the main concern at hand are refactored into the appropriate prologue/epilogue as shown. CLAMR was evaluated with the custom level of tracing where no MPI events are traced other than *MPI_Init* and *MPI_Finalize*, but custom events are

```
MPI_Init();

initial_setup();

ce_pre(loop);
for (loop) {
  ce_pre(burst);
  for (burst_loop) {
    ce_pre(fd);
    calc_fd();
    ce_post(fd);
  }
  ce_post(burst);
}
ce_post(loop);

MPI_Finalize();

StatViz: loop, burst
  before loop:
  after burst:
    visualize();

StatRef: fd
  after:
    refine_mesh();
    load_balance();

StatPrint: loop
  after:
    stats_print();
```

Figure 4.7: Outline of CLAMR with ScalaJack

traced. Also, custom events are configured to be created without the stack signature so as to reduce the trace footprint. Also, since no data is to be written as part of the callbacks, we register user callbacks with the callback mode flag. Since the goal with CLAMR is not tracing but rather refactoring tangential concerns into callbacks, we refrain from comparing trace sizes between the naive and the ScalaJack implementations. Instead, to assess the scalability, we compare the execution times of both the versions.

Figure 4.8 compares the overhead in using ScalaJack through the differences in execution

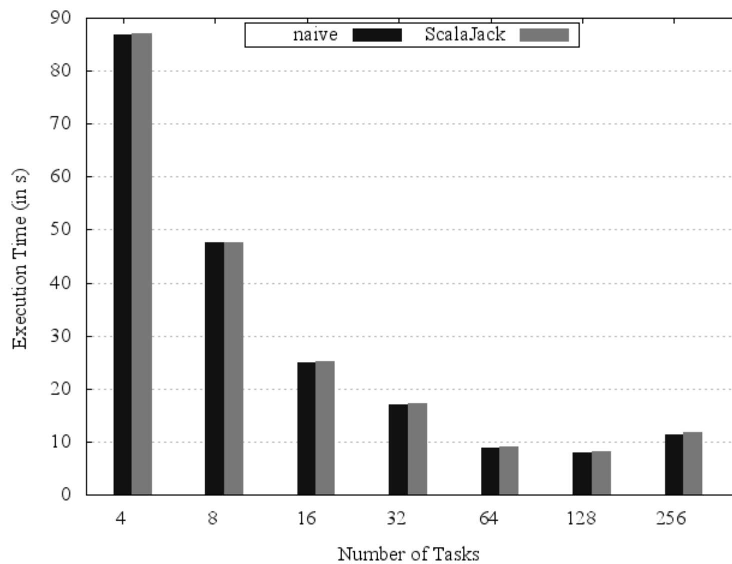

Figure 4.8: Execution times with CLAMR

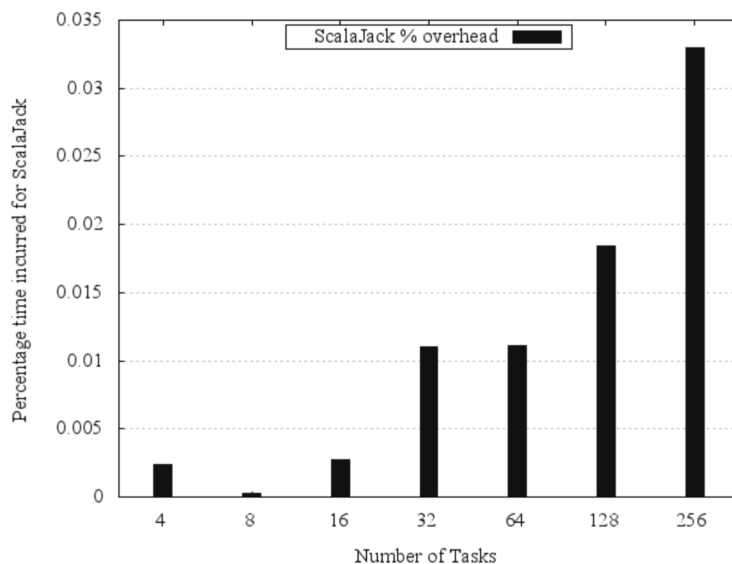

Figure 4.9: Percentage overhead with ScalaJack for CLAMR

time between the naive and the ScalaJack versions of CLAMR. As it can be seen from Figure 4.9, ScalaJack introduces an overhead of a maximum of 0.03% overhead. This is lower than that of IS because we utilize custom level tracing for CLAMR, which does not trace any MPI events.

Table 4.2 summarizes the improvements of using ScalaJack to eliminate concerns from

Table 4.2: Aspect metrics for CLAMR

	naive		ScalaJack	
	aux	main	aux	main
CONC(aux, t)	1	0.0739	1	0.0118
DOS(aux)	0.1369		0.0234	
	main	fd	main	fd
DEDI(main, s)	0.2708	0.7293	0.0540	0.9459
DOF(main)	0.2102		0.7955	

CLAMR. With CLAMR, the main component is the code that performs the finite difference, while all cross-cutting concerns are grouped as an auxiliary concern. With ScalaJack, all cross-cutting concerns are performed at the callbacks as part of custom events registered as shown in Figure 4.7. With CLAMR, the majority of the cross-cutting concern code was that of visualization because the *rank 0* task aggregates all mesh values from the other tasks for visualization. Since a major portion of the code is eliminated from the main component, we observe a better DOF score (and, consequently a lower DOS score).

4.4 Data analysis in-situ with trace analysis

As the final case study, we analyze ScalaJack's effectiveness with a MapReduce style application that can take advantage of the reduction capabilities of ScalaJack. TF-IDF is a data analysis metric used to assess the importance of a given term with respect to a document in a dictionary[20]. The two metrics involved are *term frequency tf(t,d)*, defined as the frequency of occurrence of a term t in a given document and *inverse document frequency idf(t,D)* in a set of documents D, defined as the inverse of the frequency of documents that contain a term t within a given dictionary of term. The TF-IDF metric is then defined by

$$tfidf(t, d, D) = tf(t, d) \times idf(t, D)$$

TF-IDF is a MapReduce style problem wherein a set of documents are initially mapped across a number of tasks and each task computes the *tf* and *idf* metrics separately followed by a reduction, which aggregates *idf* metrics. With such analysis problems, efficient reduction strategies that are scalable are required because a naive implementation might lead to bottlenecks and lower performance. Data analysis problems, such as TF-IDF, can exploit the internal reduction logic of ScalaJack otherwise utilized by inter-node compression. This can be done by

```
MPI_Init();

read_documents();
pre_processing();

compute_tf();

compute_idf_local();

construct_comm_tree();
receive_trace(lchild);
deserialize_idf_table();
receive_trace(rchild);
deserialize_idf_table();

reduce_idf();

serialize_idf_table();
send_trace(parent);

MPI_Finalize();
```

Figure 4.10: Outline of TF-IDF

defining a custom *ValueSet* instead of the *Histogram*, thus performing data analysis as part of a defined user callback. This allows for increased reusability of code as developers do not have to explicitly implement communication strategies themselves.

Figure 4.10 shows an implementation of TF-IDF that initially computes the *tf* and node-local *idf* and then constructs a communication tree to perform a reduction. An implementation with ScalaJack is shown in Figure 4.11 where the reduction is defined as a *ValueSet* of the *StatTFIDF* object associated with the *idf* computation event. As part of the event's epilogue, the *idf* table is added to the *Stat* object. When inter-node compression is performed at the prologue of *MPI_Finalize*, the *idf* tables are compressed as well. With the ScalaJack version, users do not have to be concerned with implementing a communication tree and use ScalaJack's internal reduction tree to perform scalable compression. In our tests, we compare the naive implementation with the ScalaJack implementation with support for inter-node compression. As with CLAMR, tracing is not the goal here. Hence, we assess the scalability through the overhead of ScalaJack over the naive implementation.

Figure 4.12 shows the overhead of ScalaJack in comparison to the naive version. As can be seen, ScalaJack introduces minimal overhead of about 0.16% as reflected in Figure 4.13, thus

```
  MPI_Init();
  ..
  compute_tf();

  SJ_ce_pre(idf);
  compute_idf_local();
  SJ_ce_post(idf);

  MPI_Finalize();

StatTFIDF: idf
  after:
    jadd(idf_table);

ValIDF: StatTFIDF
  jmerge:
    reduce_idf();
  pack:
    serialize_idf_table();
  unpack:
    deserialize_idf_table();
```

Figure 4.11: Outline of TF-IDF with ScalaJack

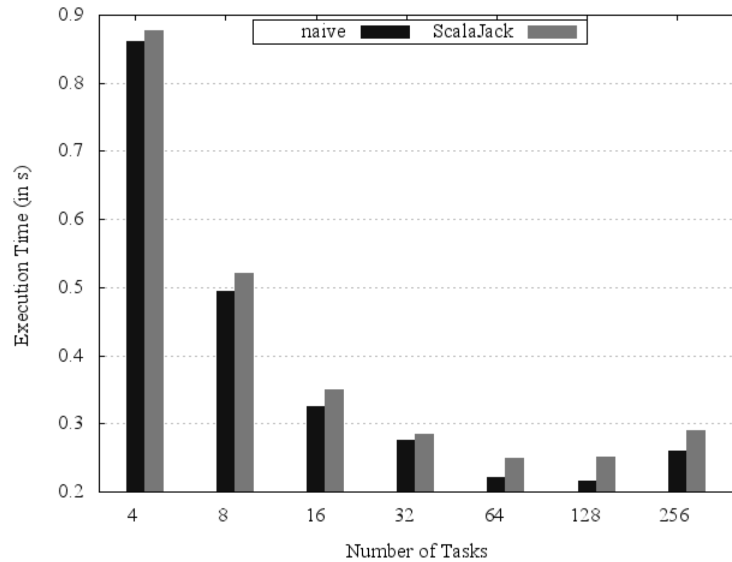

Figure 4.12: Execution times with TF-IDF

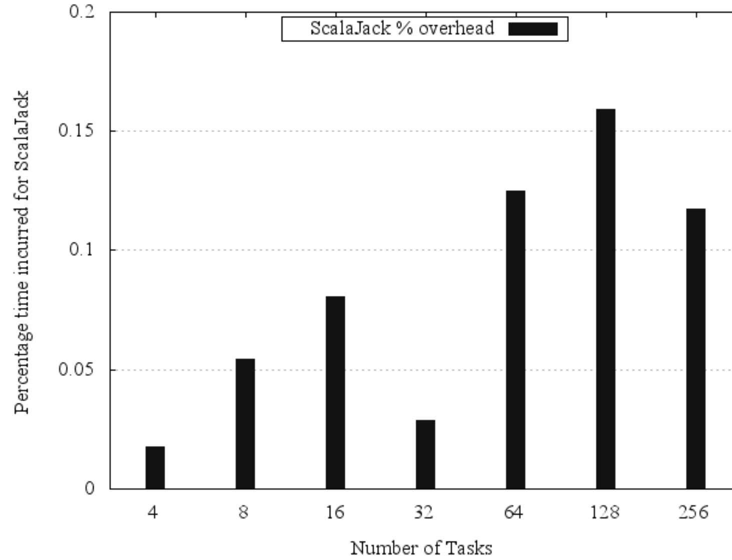

Figure 4.13: Percentage overhead with ScalaJack for TF-IDF

Table 4.3: Aspect metrics for TF-IDF

	naive		ScalaJack	
	aux	main	aux	main
CONC(comm, t)	1	0.3665	1	0.0683
DOS(comm)	0.4643		0.1273	
	main	aux	main	aux
DEDI(main, s)	0.4155	0.5945	0.1134	0.8666
DOF(main)	0.0286		0.5978	

proving to be light weight.

Table 4.3 shows the aspect-related metrics for the TF-IDF case study. With ScalaJack, concerns relating to the communication tree for final *idf* aggregation are eliminated and are made through an extension of the *ValueSet* class. This reduces the tangling of code, thus leading to better DOF and lower DOS scores.

Chapter 5

Related Work

Our implementation of customizable instrumentation with in-situ data analysis through ScalaJack is closely related to tools that support tracing or profiling of MPI programs as in [19][16].

Paraver [19] is a tracing and visualization tool that supports tracing of both shared memory and message passing programs. For MPI programs, Paraver includes a tracing library for intercepting MPI calls and saving them as individual trace file sizes during execution. These trace file sizes are then merged offline and then visualized. With Paraver, users cannot arbitrarily tag sections of the code to be instrumented thus preventing the users from identifying subtle performance anomalies. Also, ScalaJack performs compression of trace files on the fly and thus is efficient in comparison to Paraver.

VAMPIR [16] is another MPI tracing tool with support for visualization, that stores traces as flat files which are compressed later through zlib compression. Even though such tools generate scalable trace files, they do not take advantage of the underlying structure of the trace file. Thus, such trace files cannot be efficiently used for replay [23] or code generation [24]. Recent versions of VAMPIR provide support for marking regions in the trace with specific marker events for identifying potential hotspots in the trace files [4]. These markups can then be used by automated performance analysis tools like Scalasca [9] and periscope [18]. With ScalaJack, this can easily be achieved by writing instrumentation data with additional markups directly to the trace file. VAMPIR also supports tracing of arbitrary user events through automatic instrumentation via compiler abstractions or manual instrumentation of code. While VAMPIR relies on binary instrumentation through Dyninst to achieve automatic instrumentation, ScalaJack provides automatic instrumentation support through a built-in preloader in addition to supporting manual instrumentation. In contrast to VAMPIR, ScalaJack excels in providing a robust callback support using which users can offload cross-cutting concerns and reuse them efficiently in other programs. In addition, programs can leverage ScalaJack's compression tree framework to perform reduction of their own data structures efficiently without having to define

their own communication APIs.

Our work is also related to light-weight profiling tools like mpiP [22], gprof [10], HPCToolkit [1]. While these tools provide a simple and high-level information to provide a high-level understanding of performance problems, ScalaJack provides facilities to the user to do profiling of arbitrary interfaces in their programs in addition to supporting light-weight tracing. Since the instrumentation data is stored along with the trace files, users can correlate events to the data thus helping to diagnose subtle anomalies.

Chapter 6

Future Work

ScalaJack aims to reduce trace file sizes and improve scalability by providing customized instrumentation capabilities with support for in-situ data analysis. To facilitate automatic instrumentation of custom events, ScalaJack also has a built-in preloader that users can utilize to instrument their programs. ScalaJack may still be enhanced in a number of ways.

1. For MPI events, ScalaTrace internally keeps track of all the parameters. These parameters are stored as part of the trace and, hence, can be used to perform instrumentation based on predicates depending on parameters. Such support for parameters is not currently provisioned for custom events. With parameter tracking, users could instrument functions for certain invocations and, thus, can enhance scalability of trace file sizes.

2. The preloader currently has the ability to override only weak symbols. For non-weak symbols, users have to rely on manual instrumentation capabilities. This could, however, be improved by utilizing some form of binary instrumentation capabilities in the preloader so as to intercept any function. The downside of this approach will be increased overhead due to dynamic binary instrumentation.

3. The preloader does not currently provide users the ability to directly insert something as part of a prologue or an epilogue for a function, but relies on *Stat* objects registered with ScalaJack. With support for direct insertion, users could arbitrarily execute any function without relying on the user callback framework.

4. ScalaTrace internally performs intra-node compression as part of every event that has been recorded and inter-node compression as part of *MPI_Finalize*. However, for long running jobs, it is often desired to periodically perform out-of-band compression and trace-file writing so as to minimize overhead. This could be achieved through the user callback

framework where ScalaJack can internally provide callbacks for performing compression and IO to disk at specific events.

5. As ScalaJack is envisioned to be used in hybrid programs where users can instrument accelerator kernels as well, the prologues could be used to house debugging facilities for routines that execute in the kernel. This can be realized by generating code in the prologue of a kernel function to perform handshaking with a corresponding function in the CPU. As part of this handshaking process, the kernel could transfer debugging data, which can be stored as part of the trace. Conversely, the CPU can also feed specific data to the kernel routine for debugging purposes. Since the code is autogenerated by the preloader, the onus on the user is reduced to only tagging the debugging data that needs to be moved back and forth. Such a framework can be of significant benefit in hybrid codes for accelerators in environments with limited built-in debugging capabilities.

Chapter 7

Conclusion

With the race to exascale, systems are beginning to be comprised of millions of components. Diagnosis of application performance at such high scale is burdensome because tools suffer from scalability issues. Tools cannot afford to instrument interfaces exhaustively and then generate trace files that are not scalable. Such tool scalability issues can be effectively addressed by providing support for customizable instrumentation that the user can leverage and for in-situ reduction of diagnostic data, thus resulting in scalable trace file sizes. In addition, such a framework will help realise the Aspect-oriented paradigm of software engineering thus improving code readability.

We have implemented ScalaJack, a framework for customizable instrumentation with in-situ data analysis. ScalaJack provides APIs for users to tag sections of the code that need to be instrumented. This allows users to perform instrumentation at interfaces that are pertinent to the problem at hand, instead of having to instrument exhaustively, thereby often compromising scalability. ScalaJack employs novel intra-node and inter-node compression algorithms to preserve the execution structure of a program in a lossless fashion in addition to maintaining scalability. ScalaJack also provides support for automatic instrumentation of code through a preloader which generates code for interception of routines based on a specification file.

To facilitate in-situ analysis, ScalaJack provides the ability for users to perform reduction of data by registering callbacks with the framework. In addition to providing support natively to compress numeric data into histograms, ScalaJack provides APIs for users to define their own data elements depending on the application. Since the callbacks are synonymous to aspects, users can leverage them to write better code thus enhancing readability and maintainability.

An evaluation of ScalaJack with several case studies has shown that it is very light-weight, posing an overhead of under 0.2% and capable of producing lossless and near-constant trace sizes, while resulting in efficient, maintainable source codes with about 75% reduction in the degree of scattering *(DOS)*. Users can choose between the different levels of tracing provided by

ScalaJack to instrument interfaces that are pertinent to the problem at hand and thus generate trace files that orders of magnitude lower in size.

REFERENCES

[1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] Saman Amarasinghe, Dan Campbell, William Carlson, Andrew Chien, William Dally, Elmootazbellah Elnohazy, Mary Hall, Robert Harrison, William Harrod, Kerry Hill, et al. Exascale software study: Software challenges in extreme scale systems. *DARPA IPTO, Air Force Research Labs, Tech. Rep*, 2009.

[3] C Aspect. AspectC: AOP for C. 2004.

[4] Holger Brunst, Daniel Hackenberg, Guido Juckeland, and Heide Rohling. Comprehensive performance tracking with vampir 7. In *Tools for High Performance Computing 2009*, pages 17–29. Springer, 2010.

[5] Luiz DeRose, Ted Hoover Jr, and Jeffrey K Hollingsworth. The dynamic probe class library-an infrastructure for developing instrumentation for performance tools. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 7–pp. IEEE, 2001.

[6] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. Top500 supercomputer sites. *Supercomputer*, 13:89–111, 1997.

[7] Jack J Dongarra, Steve W Otto, Marc Snir, and David Walker. An introduction to the mpi standard. *Communications of the ACM*, 1995.

[8] Marc Eaddy, Alfred Aho, and Gail C Murphy. Identifying, assigning, and quantifying crosscutting concerns. In *Assessment of Contemporary Modularization Techniques, 2007. ICSE Workshops ACoM'07. First International Workshop on*, pages 2–2. IEEE, 2007.

[9] Markus Geimer, Felix Wolf, Brian JN Wylie, and Bernd Mohr. Scalable parallel trace-based performance analysis. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 303–312. Springer, 2006.

[10] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982.

[11] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. In *ACM SIGSOFT Software Engineering Notes*, volume 26, page 313. ACM, 2001.

[12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of AspectJ. In *ECOOP 2001Object-Oriented Programming*, pages 327–354. Springer, 2001.

[13] Los Alamos National Laboratory. Cell-based adaptive mesh refinement using MPI and OpenCL GPU code.

[14] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[15] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. In *Proc. Department of Defense HPCMP Users Group Conference*, 1999.

[16] Wolfgang E Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. *VAMPIR: Visualization and analysis of MPI resources*. Citeseer, 1996.

[17] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R de Supinski. ScalaTrace: Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel and Distributed Computing*, 69(8):696–710, 2009.

[18] Technical University of Munich. Periscope.

[19] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver: A tool to visualize and analyze parallel code. *WoTUG-18*, pages 17–31, 1995.

[20] Juan Ramos. Using tf-idf to determine word relevance in document queries.

[21] Open Source. Dyninst: An application program interface (api) for runtime code generation. *Online, http://www. dyninst. org.*

[22] J Vetter and C Chambreau. mpiP: Lightweight, scalable MPI profiling. *CASC/mpip*, 2005.

[23] Xing Wu. Scalable communication tracing for performance analysis of parallel applications. 2012.

[24] Xing Wu, Vivek Deshpande, and Frank Mueller. ScalaBenchGen: Auto-generation of communication benchmarks traces. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1250–1260. IEEE, 2012.

[25] Xing Wu and Frank Mueller. Scalaextrap: Trace-based communication extrapolation for spmd programs. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 113–122. ACM, 2011.

[26] Fan Zhang, Ciprian Docan, Manish Parashar, Scott Klasky, Norbert Podhorszki, and Hasan Abbasi. Enabling in-situ execution of coupled scientific workflow on multi-core platform. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1352–1363. IEEE, 2012.