ScalaJack: Customized Scalable Tracing with In-situ Data Analysis*

Srinath Krishna Ananthakrishnan and Frank Mueller

North Carolina State University, USA
mueller@cs.ncsu.edu

Abstract. Root cause diagnosis of large-scale HPC applications often fails because tools, specifically trace-based ones, can no longer record all metrics they measure. We address this problems by combining customized tracing and providing support for in-situ data analysis via ScalaJack, a framework with customizable instrumentation and pluggable extension capabilities for problem directed instrumentation and in-situ data analysis. We further eliminate cross cutting concerns by code refactoring for aspect orientation and evaluate these capabilities in case studies within and beyond the scope of tracing.

1 Introduction

Experience suggests that HPC codes suffer scalability issues each time the concurrency level increases by an order of magnitude. Analyzing the causes requires knowledge of an application's global and local behavior. Frequently, tracing is used for root cause analysis. Specific application events are identified and traced during execution. Tracing differs from profiling in that it tries to preserve more data, including the chronology of events, while profiling is inherently lossy and focuses on aggregate metrics of loops and nodes. But trace-based tools struggle to isolate problems since instrumentation costs can be prohibitive with exhaustive collection of metrics at events and results in perturbations that can mask the true problem. Traditional approaches employ periodic probing [6] instead of full instrumentation and may employ reduction in data volume through compression. However, this merely postpones the problem of analyzing the data, which requires decompression again. In-situ analysis is an alternative as it reduces data volume inherently and facilities realtime/online root cause analysis. Leveraging user knowledge for instrumentation, problem-specific tracing and analysis capabilities can thus be realized.

Contributions: We have developed ScalaJack to support *active analysis tracing*, i.e., problem-specific extraction and on-the-fly reduction of data through analysis. ScalaJack supports *user-customizable* instrumentation and user callbacks as pluggable extensions for instrumenting interfaces and a means for in-situ data analysis at specific execution points. This supports rapid generation of problem-specific analysis tools. Instrumentation via ScalaJack is *aspect-oriented*

* This work was supported in part by NSF grants 1217748, 0958311, and 0937908.

F. Silva et al. (Eds.): Euro-Par 2014, LNCS 8632, pp. 13–25, 2014.

to reduce cross-cutting concerns in source code to improve code readability, reuse, portability and maintainability, which aids in designing large and multi-scalar HPC codes. In experiments, ScalaJack shows scalable trace file sizes with increasing number of tasks and minimal overhead. Aspect-oriented analysis suggests significantly decreased scattering of cross-module code references.

2 Background

ScalaJack is a redesign of ScalaTrace to support customizable instrumentation, user callbacks and aspect-oriented program design. **ScalaTrace** [17] is a state-of-the-art scalable parallel communication tracing library for message-passing MPI programs. MPI events are traced through the PMPI profiling layer. ScalaTrace combines on-the-fly intra-node compression of MPI calls within loops with inter-node compression of events across nodes (in $MPI\_Finalize$). ScalaTrace employs RSDs and PRSDs ([Power] Regular Section Descriptors [14]) structures to represent events in a loop as constant size logs. An RSD is a tuple $< length, event_1 ... event_n >$ in one loop and a PRSD represents multiple RSDs in nested loops. E.g., two nested for loops with a barrier in the outer and a send in the inner loops correspond to PRSD $< 10, RSD_1, MPI\_Barrier >$ where RSD_1 is $< 10, MPI\_Send >$ for 10 iterations per loop level.

ScalaTrace represents events and parameters through an elastic data representation [24] that morphs scalars, vectors and histograms. Resulting trace files are scalable yet completely lossless, except for delta times between MPI events recorded as lossy histograms. A replay engine allows events to be replayed without original program code, even for non-deterministic histogram data.

Aspect-oriented programming [11] is a software engineering technique to reduce scattering of cross-cutting concerns in source code. An *aspect* is is a piece of code that cannot be factored out into procedural isolation due to cross-cutting *concerns* (e.g., logging, timing, performance monitoring, load balancing) located at *pointcuts* in the code, where *concerns* are the set of all aspects and a *pointcut* separates two regions of disjoint concerns.

Aspect-specific code is moved from the original application's *component* to an aspects specification, i.e., *advice*, which is executed at the original *pointcut* in the code (as a pre- or post-wrapper), often realized via run-time or compile-time support for aspects [12,3].

3 Design and Implementation

ScalaJack reuses compression techniques of ScalaTrace but augments and extends them by introducing an API to define custom events specific to a program and to register callbacks for in-situ analysis of live data. ScalaJack relies on aspects through run-time interpositioning of MPI calls via PMPI and dynamic pre-loading / tagging of event prologues / epilogues.

Figure 1 illustrates how ScalaJack composes generic instrumentation libraries with the application. Custom events are tagged by either augmenting the application with instrumented calls or by enumerating such events in a specification

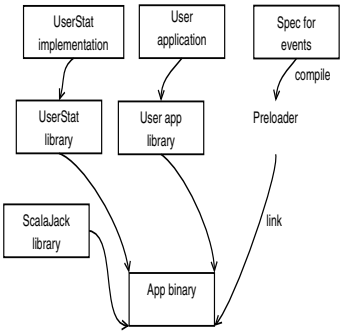


Fig. 1. Instrumentation Composition with ScalaJack

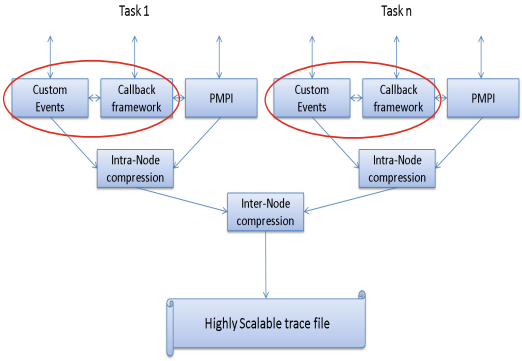


Fig. 2. ScalaJack's High-level Design

file. The user may provide an instrumentation class (UserStat, derived from the Stat class) that implements the methods to start/stop/merge specific trace events, which is compiled separately and linked into the application.

ScalaJack's high-level design is depicted in Figure 2 with novel components (circled) and redesigned existing components (non-circled). Each event is wrapped by ScalaJack with a prologue and epilogue to support tracing *and* invocation of aspect-specific callbacks. Event/user trace data within a task are compressed on-the-fly by exploiting the program's loop constructs while a second phase of compression is performed via inter-node compression over all tasks. This highly compressed single file trace is thus scalable with the number of processes.

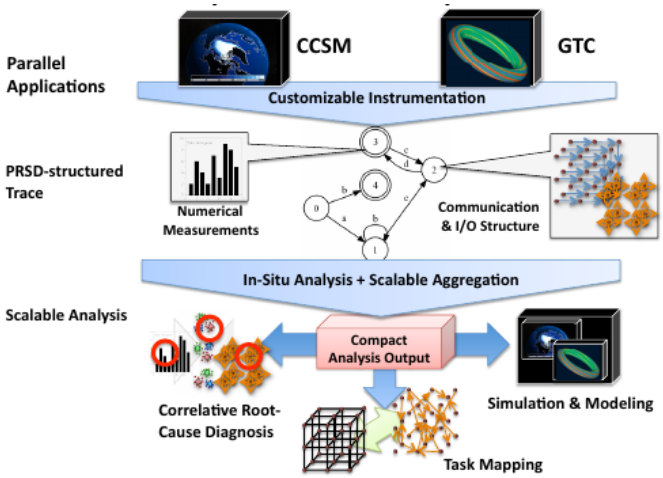


Fig. 3. Typical Application workflow with ScalaJack

A typical application workflow (Figure 3) consists of a parallel code with customized instrumentation to trace and instrument MPI routines or arbitrary functions augmented by in-situ reduction (through analysis) of instrumentation-derived data. Reduced data is co-located with the appropriate event blocks and stored as RSDs and PRSDs in a scalable fashion, preserving the structure of program/trace. Correlating data to the events provides insight into root causes to identify, e.g., performance anomalies. Other tertiary tasks due to cross-cutting concerns integrate readily, e.g., visualization, yielding better code modularity.

Custom Events can bee registered to extend ScalaTrace's scalable compression algorithms for interposing arbitrary events in programs. This level of tracing reduces default instrumentation to $MPI\_Init/Finalize$ events or, optionally, user-defined equivalents in the code, which would require user-provided alternatives for rank/size/barrier (of MPI) for internal ScalaJack functionalities, e.g., inter-node data reduction (not covered in this paper). A custom event API supports (a) event registration and (b) specification of pre-/post-wrappers.

Registration of custom events via the API returns an event code (orthogonal to MPI events) for further ScalaJack calls and internally establishes a control block for optional $flags$ for events. Flags may suppress stack signature generation (normally used to identify functions during compression). Signature omission may facilitate joint compression of event sets grouped together by a data-specific criterion or for aggregation.

Custom events invoke user-supplied arbitrary functions when triggered. Registered wrappers for pro- and epilogues resemble functions for custom events and are synonymous to those for MPI events instrumented via the PMPI interpositioning techniques. An auto-generated prelinker provides skeleton code that wraps the original function call. Custom wrappers may coexist with MPI wrappers per event, and both of their data resides in the same, single trace file. Flag-controlled tracing of just Init/Finalize facilitates inter-node compression for MPI-associated user events, while the mix of both event streams may hamper ScalaJack's default MPI compression.

Nested custom events, i.e., trace events inside pre-/post-wrappers, can cause incorrect ordering, e.g., before the epilogue of event 1, the prologue of event 2 is encountered. Instead of using stack-bloating data structures, pre-/post-wrappers are represented as two different events sequences.

User callbacks provide hooks at any communication point and selected call graph nodes, e.g., for in-situ data analysis on event or program data. They also support aspect orientation to separate cross-cutting concerns from main algorithms. Prologues of MPI events cause ScalaJack to create control blocks while epilogues consists of routines that append the events into the trace and engage in intra-node compression of trace data. User callbacks as pre-/post-wrappers serve as *pointcuts* and may augment the trace with user-collected data. User callbacks further support data analysis, optional on-the-fly compression, and, in contrast to MPI wrappers, even early reduction across nodes. A *Stat* (Statistics) class provides overloading capabilities by the user through object orientation with two instantiations for (a) the computation phase before the

event and (b) the communication phase of the event. Callbacks are established by overriding Stat's *start/end* methods or by a ScalaJack API call resulting in internal *Stat* instantiation and method overriding. Thus, callbacks before and after each compute/communicate phase are invoked out of the respective pre-/post-wrappers. An optional flag supports suppression of entries into the trace file to let users override Stat's *callback* method, which is invoked just once (without compute/communicate distinction).

User-directed compression: Data from in-situ analysis in callbacks enters the trace as a compressed histogram by default. Users can overload the $ValueSet$ class and support their own set of compression routines as callbacks invoked by our reduction framework with data marshaling. This supports *pointcuts* in programs while providing scalability even for customized user data types.

Most aspect-oriented frameworks map aspects to specific events. In contrast, ScalaJack aspects are universal across events but event-specific aspects are realized by light-weight filters. Users can access event objects of *pointcuts* to execute aspects for specific events/conditions, e.g., to access the send count of MPI events. Users can also access event trace queues in their structurally compressed form (PRSDs). This facilitates analysis on the entire trace, e.g., for trace similarity via k-means clustering to group traces based on a distance metric.

4 Evaluation

We assess the scalability of ScalaJack via traces generated by its custom event framework. In addition, the overhead incurred in using ScalaJack over a naïve implementation is studied. We evaluate ScalaJack by refactoring several case studies of typical HPC applications to utilize our 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.

All experiments were conducted on our ARC cluster with two AMD Opteron 6128 processors with 8 cores each (16 cores) per node and a QDR InfiniBand interconnect. Execution times and trace file sizes were averaged over 10 runs.

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 naïve implementation with respect to the code footprint, we utilize the degree of scattering (DOS) and degree of focus (DOF) metrics from [8,7]. Concentration ($CONC$) measures how many of the source lines related of 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. 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 for $|T| > 1$ [7]. 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

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

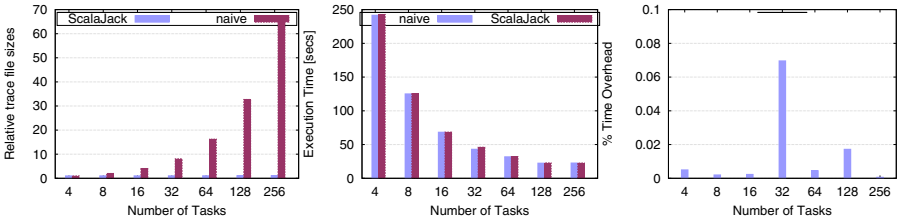
where $SLOC_{t,s}$ is the number of source lines of code ($SLOC$) in component t related to concern s, and $SLOC_t$ is the $SLOC$ in all of component t. Again, a better metric would be the normalized *degree of focus* (DOF)

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

where S is the set of concerns for $|S| > 1$. DOF is 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.

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 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 naïve 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 and both intra-node and inter-node compression are performed. The naïve version of IS initializes the PAPI library, followed by an instrumentation of the random sequence generation routine of IS with the PAPI API. 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 called to add data to the trace. The ScalaJack version differs from the naïve implementation by utilizing PMPI wrappers to trace events (and compress them) while the PAPI API calls are invoked as part of a $StatPAPI$ callback. 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.



(a) Trace file sizes (b) Execution times (c) ScalaJack % overhead

Fig. 4. IS Results

Figure 4(a) compares the trace files generated with ScalaJack and that of the naïve implementation. The trace file sizes shown are relative (normalized) to the ones generated with $n = 4$ tasks for the naïve and the ScalaJack versions, respectively. 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 naïve 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 naïve 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 for lower values of n due to timestamp data of few hundred bytes per event 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 [17,25].

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(b), ScalaJack introduces very little overhead to the naïve implementation's execution. To put it in a different perspective, Figure 4(c) shows the percentage overhead times of ScalaJack over the naïve implementation. As it can be seen, ScalaJack introduces the highest performance overhead for $n = 32$, i.e., for the best performance of IS under strong scaling, which is when instrumentation overhead (constant across n) contributes the most — but still amounts to just 0.07% overhead for $n = 32$. There is substantial variability in the overhead of ScalaJack over the naïve implementation since each task of the naïve implementation performs I/O to the parallel file system at $MPI\_Finalize$ to write n trace 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). This results in I/O contention. In contrast, only *rank* 0 performs I/O to the file system with ScalaJack after aggregating the traces from all its peers, i.e., a single file of rather moderate size (in the MBs) suffices.

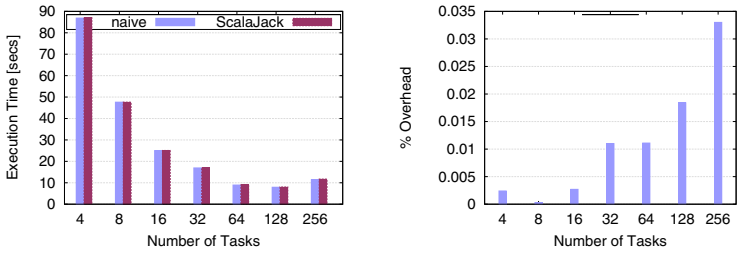
Table 1 (left columns) shows the improvement of using ScalaJack for separation of concerns over the naïve implementation. For IS, the identifiable components are main and PAPI, where the main component implements the benchmark

Table 1. Aspect metrics

	IS				CLAMR				TFIDF			
	naïve		ScalaJack		naïve		ScalaJack		naïve		ScalaJack	
	PAPI	main	PAPI	main	aux	main	aux	main	aux	main	aux	main
CONC(perf,t)	1	0.4777	1	0.0444	1	0.0739	1	0.0118	1	0.3665	1	0.0683
DOS(perf)	0.4992		0.0850		0.1369		0.0234		0.4643		0.1273	
	perf	sort	perf	sort	main	fd	main	fd	main	aux	main	aux
DEDI(main,s)	0.0588	0.9411	0.0057	0.9942	0.2708	0.7293	0.0540	0.9459	0.4155	0.5945	0.1134	0.8666
DOF(main)	0.7782		0.9770		0.2102		0.7955		0.0286		0.5978	

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 (better) DOS score and a correspondingly higher (better) DOF score for ScalaJack compared to the naïve implementation.

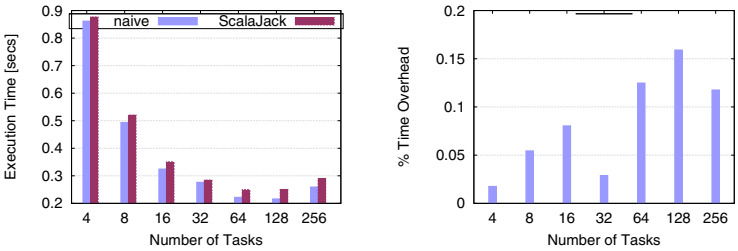
Visualization and Load balancing: We next evaluate the effectiveness of ScalaJack for aspect-oriented application scenarios beyond tracing for performance analysis. We first consider CLAMR [13], an adaptive mesh refinement solver developed at Los Alamos National Laboratory. CLAMR implements a cell-based shallow water code by computing the finite difference on AMR using MPI. CLAMR periodically refines the mesh and also performs load balancing across the nodes to redistribute the meshes. In addition, CLAMR performs OpenGL or MPE-based visualization to display the mesh's current state.



(a) Execution times (b) ScalaJack % Overhead

Fig. 5. CLAMR Results

Application codes like CLAMR have numerous conflicting concerns that can be effectively addressed using ScalaJack. In the naïve version of CLAMR, 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. In CLAMR's ScalaJack version, the various concerns that are tangential to the main concern at hand are refactored into the appropriate prologue/epilogue.CLAMR was evaluated with the custom level of tracing, i.e., only custom events are traced



(a) Execution times-IDF (b) ScalaJack % Overhead

Fig. 6. TF-IDF Results

and no MPI events except for $MPI\_Init/Finalize$. Custom events are configured to be created without the stack signature so as to reduce the trace footprint. 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 naïve and ScalaJack. Instead, to assess the scalability, we compare the execution times of both versions.

Figure 5(a) compares the overhead of ScalaJack through the differences in execution time between the naïve and the ScalaJack versions of CLAMR. Figure 5(b) shows that 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 1 (middle columns) summarizes the improvements of using ScalaJack to eliminate concerns from 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 registered custom events. 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 (higher) DOF score and thus a better (lower) DOS score.

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 terms. 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 naïve 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 is supported via the definition of a custom $ValueSet$ instead of the $Histogram$, thus performing data analysis as part of a defined user callback. Such a solution allows for increased reusability of code as developers do not have to explicitly implement communication strategies themselves.

The naïve TF-IDF initially computes the tf and node-local idf and then constructs a communication tree to perform a reduction. The ScalaJack version defines the reduction 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 naïve 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 naïve implementation.

Figure 6(a) shows the overhead of ScalaJack in comparison to the naïve version. ScalaJack introduces minimal overhead of about 0.16% as reflected in Figure 6(b), thus proving to be light weight. Table 1 (right columns) 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 (higher) DOF and better (lower) DOS scores.

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. 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 files during execution. These trace files are merged offline and then visualized. Paraver and other tracing tools [22,16,9,18] allow users to store user-defined values in a trace but they lack ScalaJack's compression of trace files on-the-fly and the ability to directly affect compression of native trace values (as opposed to user-defined trace values).

VAMPIR [16] is another tracing tool for MPI/OpenMP/CUDA events with support for visualization that stores traces as flat files, which are compressed

later through zlib compression. Even though such tools generate trace files with limited scalability, they do not take advantage of the underlying structure of the trace file. Thus, such trace files cannot be efficiently used for replay [24] or code generation [25] supported by ScalaTrace. 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]. With ScalaJack, this can easily be achieved by writing instrumentation data with additional markups directly to the trace file and using plugins for domain-specific compression.

Several tools [22,16,9,18,6,5] support tracing of arbitrary user events through automatic instrumentation via compiler abstractions, dynamic preloading or manual instrumentation of code, both statically and dynamically via binary rewriting. ScalaJack also supports built-in preloading and manual instrumentation but emphasizes separation of cross-cutting concerns via aspect orientation, which simplifies reuse for other programs. In addition, programs not only leverage ScalaJack's compression tree framework to perform reduction of their own data structures efficiently but also improve on intra-node compression and inter-node reduction of default communication tracing data, which is unprecedented.

Arnold et al. [2] identified task behavior equivalence classes using stack signature similarity. They utilized MRNet, a software overlay network that provides efficient multicast and reduction communications [21]. MRNet provides a general framework with generic plugins, each requiring an explicit implementation of compression and reduction. In contrast, ScalaTrace natively supports compression and reduction, i.e., trace-specific plugins directly complement this process or even manipulate internal data structures affecting the trace file format.

Our work is also related to light-weight profiling tools like mpiP [23], gprof [10], and HPCToolkit [1]. While these tools provide simple and high-level information to support a high-level understanding of performance problems, ScalaJack provides facilities to the user for 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 them to diagnose subtle anomalies dependent on event orders.

6 Conclusion

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 provides direct access to intra-node and inter-node compression algorithms and data structures to preserve the execution structure of a program in a lossless fashion in addition to maintaining scalability.

ScalaJack facilitates in-situ analysis provides by allowing users to perform reduction of data by registering callbacks. In addition to providing native support

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 for event parameters, while resulting in efficient, maintainable source codes with about 75% reduction in the *degree of scattering*. Overall, this demonstrates the fidelity of ScalaJack in facilitating trace generation and analysis for users.

References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.: HPCToolkit: Tools for performance analysis of optimized parallel programs. Concurrency & Comp. Practice and Experience 22(6), 685–701 (2010)
2. Arnold, D.C., Ahn, D.H., de Supinski, B.R., Lee, G.L., Miller, B.P., Schulz, M.: Stack trace analysis for large scale debugging. In: International Parallel and Distributed Processing Symposium (2007)
3. Aspect, C.: AspectC: AOP for C. (2004)
4. Brunst, H., Hackenberg, D., Juckeland, G., Rohling, H.: Comprehensive performance tracking with Vampir 7. In: Tools for HPC 2009, pp. 17–29 (2010)
5. Buck, B., Hollingsworth, J.: An API for runtime code patching. International Journal of High Performance Computing Applications 14(4), 317–329 (2000)
6. De Rose, L., Hollingsworth, J., Hoover, T.: The dynamic probe class library – an infrastructure for developing instrumentation for performance tools. In: International Parallel and Distributed Processing Symposium (April 2001)
7. Eaddy, M., Zimmermann, T., Sherwood, K., Garg, V., Murphy, G., Nagappan, N., Aho, A.: Do crosscutting concerns cause defects? IEEE Transactions on Software Engineering 34(4), 497–515 (2008)
8. Eaddy, M., Aho, A., Murphy, G.C.: Identifying, assigning, and quantifying crosscutting concerns. In: Workshop on Assessment of Contemporary Modularization Techniques, pp. 2–2 (2007)
9. Geimer, M., Wolf, F., Wylie, B.J.N., Abraham, E., Becker, D., Mohr, B.: The scalasca performance toolset architecture. In: International Workshop on Scalable Tools for High-End Computing (June 2008)
10. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: A call graph execution profiler. ACM Sigplan Notices 17(6), 120–126 (1982)
11. Kiczales, G., Hilsdale, E.: Aspect-oriented programming. In: ACM SIGSOFT Software Engineering Notes, vol. 26, p. 313 (2001)
12. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–354. Springer, Heidelberg (2001)
13. Laboratory, L.A.N.: Cell-based adaptive mesh refinement using MPI and OpenCL GPU code, `https://github.com/losalamos/CLAMR`
14. Marathe, J., Mueller, F., Mohan, T., de Supinski, B.R., McKee, S.A., Yoo, A.: METRIC: Tracking down inefficiencies in the memory hierarchy via binary rewriting. In: Int'l Symp. on Code Generation and Optimization, pp. 289–300 (March 2003)

15. Mucci, P.J., Browne, S., Deane, C., Ho, G.: PAPI: A portable interface to hardware performance counters. In: HPCMP Users Group Conference (1999)
16. Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: Visualization and analysis of MPI resources. Supercomputer 12(1), 69–80 (1996)
17. Noeth, M., Ratn, P., Mueller, F., Schulz, M., de Supinski, B.R.: ScalaTrace: Scalable compression and replay of communication traces for high-performance computing. Journal of Parallel Distributed Computing 69(8), 696–710 (2009)
18. of Dresden, T.U.: Score-p: Application instrumentation, `https://silc.zih.tu-dresden.de/scorep-current/html`
19. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVER: A tool to visualise and analyze parallel code. In: WoTUG-18: Transputer and occam Developments.Transputer and Occam Engineering, vol. 44, pp. 17–31 (April 1995)
20. Rajaraman, A., Ullman, J.: Mining of Massive Datasets. Cambridge Press (2011)
21. Roth, P., Arnold, D., Miller, B.: MRNet: A software-based multicast/reduction network for scalable tools. Supercomputing, 21–36 (2003)
22. Shende, S.S., Malony, A.D.: The tau parallel performance system. Int. J. High Perform. Comput. Appl. 20(2), 287–311 (2006)
23. Vetter, J., Chambreau, C.: mpiP: Lightweight, scalable MPI profiling. CASC/mpip (2005), `http://mpip.sourceforge.net/`
24. Wu, X., Mueller, F.: Elastic and scalable tracing and accurate replay of non-deterministic events. In: Int'l Conference on Supercomputing, pp. 59–68 (June 2013)
25. Wu, X., Deshpande, V., Mueller, F.: ScalaBenchGen: Auto-generation of communication benchmarks traces. In: International Parallel and Distributed Processing Symposium, pp. 1250–1260 (2012)