Elastic and Scalable Tracing and Accurate Replay of Non-Deterministic Events [*]

Xing Wu, Frank Mueller
North Carolina State University
mueller@cs.ncsu.edu

ABSTRACT

SCALATRACE represents the state-of-the-art of parallel application tracing for high performance computing (HPC). This paper presents SCALATRACE II, a next generation tracer that delivers even higher trace compression capability, even when events are not always regular. In this work, we contribute a spectrum of novel compression and replay techniques that are fundamentally different from our past approaches. SCALATRACE II features a redesigned low-level encoding scheme of trace data such that data elements are *elastic* and self-explanatory. With this new encoding scheme, trace compression is enhanced by introducing innovative intra-node and inter-node trace compression algorithms that guarantee high compression rates in a loop structure agnostic fashion. In practice, the improved compression scheme is particularly efficient for scientific codes that demonstrate inconsistent behavior across time steps and nodes. A novel approach is further contributed to probabilistically replay sequences of non-deterministic events. To assess the compression efficacy of SCALATRACE II, we conduct experiments not only with computational kernels but also a real-world application, the Parallel Ocean Program (POP). Compared to the first generation SCALATRACE, we observe key improvements on trace compression for benchmarks with inconsistent time step behavior and diverging task level behavior while retaining timing accuracy even under probabilistic replay.

1. INTRODUCTION

The compute power of supercomputers has been doubling each year in the past two decades. The era of exascale computing is projected to arrive in the near future. With such large systems, recording the program behavior of parallel applications for post-mortem performance analysis is becoming increasingly difficult. On the one hand, analyzing complicated scientific applications requires complete and accurate performance data. On the other hand, the large number of processors/cores and the increasing gap between computational power and I/O performance pose great challenges in terms of efficiency and scalability of performance analysis tools. Consequently, traditional analysis tools either collect lossless traces by sacrificing scalability [23] or report only aggregated statistical information that might be insufficient for in-depth performance analysis and debugging [32]. To address this discrepancy, we designed SCALATRACE, a scalable parallel communication and I/O tracing library that features on-the-fly trace compression [26, 29]. For single program, multiple data (SPMD) parallel applications, SCALATRACE is able to collect lossless traces that are much more space-efficient than the past approaches.

SCALATRACE is one tool that represents the state-of-the-art of parallel application tracing for high performance computing (HPC), specifically for communication events. In this paper, we present SCALATRACE II, the next generation SCALATRACE that features a fundamental redesign in every aspect. SCALATRACE II is designed to address the shortcomings of previous work. It targets inefficiencies in the compression of communication traces for applications with inconsistent program behavior across time steps and diverging parallel control flow. For example, coupled large-scale scientific codes such as the Community Earth System Model (CESM) [5] exhibit multiple program, multiple data (MPMD) behavior. They perform multi-physics simulation with different modules using diverse inputs, executing a multitude of algorithms, and running on different sets of processors. To generate scalable traces for these applications, methodologies that better exploit trace similarities across time steps and MPI tasks are in demand.

With SCALATRACE II, we contribute a spectrum of novel compression and replay techniques that are fundamentally different from our past approaches. In SCALATRACE II, MPI parameters and loop information are stored as elastic data element representations, a redesigned low-level encoding scheme that is automatically evolving and self-explanatory. By annotating loop information with participant information, we designed a loop agnostic inter-node compression scheme that ensures perfect event matching for applications with task-specific communication patterns. We also redesigned the task-level loop compression to perform approximate loop iteration matching, which is particularly effective for applications with inconsistent behavior across time steps.

SCALATRACE II inherits the lossy philosophy proposed in our previous work [37]. In essence, SCALATRACE II can be configured to compress MPI parameters to the utmost using probabilistic methods while still preserving most of the advantages of the lossless approach. We further developed SCALAREPLAY II, a brand-new probabilistic replay engine that is compatible with the loop agnostic trace format. With SCALAREPLAY II, we improved the coordinated random value selection of the trace replay algorithm. Optimizations such as multi-context traversal are also designed to boost the robustness, accuracy, and scalability of the replay engine.

We evaluated SCALATRACE II with regard to two aspects: (1) effectiveness of trace compression and (2) correctness and timing accuracy of the probabilistic replay. We conducted experiments with a subset of the NAS Parallel Benchmark suite, the Sweep3D neutron-transport kernel, and the Parallel Ocean Program (POP). Experimental results demonstrate that SCALATRACE II achieves key improvements on trace compression for benchmarks exhibiting task-specific communication patterns, inconsistent loop behavior, and/or diverging parallel control flow. Results on probabilistic replay show that SCALAREPLAY II is able to accurately re-produce

[*]This work was supported in part by NSF grants CCF-1217748, CNS-0958311, and HECURA-0937908.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'13, June 10–14, 2013, Eugene, Oregon, USA.
Copyright 2013 ACM 978-1-4503-2130-3/13/06 ...$15.00.

the execution times of the original applications. Across all the test cases, the mean absolute percentage error of the replay times is only 5.7%. Given such accuracy, we conclude that the lossy compression scheme, powerful in reducing trace sizes, is equally applicable to scenarios where timing accuracy is required.

This work makes the following contributions:

- We propose SCALATRACE II, a fundamental redesign of SCALATRACE that features a spectrum of novel compression techniques to improve trace compression for applications with inconsistent loop-level and task-level behavior.
- We designed SCALAREPLAY II, a probabilistic replay engine compatible with SCALATRACE II that significantly improves robustness, accuracy, and scalability.
- By comparing the compression ratio of two generations of SCALATRACE with computational kernels and real-world applications, we investigate the causes of compression inefficiency and demonstrate potential solutions to obtain better compression.

2. PREVIOUS WORK

This section summarizes our previous work on the last generation SCALATRACE and provides the background for the improvements and redesigns discussed in Section 3.

2.1 Intra-node and Inter-node Compression

SCALATRACE is a communication tracing framework for parallel applications using the Message Passing Interface (MPI), the *de facto* standard for scientific computing. It utilizes the MPI profiling layer (PMPI) to intercept MPI calls. It collects lossless, order-preserving, and space-efficient communication traces by exploiting the program structure and performing a two-stage trace compression, i.e., intra-node and inter-node compression.

SCALATRACE utilizes Extended Regular Section Descriptors (RSDs) to capture loop structures of one or multiple communication events. Power-RSDs (PRSDs) are utilized to recursively specify RSDs in nested loops. To perform intra-node loop compression, SCALATRACE constantly searches for repeating event sequences ending with the current tail event. In order for two event sequences to be considered matching, SCALATRACE requires not only equal lengths but also identical call stack signatures, loop structures, and equivalent MPI parameter values for events at corresponding positions in the two sequences. When a new loop iteration (a repeating sequence of events) is identified, SCALATRACE eagerly compresses it with the previous iteration by inserting a new outer loop or increasing the trip count of an existing RSD/PRSD by one. As such, SCALATRACE is able to generate scalable traces with respect to the number of time steps if loop trip counts are consistent.

At program completion, SCALATRACE performs an inter-node compression along a radix tree rooted in rank 0. During this reduction, internal nodes combine their traces with other task-level traces received from child nodes. To perform compression, two task-level event sequences are compared at the granularity of a loop structure, i.e., matching standalone events and whole loops are merged. Subsequently, diverging subsequences bounded by matching preceding and succeeding events are concatenated. For applications with regular SPMD behavior, SCALATRACE generates near constant sized traces irrespective of the number of nodes of a parallel job.

2.2 The Ranklist Representation

Inter-node compression identifies events that are executed by multiple MPI tasks. To perform the compression for one such event, SCALATRACE attaches a *ranklist,* i.e., a list of the ranks of the participating MPI tasks, to the event. Ranklists are constructed in a recursive manner. Using the EBNF meta-syntax, a ranklist is represented as

$$< dimension\ start\ iters\ stride\ \{iters\ stride\} >,$$

where *dimension* is the dimension of the group, *start* is the rank of the starting node, and the $< iters\ stride >$ pair is the iteration and stride of each dimension. With the ranklist representation, SCALATRACE is able to capture the spatial characteristics of a set of nodes and to describe it succinctly.

2.3 Histogram-based Compression

SCALATRACE preserves the timing information of an execution by recording the delta times, i.e., relative times between successive communication and computational stages. In the case where a single communication/computational stage in the source code generates multiple delta times due to loop and/or parallel execution, these delta times are merged using histograms during loop compression and/or cross-node reduction. The histogram-based compression is a lossy approach that captures the distribution of a set of values instead of memorizing the exact values. While the histogram approach is particularly effective for scalable compression of performance data in the form of numerical values, it is also applicable to MPI parameters, such as DEST and SOURCE, to more closely capture the communication pattern.

3. TRACE COMPRESSION AND REPLAY

3.1 Elastic Data Element Representation

SCALATRACE II features a complete redesign of SCALATRACE, ranging from the very low-level data structures to the core trace compression algorithms. In this section, we introduce the novel elastic data element representation.

A SCALATRACE trace file is a human-readable text file. A data element in the trace file is an integer that represents the value of either an MPI event parameter, such as the destination of a send, or a program control flow parameter, such as the trip count of a loop. From our experience with the previous version of SCALATRACE, we learned that even though a data element is apparently simple as an integer, it may become complicated when the trace is compressed to a high degree. For example, assume a scalar integer value d represents the destination of an MPI_Send operation. When the same MPI_Send is called twice with destinations d_1 and d_2 in consecutive loop iterations, the two events will be compressed due to SCALATRACE's loop compression mechanism (see Section 3.2). Thus, the scalar value d evolves into a vector (d_1, d_2). When the same loop has multiple iterations, the destination vector may grow in a non-scalable fashion. Hence, a vector compression mechanism is activated. Finally, the inter-node compression further imposes another level of complexity on the data element representation. In a nutshell, a compressed data element ought to contain not only the parameter value, but also loop details and participant information.

In SCALATRACE II, we introduce the elastic data element representation and apply this representation to all the data elements in the trace. The elastic data element representation is a list of $< value\ vector, ranklist >$ pairs, where a *value vector* is simply a C++ vector of primitive integer values. On initialization, there is only one such pair in the list consisting of a value vector of a single parameter value and a ranklist of a single participant. During loop compression, new values are appended to the value vector in the order that they are generated so that the replay engine or other trace analyzers can traverse the values in the correct order. During inter-node reduction, we merge the ranklists when the value vectors fully match, otherwise a new $< value\ vector, ranklist >$ pair is added to the list. As such, the data element is fully self-explanatory;

no additional information is required to resolve a data element for a particular rank and loop iteration.

To keep the size of the value vector scalable, we constantly perform a loop compression against the entries in the vector. Whenever possible, a vector of $m \cdot n$ elements is represented as a vector of m elements and n iterations. Note that the choice of the vector compression algorithm is not set in stone. For example, run-length encoding might be more efficient for loop parameters when an MPI event is specific only to certain loop iterations (see Section 3.2.1). Since the vector compression mechanism is encapsulated in the elastic data element representation, it is possible to intelligently choose the best compression strategies and convert to the most space-efficient format when necessary.

As the fundamental data structure of ScalaTrace ii, it is vital to guarantee its scalability even under extreme circumstances. We realized that there always exist cases where even a sophisticated algorithm fails to compress the value vectors. We therefore further enhanced the elastic data element representation to exploit probability-based compression using histograms (see Section 2.3) for selected parameter types, such as SOURCE, DEST, and COUNT of point-to-point communication routines. Nonetheless, utilizing such lossy compression techniques poses challenges for the trace replay. We have updated our past probabilistic trace replay technique to address these challenges. The new approach is discussed in detail in Section 3.5.

3.2 Compressing Partially Matching Loops

ScalaTrace utilizes the MPI profiling layer (PMPI) to intercept MPI calls during application execution. It performs loop detection and compression by searching for consecutive repeating patterns in the MPI event sequence. In contrast to the instruction-level binary instrumentation, which is able to pinpoint the entry and exit points of loop structures, the ScalaTrace approach relies heavily on recognizing repeating patterns. With ScalaTrace ii, we redesigned the task-level loop compression algorithm from the ground up to support the compression of loops with iteration-specific behavior.

3.2.1 Handling Iteration-specific Behavior

Production-grade scientific applications such as the Parallel Ocean Program (POP) demonstrate inconsistent behavior across time steps. POP performs a set of computations and communications of an inner loop in multiple iterations in each time step. Due to inconsistent data-dependent convergence in the computation, the trip counts of the inner loop vary across different time steps. In addition, branches inside loop structures also lead to loop iterations with different event counts and unmatching event sequences. This behavior can also be observed in many Adaptive Mesh Refinement (AMR) applications in which the input set is dynamically rebalanced on a periodic basis. Due to the iteration-specific behavior, ScalaTrace's task-level loop compression fails to compress loop iterations because the loop detection algorithm requires identical event sequences for loop iterations with matching inner loop structures.

In ScalaTrace ii, we overcome the shortcoming by loosening the iteration matching criteria. In order for two consecutive sequences of events $E1,\ldots,Ea$ and $Eb,\ldots,En$ to be considered a match, ScalaTrace ii only requires their beginning and ending events to match, i.e., $E1 == Eb$ and $Ea == En$. Under such criteria, once two matching loop iterations are identified, ScalaTrace ii's aggressive longest common subsequence (LCS) based loop compression algorithm will merge the rest of the events, irrespective of the loop lengths or the inner loop structures. As an example, if a certain node executes the following code,

```
for(int i=0; i<2; i++){
    MPI_Barrier(); // E1
    if(i == 0)
        MPI_Isend(); // E2
    if(i == 1)
        MPI_Irecv(); // E3
    MPI_Barrier(); // E4
}
```

Figure 1: Loop with Iteration-specific Behavior

the trace after loop compression will be

$$E1_{(4,2)}\ E2\ E3\ E4,$$

where the subscript of $E1$ indicates that $E1$ is the beginning event of a loop structure of with 4 members events and 2 iterations. In general, we use the following mnemonic to describe the loop stack associated with a loop head event E:

$E_{(m_1,i_1)(m_2,i_2)\ldots(m_n,i_n)}$: *E is the head event of a series of n-nested loops, where the outermost loop has a loop length (member event count) of* m_1 *and an iteration count of* i_1*, the second outermost loop has a loop length of* m_2 *and an iteration count of* i_2*, and so on.*

A unique challenge of forcibly merging partially matching loop iterations is to preserve the information of in which iteration a certain event was actually called. As in the example above, a mechanism is needed to tell that $E2$ was only called in the first iteration whereas $E3$ was only called in the second iteration. To address this problem, we represent the loop information as elastic data elements. Recall that the value vector grows as more values are appended to the vector due to loop compression. For example, assuming event E is the head event of loop L, $E_{(a_1\ a_2,b_1\ b_2)}$ indicates that when the first time loop L is executed, it has a_1 member events and b_1 iterations, whereas for the second time, it has a_2 member events and b_2 iterations. (This is possible when loop L is executed during two different iterations of its parent loop). We hence treat every event as a loop of length 1 and iteration 1. During loop compression, we merge events according to the results of the longest common subsequence analysis and manipulate the loop information according to the following rules:

1. An event $E_{(1,0)}$ is called a *dummy event* because the loop information indicates that it is executed zero times.

2. For event $E_{(a_1\ a_2\ldots,b_1\ b_2\ldots)(c_1\ c_2\ldots,d_1\ d_2\ldots)\ldots}$, adding an outer loop $E_{(a_1\ a_2\ldots,1\ 1\ldots)(a_1\ a_2\ldots,b_1\ b_2\ldots)(c_1\ c_2\ldots,d_1\ d_2\ldots)\ldots}$ does not change the loop structure in terms of the iteration times and event order w.r.t. execution in nested loops. The added outer loop is thus called a *pseudo-loop* with just one iteration.

3. If event $E1_{(a_1\ a_2\ldots,b_1\ b_2\ldots)(c_1\ c_2\ldots,d_1\ d_2\ldots)\ldots}$ in iteration $I1$ matches with $E2_{(i_1\ i_2\ldots,j_1\ j_2\ldots)(k_1\ k_2\ldots,l_1\ l_2\ldots)\ldots}$ in iteration $I2$, merge $E1$ and $E2$ by merging the loop information at corresponding levels. Merging the value vectors is accomplished by appending the value vector of $I2$ to that of $I1$. If the loop stack depth d_1 at $E1$ does not match the depth d_2 at $E2$, e.g., $d_1 > d_2$, align the loop stacks by adding pseudo-loops to the top of the loop stack at $E2$ as placeholders to avoid mismatches between the extra outer loops at $E1$ and $E2$ during traversal.

4. If event E is in iteration $I1$ but not in iteration $I2$, create a dummy event E' of E and insert it into $I2$ immediately before the matching event M of $I1$ and $I2$ returned by the longest common subsequence analysis. The loop stack of E' is created according to that of E by adding pseudo-loops. The vector values at each nest level of E' are generated by referring to M for the number of times it would be encountered if it were in $I2$. As such, E' acts as a placeholder to avoid mistakenly calling E when executing $I2$. After the insertion of all the iteration-specific events, merge iterations $I1$ and $I2$ according to the third rule.

5. When merging iteration $I2$ into iteration $I1$, if the outermost loop of the head event E of $I1$ is not a loop of the entire event sequence of $I1$, a new outer loop is identified and a new loop descriptor $(n,2)$ is added to the top of the loop stack of E, where n equals to number of events in both $I1$ and $I2$.

In essence, the rules above ensure that the iteration-specific events will only be executed in the correct iterations. The introduction of pseudo-loops as placeholders guarantees that 1) iteration-specific events are evaluated but not executed in loop iterations they do not belong to, and, therefore, 2) meaningful loop information is evaluated and fetched in the correct loop iteration. Nevertheless, the core of the loop compression algorithm is still the longest common subsequence analysis performed against the two sequences of MPI events. The matching event pairs returned by the LCS analysis are the basis for loop information adjustment. Therefore, the complexity of this algorithm is $O(m \cdot n)$, where m and n are the numbers of events in the two event sequences, respectively.

By applying the aforementioned loop compression guidelines, SCALATRACE II is able to compress loops with iteration-specific behavior. For example, the MPI code shown in Figure 1 is eventually compressed as follows:

$$E1_{(4,2)}\ E2_{(1,1\ 0)}\ E3_{(1,0\ 1)}\ E4$$

3.2.2 Handling Trailing Iterations

The NAS Parallel Benchmarks (NPB) BT code exemplifies a pattern of an MPI event sequence that a multitude of stencil codes share at the end of a time step. Each MPI task communicates with its neighbors with a series of send, receive, and wait operations in a loop, as illustrated in the simplified example of Figure 2.

```
/* m time steps */
for(int i=0; i<m; i++){
    ... // MPI events
    for(int j=0; j<n; j++){
        MPI_Isend();
        MPI_Irecv();
        MPI_Waitall();
    }
}
```

Figure 2: Loop with Trailing Iterations

The original SCALATRACE compresses the outer loop only if n, the trip count of the inner loop, is a constant. In contrast, SCALATRACE II's more aggressive loop compression discussed in Section 3.2.1 can always compress the outer loop even if n is not a constant, e.g., $n = f(i)$. However, since SCALATRACE II eagerly compresses detected loops, the second iteration of the outer loop will be compressed immediately when the first iteration of the inner loop terminates. As a result, the remaining $n-1$ iterations of the inner loop cause the *trailing iterations* problem.

To address the trailing iterations problem, we redesigned the entire loop detection and compression algorithm to perform a delayed merge, as shown by Algorithm 1. Essentially, Algorithm 1 does not eagerly merge a newly identified loop iteration. It rather marks it as a *pending iteration* so that a potential trailing iteration can be merged with the pending iteration later without having to perform any decompression. Specifically, after an event E is appended to the trace, DETECTLOOP() is called to find the *target_head*, *merge_tail*, and *merge_head* for two matching loop iterations ending with $target\_tail == E$. Assuming a loop structure is found, the sequence *merge_head*, ..., *merge_tail* can be 1) the ending subsequence of a pending iteration detected previously, 2) a pending iteration whose match has already been found, or 3) a sequence independent of any prior loop structures. In the first case, the sequence *target_head*, ..., *target_tail* is identified as a trailing iteration. An event sequence can simultaneously be the trailing iteration of multiple pending iterations in a nested manner, i.e., line 16 - 21 of Algorithm 1 updates all the pending iterations by appending the trailing iteration to each of them. In the second case, where the sequence *merge_head*, ..., *merge_tail* is itself a pending iteration, the sequence *target_head*, ..., *target_tail* is then identified as the next iteration of the pending iteration. Since it is now safe to conclude that there will be no more trailing events for the pending iteration *merge_head*, ..., *merge_tail*, Algorithm 1 performs a delayed iteration merge by calling MERGEPENDINGITERATION() (line 22 - 24). Finally, the newly detected iteration *target_head*, ..., *target_tail* is always marked as a pending iteration in either case. The function MERGEPENDINGITERATION() merges the iteration between the events *pending_head* and *pending_tail* with the iteration between the memorized events *head* and *tail*. Before calling LCSLOOPCOMPRESSION() which implements the algorithm introduced in Section 3.2.1, MERGEPENDINGITERATION() first compresses pending iterations of all nested inner loops (line 32 - 51) by recursively calling itself. Finally, when MPI_Finalize is called, all loops either have zero or one pending iteration. To eventually merge them, the function FINALIZEPENDINGITERATIONS() is called, which treats the entire trace as an iteration and recursively merges inner loops by calling MERGEPENDINGITERATION().

3.3 Approximate Stack Signature Matching

SCALATRACE preserves a call stack signature by logging the call sites of the calling stack for each event. Using these stack signatures, SCALATRACE is able to distinguish MPI calls of the same type by their locations in the program. The stack signature therefore serves as the only basis for comparison of events, which then makes loop detection possible. Nonetheless, strictly enforced stack signature comparison may not always benefit trace compression. For example, POP wraps MPI_Bcast of different data types with different functions, which are then invoked in 36 different files at approximately 400 different locations. Experimental results show that due to such usage, the size of the trace of the initialization stage — a stage that is usually less important for performance analysis — accounts for 26% of the size of the final trace. More commonly, a number of scientific applications, including POP and the NPB BT and SP codes, are coded according to the data decomposition, their communication topology, or their simulation stages, with the MPI events hidden deep in the call stack, as shown in Figure 3. As a result, these applications also create trace events with various stack signatures even though they are functionally symmetric. In both cases, there exists the need to trade the call stack information for better compression.

```
simulate(){     solve_x(){      solve_y(){
  while(i<s){     // compute      // compute
    solve_x();    MPI_Isend();    MPI_Isend();
    solve_y();    MPI_Irecv();    MPI_Irecv();
    i++;          MPI_Wait();     MPI_Wait();
  }               MPI_Wait();     MPI_Wait();
}               }               }
```

Figure 3: The Simplified NPB BT Code

In SCALATRACE II, we loosened the stack signature comparison criteria to tolerate a pre-defined number of different frames. When comparing two stack signatures, we start from the first call site (the *main* function) and compare the call sites in corresponding frames one by one. The comparison returns *true* only if the number of different frames is less than the user-defined threshold. As an ongoing improvement, we also allow users to specify a range of instruction addresses, so that the call sites within it will always be considered a match. During loop compression, we compare event $E1$ in iteration $I1$ with event $E2$ in iteration $I2$ in term of both event

Algorithm 1 Loop Compression with Delayed Merge

Precondition: T: the trace after a new event was appended as the new tail

```
 1: function DetectLoop(T)
 2:     target_tail ← T.tail
 3:     merge_tail ← T.FindMatch(target_tail)
 4:     target_head ← merge_tail.next
 5:     while true do
 6:         merge_head ← T.FindMatch(TARGET_HEAD)
 7:         if merge_head.isPendingMember == true then      ▷ potential trailing iteration, more checks
 8:             pending_tail ← FindPendingTail(merge_head)
 9:             if pending_tail == merge_tail then      ▷ trailing iteration of a pending iteration must follow the pending iteration immediately
10:                 break
11:             end if
12:         else                                         ▷ not a trailing iteration, no more check
13:             break
14:         end if
15:     end while

16:     if target_head...target_tail is a trailing iteration then
17:         PIs ← FindPendingsForTrailing(target_head, target_tail)
18:         for pendingIteration ← PIs.first, PIs.last do
19:             pendingIteration.AddTrailing(target_head, target_tail)
20:         end for
21:     end if

22:     if merge_head...merge_tail is a pending iteration then      ▷ perform the delayed merge: merge a pending iteration only when the next iteration (namely, target_head...target_tail) is found
23:         MergePendingIteration(merge_head, merge_tail)
24:     end if

25:     for event ← target_head, target_tail do
26:         event.isPendingMember ← true
27:     end for
28: end function

29: function MergePendingIteration(pending_head, pending_tail)
30:     head ← FindKnownMergeHead(pending_head)
31:     tail ← FindKnownMergeTail(pending_tail)

32:     for event ← head, tail do
33:         if event.IsKnownMergeHead() == true then
34:             h ← FindPendingHead(event)
35:             t ← FindPendingTail(event)
36:             new_tail ← MergePendingIteration(h, t)
37:             if tail == t then
38:                 tail ← new_tail
39:             end if
40:         end if
41:     end for

42:     for event ← pending_head, pending_tail do
43:         if event.IsKnownMergeHead() == true then
44:             h ← FindPendingHead(event)
45:             t ← FindPendingTail(event)
46:             new_tail ← MergePendingIteration(h, t)
47:             if pending_tail == t then
48:                 pending_tail ← new_tail
49:             end if
50:         end if
51:     end for

52:     LCSLoopCompression(head, tail, pending_head, pending_tail)
53:     DeleteEvents(pending_head, pending_tail)
54:     return tail
55: end function

56: function FinalizePendingIterations(T)
57:     ...
58: end function
```

type and stack signature. If their signatures differ less than the predefined limit, we replace $E2$'s signature with that of $E1$ and update all the signature-annotated statistics of $E2$'s succeeding events accordingly. With such user-configurable stack signature imprecision, ScalaTrace II is able to better exploit the potential of trace compression than the original ScalaTrace for applications like POP or BT, as illustrated in Figure 3. The user-defined limit serves as a tuning parameter to trade off compression against accuracy.

3.4 Loop Agnostic Inter-node Compression

ScalaTrace performs inter-node trace compression to exploit the single-program multiple-data (SPMD) paradigm of scientific applications. However, the compression capability of the original ScalaTrace is limited by the fact that loop structures must be treated as an indivisible unit. For example, the code in Figure 4 cannot be compressed across nodes because the *for* loops on different nodes have mismatching trip counts and event sequences. The strong dependence on the perfect matching of loop structures is partially alleviated by the recursive loop matching algorithm proposed in our previous work [37], but it still cannot handle the case where loop trip counts do not match.

```
Rank 0:                     Rank 1:
1: for(i=0;i<5;i++){        1: for(i=0;i<6;i++){
2:     MPI_Isend(1);        2:     MPI_Isend(0);
3:     MPI_Irecv(1);        3:     MPI_Irecv(0);
4: }                        4:     MPI_Waitall(2);
5: MPI_Isend(1);            5: }
6: MPI_Irecv(1);
7: MPI_Waitall(12);
```

Figure 4: Code Needs Loop Agnostic Inter-node Compression

The restrictions on loop structures for inter-node compression are eliminated in ScalaTrace II due to the introduction of the elastic data element representation. Close examination shows that the crux of the compression problem stated above is the coupling of the loop information and the event participants information. Specifically, the loop structure is formed during task-level loop compression and only applies to the same task. Given two loop head events from different MPI tasks, they cannot be merged if the loop structures diverge, because there is no mechanism to recover the task-specific loop information once it is compressed. By representing the loop information as elastic data elements, a separate ranklist is attached for each loop data element (including trip count and loop length). When merging two loop head events, we simply merge the loop structures at each corresponding loop level by following the general rules of compression of the elastic data element representation. Namely, if the loop structures match, they are compressed by merging their ranklists. If the loop structures at a certain loop level do not match, they are merged by adding another $< value\ vector, ranklist >$ pair to distinguish their task-specific loop information.

By decoupling loop information from event participant information, ScalaTrace II is able to perform loop structure agnostic inter-node compression. During inter-node compression, the longest common subsequence is determined by evaluating only the stack signatures of the events originating from different MPI tasks, and the events are then merged accordingly. For example, the events of the code shown in Figure 4 will be merged into the trace shown in Figure 5, where the number in brackets shows the ranklist of the loop information prior to merging, and the MPI parameters are ignored.

Rank	Event
0 1	$MPI\_Isend()_{(2,5)[0](3,6)[1]}$
0 1	$MPI\_Irecv()$
0	$MPI\_Isend()$
0	$MPI\_Irecv()$
0 1	$MPI\_Waitall()$

Figure 5: Final Trace of the Code in Figure 4

Lastly, since the new inter-node compression algorithm is loop agnostic, it does not have to perform LCS analysis recursively for each level of nested loops. Hence, assuming m and n are the lengths of two task-level traces, the complexity of the inter-node compression algorithm is $O(m \cdot n)$.

3.5 Replaying Non-deterministic Traces

For scalability reasons, SCALATRACE II's elastic data element representation may internally transform from a lossless $< value\ vector, ranklist >$ pair representation to a lossy histogram representation for pre-configured parameters including SOURCE, DEST, and COUNT. As was discussed in our prior work, representing non-performance data, such as DEST, as histograms still preserves meaningful information regarding the communication topology [37]. However, it poses a great challenge to the re-creation of the program behavior from the probabilistic trace because critical communication parameters are not accurate anymore.

SCALAREPLAY II is the new replay engine designed to cope with probabilistic traces. In contrast to the previous version of SCALAREPLAY, we redesigned the trace traversal algorithm to support SCALATRACE II's loop agnostic traces. We have also made key improvements in SCALAREPLAY II to boost the robustness and replay accuracy. SCALAREPLAY II utilizes a coordinated random value selection approach described in our previous work. In essence, during replay, nodes parse send events but skip receive events in the trace. At send events, senders select receivers from the DEST histograms by referring to random numbers. In order to generate matching receives for the send operations, each node parses the traces of other nodes to locate send operations addressed to itself. This is made possible by the fact that senders and their potential receivers agree on the random number used for value selection (by generating an identically seeded sequence of random numbers). In this way, the overhead of exchanging control messages via back-channel communication is avoided.

Improvements for SCALAREPLAY II center around a novel trace traversal strategy. In the past approach, each node used a single pointer to traverse a global trace; whenever there is a loop structure, the node traverses it as a participant. However, this is impossible with SCALATRACE II's new trace format because loop structures interleave in the final trace, as shown in Figure 5. In addition, traversing with a single pointer also causes timing accuracy problems. With the previous approach, a node parsed every event in the order it was seen during the traversal. However, due to a stack signature mismatch, events that happened simultaneously may be recorded far apart in the trace, as shown in Figure 6. With the single pointer approach, task 0 will not issue MPI_Irecv(1) until it reads event 4 after having performed some computation for 10 seconds at event 2. As a result, task 1 will be blocked on the blocking send (event 4) for 10 seconds and the total runtime of the program approaches 20 seconds, i.e., almost twice as much as it ought to be.

```
if(rank==0){
    MPI_Irecv(1);
    compute(10s);
    MPI_Wait();
}else if(rank==1){
    MPI_Send(0);
    compute(10s);
}
```

ID	Rank	Event
1	0	*MPI_Irecv*(1)
2	0	*compute*(10*s*)
3	0	*MPI_Wait*()
4	1	*MPI_Send*(0)
5	1	*compute*(10*s*)

Figure 6: Trace Needs Multiple Context Pointers for Replay

To address these issues, SCALAREPLAY II utilizes multiple traversal context pointers during replay. Intuitively, a trace reflects the result of a parallel execution, where the parallel processes progress concurrently through potentially distinct control flows with occasional synchronizations. The replay engine mimics this process by replaying with one primary traversal pointer while keeping track of the other nodes' parallel executions with multiple additional traversal context pointers. In SCALAREPLAY II, a traversal context is a lightweight data structure that keeps track of the progress of the traversal on behalf of a certain MPI task. It consists of an event pointer, a loop information manager, a random number manager, and a timer. The event pointer always points to the next event to be replayed/evaluated. It supports operations such as *hasNext*() and *next*(). The loop information manager keeps track of the traversal by memorizing the current loop stack as well as iteration counts at each loop level. The random number manager guarantees that contexts of the same rank on different nodes always agree on the same series of random numbers. Lastly, the timer is used to calculate the aggregated execution time at a certain event according to the recorded times of events traversed so far.

During replay, each node progresses according to its primary context, i.e., the context with the rank of a node. It issues all the events other than receives, and sleeps for all recorded computational phases within its context as a normal replay. To post matching receives for potential senders, each node also traverses the trace of the other nodes by maintaining secondary traversal contexts for them. When traversing secondary contexts, all non-send events and computations are ignored so that the current node can quickly identify send operations addressed to itself. However, the current node does not post a receive immediately when a send is identified. Instead, it postpones the receive to approximately the time that a corresponding send operation is issued on the sender side, which can be estimated by referring to the timer of the sender's secondary context. By posting receives in this way, SCALAREPLAY II manages to clear the system receiving buffer in a timely manner and thus improves replay time accuracy.

To further improve the performance and scalability of SCALAREPLAY II, additional optimizations are implemented. In practice, most parallel applications are designed such that each node only has a limited number of point-to-point communication destinations (otherwise, collectives are used). We therefore have introduced a negotiation stage before the replay in which each node calculates a destination set of a configurable size and informs selected receivers so that each node only has to maintain a limited number of traversal contexts during replay. In addition, we also improved the performance of the replay engine by overlapping context management with simulated compute times that the primary context has to perform anyhow. With these optimizations, the replay engine manages to scale to a large number of nodes.

4. EVALUATION

We evaluated SCALATRACE II with regard to two aspects: (1) its effectiveness of trace compression, as well as the advantages of different compression optimizations, and (2) correctness and timing accuracy of probabilistic replay. For experiment (1), we used a subset of the NAS Parallel Benchmark suite (version 3.3 for MPI) [2], including BT, CG, LU, MG, and SP, the Sweep3D neutron-transport kernel [33], and the Parallel Ocean Program [28]. We chose these benchmarks because they exercise both collectives and point-to-point communications in multiple time steps. Besides, some of these benchmarks either do not have consistent loop behavior or do not show strict SPMD regularity. Consequently, these benchmarks poses great challenges to the existing trace compression libraries, including the last generation SCALATRACE. In these experiments, we configured SCALATRACE II to use different compression algorithms and we study the impact of each option. In experiment (2), we assessed SCALATRACE II's and SCALAREPLAY II's capabilities of preserving and re-producing computational performance with respect to wall clock execution times. Particularly, we conducted all replay experiments with probabilistic traces, which is significantly more challenging than replaying with lossless traces. The second experiment used the same benchmarks as in experiment (1).

All experiments were conducted on ARC, a 1,728 core cluster with 108 compute nodes, 32 GB memory per node, and Infiniband QDR interconnect. Due to limited access to the system, our experiments run on a subset of nodes sufficient to reflect trends of trace size growth with respect to increasing execution scale.

4.1 Trace File Size

The first experiment evaluates ScalaTrace II's compression effectiveness with the NPB BT, CG, LU, MG, SP codes, Sweep3D, and POP. We chose these benchmarks because they are stencil codes exhibiting multi-dimensional communication topologies and complicated loop structures. Among these benchmarks, MG, SP, and POP demonstrate inconstant message sizes and irregular SPMD behavior, and are hence particularly challenging for lossless and structure-preserving trace compression. In these experiments, we compared ScalaTrace II with our past approach. In order to demonstrate the effect of different configurations, we itemize the optimizations (as explained below) and collected traces by applying the options in an incremental manner:

• *ScalaTrace II*: features only the loop agnostic inter-node compression enabled by the elastic data element representation;

• *LCS Loop Compression*: additionally performs the longest common subsequence based loop compression;

• *Approximate Signature Matching*: adds a finer-grained optimization that matches and merges events when stack signatures differ by no more than a pre-defined threshold;

• *Parameter Histogram*: adds lossy compression that converts overlong value vectors into histograms.

According to the improvements achieved with ScalaTrace II, benchmarks are divided into three categories and results are depicted in Figure 7.

The first category consists of BT and CG. These benchmarks either demonstrate perfectly matching loop iterations or regular SPMD behavior that leads to structurally identical task-level traces. Hence, our past approach is able to capture the loop structures and has little difficulty merging time step loops across tasks. Consequently, ScalaTrace II only shows limited reduction in trace size for BT and CG when configured to be fully lossless. Nevertheless, there is still room for improvement when fuzziness is allowed. For example, by applying the approximate stack signature matching, ScalaTrace II manages to deliver another 22% trace size reduction for BT on average. More importantly, when parameter histograms are enabled for elastic data elements, we eventually obtained constant sized traces for BT and CG — a critical improvement that makes a key difference at large scale.

LU and Sweep3D constitute the second category for which ScalaTrace II improves trace compression by forcibly merging task-level traces in a loop agnostic way. Both LU and Sweep3D are stencil codes with a 2D task layout. Depending on the location in the 2D communication topology, a node may have a different number of neighbors, and thus follow a communication pattern that is unique to one of the nine communication groups (4 corners, 4 boundaries, and interior nodes). Because the past approach matches the loop structures as an entirety during inter-node reduction, it fails to exploit the similarities in time step loops across communication groups. ScalaTrace II takes advantages of these similarities. Consequently, by applying loop agnostic inter-node compression alone, ScalaTrace II manages to reduce trace sizes by 28% and 41% for Sweep3D and LU, respectively. By enabling extra optimizations, ScalaTrace II eventually generates traces that are 43% and 65% smaller for LU and Sweep3D, respectively.

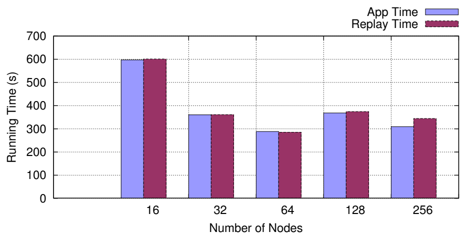

Figure 9: Probabilistic Replay Time Accuracy of POP

The most compelling improvement, however, is observed for the NPB MG and SP codes, and the Parallel Ocean Program. Among these benchmarks, SP sends messages with fluctuating sizes in loops. This prevents loop compression of the last generation ScalaTrace but can be handled with ScalaTrace II where all parameters are represented as elastic data elements. MG is the most challenging test case of the NAS Parallel Benchmark suite. It features a complicated communication pattern consisting of a primary 7-point 3D torus and a secondary nested 3D torus among nodes at particular positions in the topological space. Due to the interleaving of these two patterns, MG demonstrates both iteration-specific behavior and task-specific behavior, and thus poses great challenges for both intra-node and inter-node compression. By utilizing approximate loop iteration matching and loop agnostic inter-node compression, ScalaTrace II produces lossless traces that are orders of magnitude smaller for MG and SP, as depicted in Figure 7(f) and 7(g) on a logarithmic *y* axis. To further compress the varying message sizes for SP, we enabled lossy tracing with parameter histograms and thus obtained near constant sized traces that are collectively only 0.5% of the trace size of the past approach.

POP performs ocean simulations for multiple time steps. It features both inconsistent loop behavior across time steps and diverging task-level behavior that hinders inter-node compression. In previous work, we applied probabilistic compression techniques at the MPI parameter level and managed to greatly reduce POP's trace size [37]. With ScalaTrace II, we improve our previous work by more systematically exploiting structural properties of the trace. As shown in Figure 7(h), by utilizing loop agnostic inter-node compression, we reduced the trace size by a maximum of two orders of magnitude. This is almost as much of an improvement as we obtained with the previous lossy approach, yet we still maintain lossless traces. Furthermore, after additional optimizations are enabled, we eventually obtained near constant sized traces that are 48 to 351 times smaller. Besides, we also conducted experiments to assess the time step scalability, namely how efficient ScalaTrace II's longest common subsequence based loop compression can detect and compress time step loops. For this experiment, we keep the execution scale at 64 MPI tasks and increase the number of time steps. Experimental results in Figure 7(i) show that, by utilizing the LCS based approximate loop matching, ScalaTrace II is able to produce a constant sized trace. In contrast, the traditional approach is sensitive to iteration-specific events and thus is not time step scalable for POP. From these experimental results, we conclude that even with parameter-level probabilistic compression techniques, it is still possible to analyze structural properties in the trace systematically and perform compression in a top-down manner.

4.2 Probabilistic Replay Time Accuracy

The second set of experiments assesses ScalaTrace II's and ScalaReplay II's capabilities of preserving and re-producing computational times. We focus on 1) checking if ScalaReplay II is able to correctly coordinate the random value selection across nodes to replay probabilistic traces without deadlock, and 2) determining how accurate probabilistic replay can re-produce execution times of an original application. We conducted these exper-

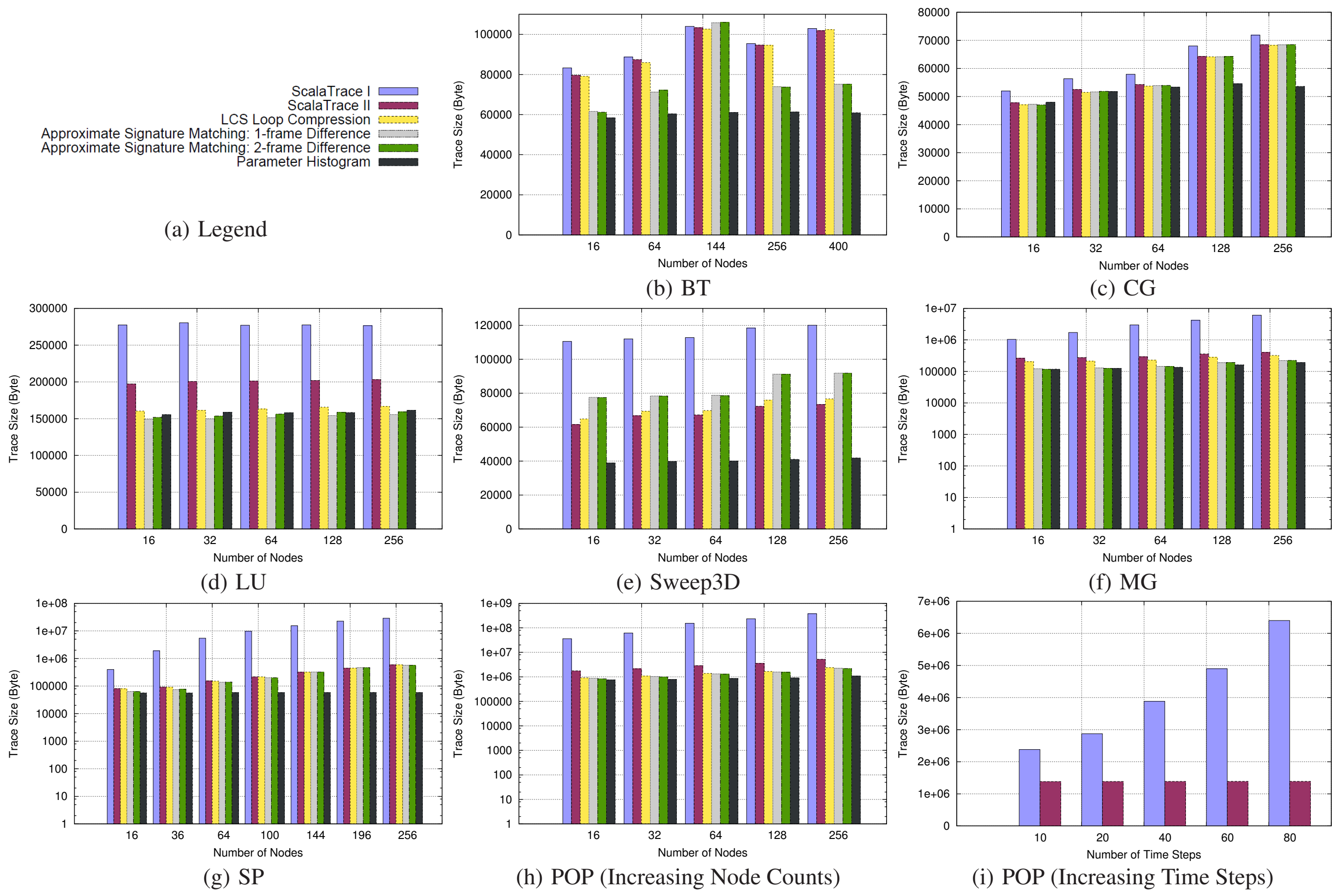

Figure 7: Trace File Sizes [Bytes] for Varying Number of Nodes of NPB BT, CG, LU, MG, SP, Sweep3D, and POP

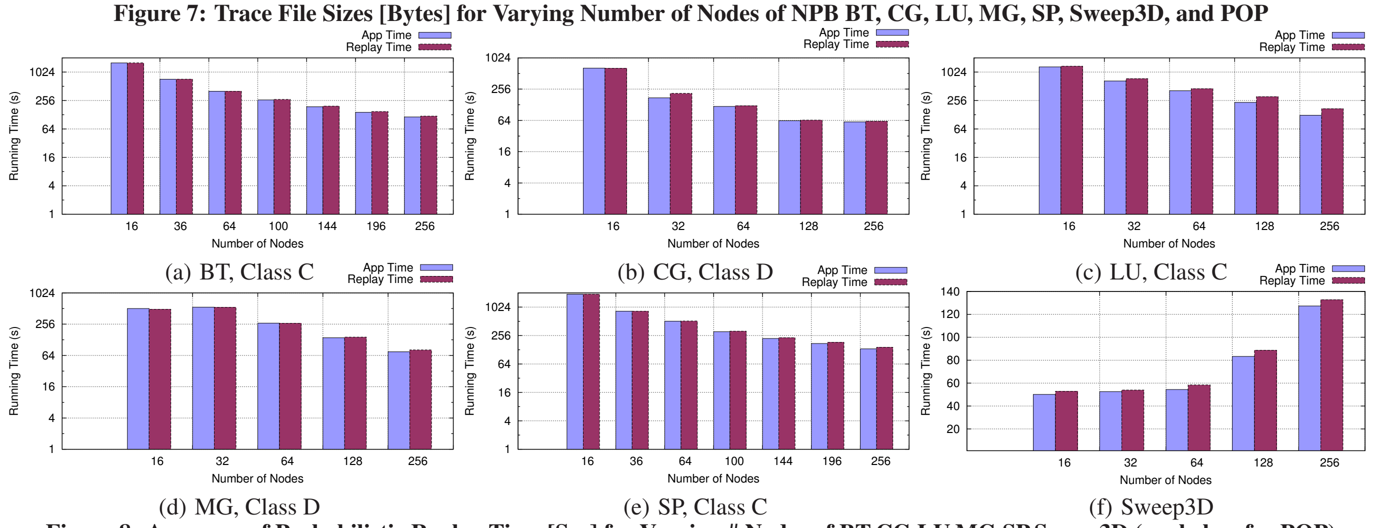

Figure 8: Accuracy of Probabilistic Replay Time [Sec] for Varying # Nodes of BT,CG,LU,MG,SP,Sweep3D (see below for POP)

iments with the same set of benchmarks used for the first experiment. Among these benchmarks, the NAS Parallel Benchmarks and POP are strong-scaling codes while Sweep3D is a weak-scaling code. Problem sizes were chosen for strong-scaling benchmarks to ensure that a reasonable amount of computation exists across all tested scales. To provide input for the replay engine, we configured ScalaTrace II to collect probabilistic traces where critical MPI parameters (e.g., SOURCE, DEST, COUNT) are represented as histograms. Particularly, we set the histogram-triggering threshold to 1 to forcibly convert all elastic data elements into histograms irrespective of how concise lossless value vectors actually are.

Figures 8,9 compare the probabilistic replay times with the execution times of the original applications. First, being able to obtain these results allows us to validate that the coordinated random value selection of the replay algorithm is deadlock free for the evaluated benchmarks. Besides, the experimental results also show that ScalaReplay II can accurately re-produce the computational performance of the original applications. Quantitatively, the mean absolute percentage error of the replay times (i.e., $100\% \times |(T_{\text{replay}} - T_{\text{app}})/T_{\text{app}}|$) across all test cases in Figures 8,9 is only 5.7%. Such high replay timing accuracy indicates that 1) the execution times are accurately preserved within lossy traces and 2) the probabilistic replay approach is able to re-produce the runtimes of the original applications without introducing unmanageable control overhead. Overall, given the accurately preserved and re-produced performance characteristics, we conclude that the histogram-based lossy compression, which has been shown to produce powerful reductions in the trace size, is equally viable in scenarios where timing accuracy is required.

5. RELATED WORK

Our work is closely related to prior research in the area of parallel application tracing and profiling [23, 27, 13, 31]. Traditional tracing tools such as Vampir [23], Extrae [8], and Paraver/Dimemas [27] collect plain application traces that are not scalable due to the sheer size of the performance data gathered. The Open Trace Format (OTF) aims at scalable tracing [17]. But OTF utilizes regular zlib compression, and tools based on it generally lack structure-aware compression. Hence, these tools cannot fully exploit structural similarities, nor are they suitable for trace-based scalability analysis and code generation [35, 36, 34].

SEQUITUR exploits hierarchical structures in sequences of discrete symbols for compression [24, 25]. It constructs a context-free grammar for a given sequence by representing repeating diagrams as non-terminals. Because SEQUITUR excels in both data compression and structural inference, it is employed by an array of algorithms and tools as the compression infrastructure for a variety of purposes. For example, Marathe et al. utilize SEQUITUR to compress data access instructions in their memory tracing work [22]. Larus proposes whole program paths (WPP) to capture a program's dynamic control flow, where an enhanced SEQUITUR algorithm is designed to compress acyclic path traces [19]. Krishnamoorthy et al. present a trace compression algorithm that is largely based on SEQUITUR [18]. To fully exploit the pattern detection capability of SEQUITUR, this work performs trace compression at argument level instead of event level. While this optimization is effective in improving the compression, it also makes the final trace unreadable and not structure-preserving. In general, if the program structure is not preserved at the event level, most post-processing or trace-based performance analysis becomes difficult or even infeasible because decompressing and effectively rendering the trace may require large amounts of memory and computing power that are not available on commodity desktops or laptops.

Recent advances in online trace compression utilize domain-specific techniques to achieve trace size reduction. SCALATRACE performs task-level loop compression and cross-node trace compression in a memory-efficient manner [26]. It generates near constant sized or orders of magnitude smaller traces for SPMD codes. Xu et al. construct coordinated performance skeletons from traces to estimate application execution time in new hardware environments [38, 39]. They adapt a pattern analysis algorithm from bioinformatics to perform loop analysis. Nonetheless, due to the lack of on-the-fly loop compression, this tool is subject to limitations on time step scalability. Knupfer et al. utilize Complete Call Graph (CCG) to hierarchically store an application trace according to the call stack [16]. By comparing and merging similar subgraphs, the trace is compressed in a bottom-up fashion. In contrast to our work, CCGs cannot handle inconsistent program behavior that leads to mismatching low-level sub-structures in CCGs.

Besides lossless or near lossless tracing techniques, our work is also related to the work that provides lightweight application profiling functionalities. For example, mpiP collects aggregated statistical information about MPI functions and computation times [32]. Gprof measures the durations and frequencies of procedures using a hybrid approach of instrumentation and sampling [14]. HPCToolkit collects call path profiles [9, 1]. To further reduce the overhead involved in profiling, Gamblin et al. utilize statistical sampling and parallel clustering techniques to reduce the number of parallel processes from which performance data is collected, and thus improve the scalability of parallel profiling tools [12, 11, 10]. In contrast to the lossless tracing approach, tools like mpiP generally report simple and high-level information that is only suitable for a superficial understanding of performance problems. For in-depth performance debugging or complicated analysis, application tracing is still necessary. As an instrumentation framework for both communication event tracing and performance data collection, SCALATRACE II can employ the statistical methods proposed in prior research to improve its numerical performance data collection and compression. This is subject to future work.

A unique approach for quick acquisition of communication traces involves program slicing. Program slicing is a source code analysis technique that effectively reduces a program to a subset of the statements (a program slice) that is relevant to a target statement or variable. As an example, Zhai et al. proposed the FACT approach [42]. This approach constructs a program slice for MPI calls and strips out the computation. With the communication-only program slice, it then becomes feasible to obtain the communication trace readily without executing the computational part. While it is possible to combine program slicing with existing communication trace compression techniques, an inherent shortcoming of slicing is that it neither captures execution times, nor can it handle data-dependent control flows. Consequently, this technique is only applicable in limited scenarios.

In addition to techniques on communication tracing and profiling, our work is also relevant to prior research on parallel replay. For example, Rolt^{MP} proposes a Lamport timestamp-based approach for deterministic replay of programs with non-deterministic receives [30]. MPIWIZ is a deterministic replay method that can only replay a subset of the tasks of an MPI application [40]. PHANTOM employs deterministic replay and cross-node performance clustering techniques to predict the performance of parallel applications on future systems [41].

In a broader sense, prior research on memory tracing and memory trace-based performance analysis supplements this work. As the next generation of SCALATRACE, SCALATRACE II continues to utilize Regular Section Descriptors (RSDs) and Power-RSDs (PRSDs) to describe nested loop structures in a trace. RSDs were originally proposed to track inter-procedural side effects on common substructures of arrays to promote compiler-aided parallelization [15]. Marathe et al. adapted the RSD representation and proposed PRSDs for memory trace compression [21, 20]. Budanur et al. further designed Extended-PRSDs to perform multi-level scalable parallel memory tracing in SCALAMEMTRACE [3]. SIGMA employs online trace compression to collect lossless memory traces for simulation and performance tuning [6]. Elnozahy et al. utilize loop detection and reduction for address trace compression [7]. VPC3 employs value predictors to compress events comprising program counter values and extended data fields [4].

6. CONCLUSION

Application tracing is one of the most important and useful vehicles for performance analysis and debugging of parallel applications. Yet, designing scalable and efficient parallel tracing tools for exascale systems and grand-challenge HPC applications remains an open problem. In this work, we contribute SCALATRACE II, a fundamental redesign of SCALATRACE that features a spectrum of innovative lossless compression techniques aiming at scalable trace compression of large-scale scientific codes with irregular SPMD behavior or even MPMD characteristics. We designed an elastic data element representation to address compression inefficiencies of previous work. Enabled by the new encoding scheme, novel algorithms are devised to perform approximate loop matching and loop agnostic cross-node trace compression. We also incorporated parameter histogram-based lossy compression capabilities into SCALATRACE II and adapted the replay subsystem to enable more accurate probabilistic trace replays. We evaluated SCALATRACE II with the NAS Parallel Benchmarks, Sweep3D, and a real-world application, the Parallel Ocean Program. Experimental results demonstrate that the redesigned trace compression algorithms are particularly effective for applications with inconsistent behavior across time steps and MPI tasks. In comparison to prior research, we deem SCALATRACE II a solid improvement towards exascale performance analysis.

7. REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, Apr. 2010.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[3] S. Budanur, F. Mueller, and T. Gamblin. Memory trace compression and replay for SPMD systems using Extended PRSDs. *SIGMETRICS Perform. Eval. Rev.*, 38(4):30–36, Mar. 2011.

[4] M. Burtscher. Vpc3: A fast and effective trace-compression algorithm. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 167–176, N.Y., June 2004.

[5] Community earth system model. http://www.cesm.ucar.edu/index.html.

[6] L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia. SIGMA: A simulator infrastructure to guide memory analysis. In *Supercomputing*, Nov. 2002.

[7] E. N. Elnozahy. Address trace compression through loop detection and reduction. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–215, 1999.

[8] Extrae. http://www.bsc.es/computer-sciences/extrae.

[9] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 81–90, 2005.

[10] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed. Scalable load-balance measurement for spmd codes. In *Supercomputing*, pages 1–12, 2008.

[11] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed. Clustering performance data efficiently at massive scales. In *Int'l Conf. on Supercomputing*, pages 243–252, 2010.

[12] T. Gamblin, R. J. Fowler, and D. A. Reed. Scalable methods for monitoring and detecting behavioral equivalence classes in scientific codes. In *IPDPS*, pages 1–12, 2008.

[13] M. Geimer, F. Wolf, B. J. N. Wylie, E. Abraham, D. Becker, and B. Mohr. The scalasca performance toolset architecture. In *International Workshop on Scalable Tools for High-End Computing*, June 2008.

[14] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A call graph execution profiler. In *Symposium on Compiler Construction*, pages 276–283, June 1982.

[15] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[16] A. Knupfer. Construction and compression of complete call graphs for post-mortem program trace analysis. In *International Conference on Parallel Processing*, pages 165–172, 2005.

[17] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. Introducing the Open Trace Format (OTF). In *Int'l Conf. on Computational Science*, pages 526–533, May 2006.

[18] S. Krishnamoorthy and K. Agarwal. Scalable communication trace compression. In *Conference on Cluster, Cloud and Grid Computing*, pages 408–417, 2010.

[19] J. R. Larus. Whole program paths. *SIGPLAN Not.*, 34(5):259–269, May 1999.

[20] J. Marathe, F. Mueller, and B. R. de Supinski. A hybrid hardware/software approach to efficiently determine cache coherence bottlenecks. In *Int'l Conf. on Supercomputing*, pages 21–30, June 2005.

[21] J. Marathe, F. Mueller, T. Mohan, B. R. de Supinski, S. A. McKee, and A. Yoo. METRIC: Tracking down inefficiencies in the memory hierarchy via binary rewriting. In *International Symposium on Code Generation and Optimization*, pages 289–300, Mar. 2003.

[22] J. Marathe, F. Mueller, T. Mohan, S. A. McKee, B. R. de Supinski, and A. Yoo. Metric: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. *ACM Transactions on Programming Languages and Systems*, 29(2):1–36, Apr. 2007.

[23] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[24] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2/3), 1997.

[25] C. G. Nevill-Manning and I. H. Witten. Linear-time, incremental hierarchy inference for compression. In *Proceedings of the Conference on Data Compression*, DCC '97, pages 3–, Washington, DC, USA, 1997. IEEE Computer Society.

[26] M. Noeth, F. Mueller, M. Schulz, and B. R. de Supinski. Scalatrace: Scalable compression and replay of communication traces in high performance computing. *Journal of Parallel Distributed Computing*, 69(8):969–710, Aug. 2009.

[27] V. Pillet, V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A tool to visualize and analyze parallel code. In *Proceedings of the 18th Technical Meeting of WoTUG-18: Transputer and Occam Developments*, pages 17–31, 1995.

[28] The parallel ocean program (POP), 1996. http://climate.lanl.gov/Models/POP/.

[29] P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz. Preserving time in large-scale communication traces. In *Int'l Conf. on Supercomputing*, pages 46–55, June 2008.

[30] M. Ronsse and D. Kranzlmueller. Roltmp-replay of lamport timestamps for message passing systems. *Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, page 0087, 1998.

[31] S. S. Shende and A. D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[32] J. Vetter and M. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2001.

[33] H. Wasserman, A. Hoisie, and O. Lubeck. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *The International Journal of High Performance Computing Applications*, 14:330–346, 2000.

[34] X. Wu, V. Deshpande, and F. Mueller. ScalaBenchGen: Auto-generation of communication benchmarks traces. In *International Parallel and Distributed Processing Symposium*, pages 1250–1260, 2012.

[35] X. Wu and F. Mueller. ScalaExtrap: Trace-based communication extrapolation for spmd programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 113–122, Feb. 2011.

[36] X. Wu, F. Mueller, and S. Pakin. Automatic generation of executable communication specifications from parallel applications. In *Int'l Conf. on Supercomputing*, pages 12–21, 2011.

[37] X. Wu, K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth. Probabilistic communication and i/o tracing with deterministic replay at scale. In *International Conference on Parallel Processing*, pages 196–205, 2011.

[38] Q. Xu, R. Prithivathi, J. Subhlok, and R. Zheng. Logicalization of MPI communication traces. Technical Report UH-CS-08-07, Dept. of Computer Science, University of Houston, 2008.

[39] Q. Xu and J. Subhlok. Construction and evaluation of coordinated performance skeletons. In *International Conference on High Performance Computing*, pages 73–86, 2008.

[40] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. Voelker. Mpiwiz: subgroup reproducible replay of mpi applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 251–260, 2009.

[41] J. Zhai, W. Chen, and W. Zheng. Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 305–314, 2010.

[42] J. Zhai, T. Sheng, J. He, W. Chen, and W. Zheng. FACT: Fast communication trace collection for parallel applications through program slicing. In *Proceedings of SC'09*, pages 1–12, 2009.