Memory Trace Compression and Replay for SPMD Systems using Extended PRSDs [*]

Sandeep Budanur Frank Mueller
North Carolina State University
Raleigh, NC, USA
mueller@cs.ncsu.edu

Todd Gamblin
Lawrence Livermore National Laboratory
Livermore, CA, USA
tgamblin@llnl.gov

ABSTRACT

Concurrency levels in large-scale supercomputers are rising exponentially, and shared-memory nodes with hundreds of cores and non-uniform memory access latencies are expected within the next decade. However, even current petascale systems with tens of cores per node suffer from memory bottlenecks. As core counts increase, memory issues will become critical for the performance of large-scale supercomputers. Trace analysis tools are thus vital for diagnosing the root causes of memory problems. However, existing memory tracing tools are expensive due to prohibitively large trace sizes, or they collect only statistical summaries and omit potentially valuable information.

In this paper, we present ScalaMemTrace, a novel technique for collecting memory traces in a scalable manner. ScalaMemTrace builds on prior trace methods with aggressive compression techniques to allow lossless representation of memory traces for dense algebraic kernels, with near-constant trace size irrespective of the problem size or the number of threads. We further introduce a replay mechanism for ScalaMemTrace traces, and discuss the results of our prototype implementation on the x86_64 architecture.

1. INTRODUCTION

This decade is projected to usher in exascale computing with systems three orders of magnitude more performant than the fastest high-performance computing (HPC) systems today. Harnessing such massive compute power will not be easy at such scale. Experience with current petascale systems, *e.g.*, Jaguar, BlueGene family installations, and RoadRunner, has shown that production codes tend to face scalability problems each time the core count is increased by a factor of 10. The causes of these problems are many-fold and require root cause analysis of application/system behavior. Today's tools fail to support root cause diagnosis at scale. Causes may only become apparent when cross-node correlation of symptoms is applied. But extensive trace collection across nodes becomes infeasible at scale.

The most common causes of scalability problems are communication, I/O, and memory inefficiencies. This work focuses on memory inefficiencies. In particular, we focus on SPMD codes that employ a multi-threaded shared-memory model on-node (such as OpenMP or POSIX threads) and a distributed memory model across nodes (such as MPI or Hadoop). Effective execution on multi-cores requires efficient use of the memory hierarchy across threads. Tools are required to analyze memory interactions, but most tools generate excessively long memory traces. Since the total size of these tools' trace data scales with the thread and core count, these tools will not scale to higher thread and core counts, and their analysis capabilities are severely limited. Some tools reduce trace size using statistical summaries, but lossy compression is difficult to use for scalability analysis.

To meet the challenges of performance analysis at exascale, we must depart from traditional approaches. Ideally, we would reduce data collection to only those metrics relevant to a symptomatic problem. Even this is difficult, though, as frequent probing on even a single thread generates excessive data for long-running exascale production jobs. The extracted data must be compressed on-the-fly either without loss of accuracy or with minor losses that retain the original application behavior for the sake of analysis.

Recent research in scalable compression of communication and I/O traces [1, 2] has demonstrated that the traces can be stored in near-constant size irrespective of the problem size or concurrency level. However, these traces do not reflect memory access patterns across threads, which are vital in identifying bottlenecks in memory hierarchies.

Contributions: We have developed a memory trace generator, a generic trace compression template library, and a signature tree library in C++. We have used these components to build a memory trace compression tool, ScalaMemTrace, that generates near-constant size memory traces that preserve the temporal order of accesses for dense algebraic kernels, irrespective of problem size and concurrency size. We have devised a novel abstraction of Extended Power Regular Section Descriptors (EPRSD) suitable for scalable trace compression on-the-fly. Our generic EPRSD template library, instantiated for memory tracing in this work, can be reused to ease the development of other PRSD-based trace compression tools for arbitrary application domains.

ScalaMemTrace implements a multi-level memory trace compressor involving intra-thread, inter-thread and inter-process compression schemes. A signature tree optimization further speeds up the trace compression process. This work also contributes an optimized trace generation process by implementing a per-function stack-walk approach instead of per-instruction stackwalks.

We experimentally assess the trace reduction capabilities of ScalaMemTrace. Our traces are orders of magnitude smaller than uncompressed traces and near-constant size for kernels with regular memory access patterns.

Overall, ScalaMemTrace is the first approach to provide scalable and complete memory traces instead of just partial traces, and it preserves the ordering of memory accesses

[*] This work was supported in part by NSF grants 0410203, 0429653, 0237570 (CAREER), 0937908, and 0958311. Part of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contracts DE-AC05-00OR22725, W-7405-Eng-48, and DE-AC52-07NA27344. LLNL-JRNL-486132

that they can be deterministically replayed. Preservation of memory reference order provides the basis for future work on scalable root cause analysis of memory inefficiencies. This work also provides novel opportunities for system simulators, such as Dimemas, SST etc. [3, 4, 5, 6, 7] to efficiently simulate memory artifacts and reason about performance in terms of scaling for larger problem sizes or node counts based on our proof-of-concept replay engine. ScalaMemTrace is also a key component of our ongoing effort to automatically generate benchmarks from large-scale applications at various levels of detail, including memory characteristics. Currently, such benchmark generation efforts are constrained to pure communication skeletons and computation is resembled by sleep intervals [8]. ScalaMemTrace provides the means to substitute these sleep intervals with sequences of replayed memory references for a respective code section between synchronization points. This would enable what-if analysis of auto-generated benchmarks assessing not only network interconnect characteristics but also memory hierarchy properties of hypothetical or actual architectures in simulation or actual runs, respectively.

2. MEMORY TRACE COMPRESSION

Our memory trace compression scheme is based on the PRSD abstractions [1, 9], but is more fine-grained and, hence, called Extended PRSDs (EPRSDs). EPRSDs preserve the order of memory references and generalize memory access patterns across threads and processes along with loop dependencies. EPRSDs differ from PRSDs in that EPRSDs additionally represent inter-thread dependencies. Our approach extracts complete memory traces that are orders of magnitude smaller than the conventional memory traces and near-constant in size irrespective of the problem size for dense algebraic kernels.

We rely on a binary instrumentation tool, Pin [10], to extract memory accesses from an application. Pin's instrumentation records an application's memory references, and the output is fed into a compressor module that dynamically combines the streams into a single compressed trace. The tool module is built using the C++ EPRSD template library, which handles dynamically merging incoming memory references into a near-constant size trace file. Figure 1 shows the data flow of our memory trace compressor.

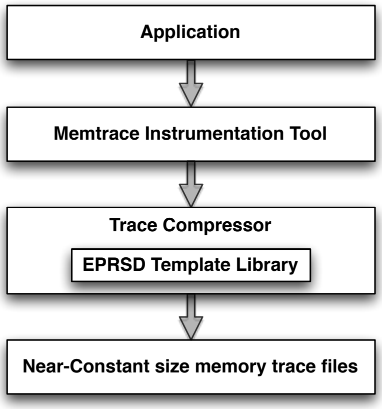
Application
Memtrace Instrumentation Tool
Trace Compressor
EPRSD Template Library
Near-Constant size memory trace files

Figure 1: Data Flow of the Trace Compressor

Pin is run as a set of MPI processes, either on a single node or across multiple nodes. Each Pin process instruments an SPMD program and writes the resulting memory trace to a named pipe. Named pipes were chosen to increase the versatility of the tool chain. Pin and the compressor operate conceptually on files, which allows users to optionally store per-thread traces. ScalaMemTrace connects multiple instances of Pin and compressor components via named pipes to avoid the overhead of storing intermediate files without compression. In parallel, compressor modules run in separate MPI processes and consume the memory traces from the pipe. Each pipe is uniquely named using the MPI rank of the Pin and compressor processes it connects. For example, Pin with rank 0 writes to a pipe named 'pipe0' and the compressor with rank 0 reads from the same pipe. This arrangement is depicted in Figure 2.

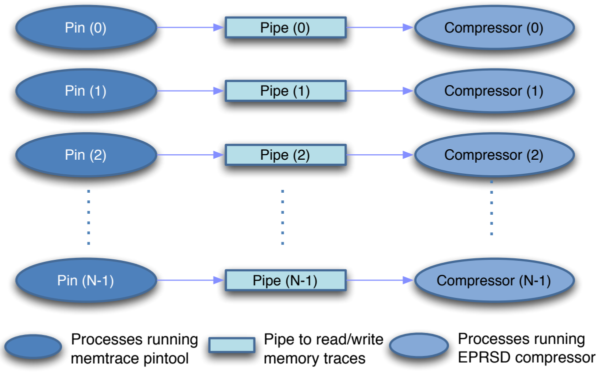

Figure 2: Design of the Memory Trace Compressor

Each compressor process generates EPRSDs for all the threads in its corresponding application process. It performs intra-thread and inter-thread merging before passing the EPRSDs to another compressor process for inter-process merging. Inter-process merging combines repetitive memory access patterns (or EPRSDs) across multiple processes of a SPMD application. The MPI compressor processes involved in an inter-process merge can be running on a single or different machines in a cluster. This results in order preserving, lossless and near-constant size memory traces for dense algebraic kernels, which can be used for replay and extrapolation. Our replay tool verifies the correctness of our compression scheme and can aid in the analysis of problem scaling. These techniques have proved highly scalable for communication tracing in ScalaTrace [1]. Notice that the order of memory references between synchronization points of threads is only loosely defined. It has been shown that an equal-rate processing of memory references across threads closely resembles the original coherence traffic [11]. Our work follows this scheme (but could also support other orderings).

2.1 Intra-thread Compression

Intra-thread compression exploits the repetitive behavior of applications. Regular Section Descriptors (RSDs) [1] capture load and store instructions within a loop in constant size and Power Regular Section Descriptors (PRSDs) capture RSDs nested in multiple loops [1]. EPRSDs extended PRSDs with the notion of thread dependencies. For example, $RSD1 :< 1000, ST_B, LD_A >$ represents alternating store and load instructions repeating 1000 times. $PRSD1 : < 100, RSD1, ST_D, LD_C >$ represents 100 occurrences of RSD1 loop followed by store and load instructions with

Algorithm 1 MERGE(newref)

Require: A memory reference newref.

```
add newref as tailnode
$rightTail = tailnode$
$leftTail$ = search matching node for tailnode
if $leftTail$ then
    $current = rightTail$
    while $current \neq leftTail.Next$ do
        $current = current.Prev$
        $leftHead$ = find match for current
        if $leftHead == NULL$ then
            break
        else
            $rightHead = current$
        end if
    end while
    if $current == leftTail.Next$ then
        merge the sequence ($rightHead$ ... $rightTail$)
        with ($leftHead$ ... $leftTail$)
    end if
end if
```

thread dependencies ignored. The code snippet in Figure 3 corresponds to the PRSD mentioned above.

```
for(i = 0; i < 100; i++) {
    for(j = 0; j < 1000; j++) b = a;
    d = c;
}
```

Figure 3: Sample Code for PRSDs

The algorithm for intra-thread compression is shown in Algorithm 1. The compression algorithm maintains a compressor object, which is a list of EPRSDs. While parsing the memory traces generated by Pin, new entries are appended to the list if no match is found, otherwise they are added to the matching window to find a repetitive sequence. The compression algorithm compares each memory reference in the input with a set of previous memory references, and it creates an RSD when a sequence of repetitions is found. The length of trace segments compared depends on the size of the window used to buffer the trace. The larger the window size, the higher the compression ratio and vice versa. To identify a loop of N memory references, a window size of at least $2N$ should be used for good compression. For M memory references, the algorithm runs in $O(M^2)$ time, if the window is not used and all previous references are compared. With a window of size S, the algorithm runs in $O(MS)$ time.

Figure 4 shows an example of intra-thread compression. The memory references op1, op2 and op3 are added to a list and matching patterns are found dynamically. Per Algorithm 1, on finding a matching sequence, the right portion is merged with the left portion and the RSD count is incremented. This process repeats for subsequent sequences of op1, op2 and op3 until the pattern disappears.

Intra-thread compression takes place during execution. Once execution completes, inter-thread memory trace compression begins. During trace generation, each instruction must be identified uniquely. Hence, a unique signature is computed for each instruction by performing a stack-walk. A series of program counter values form the whole signature and their values are XORed to compute the XOR-signature.

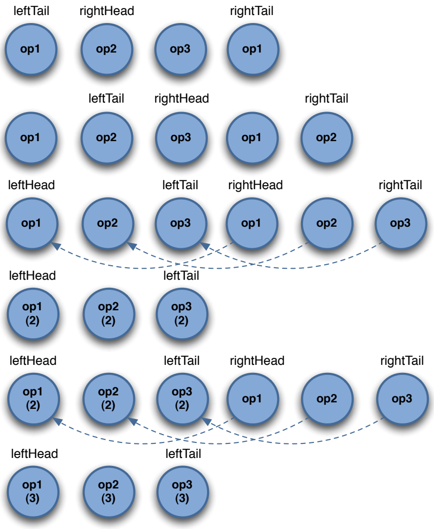

Figure 4: Example of Intra-thread Compression

XOR-signature matching is a necessary (but not sufficient) condition for EPRSD merging. XOR-signatures are compared to speed up the matching process. If two signatures match, a full signature match (a pairwise stack match) decides if EPRSDs are merged. The stack-walk mechanism for computing the signature is included in the instrumentation tool discussed earlier, and these signatures are part of the memory trace fed to the compressor tool. To speed up the signature matching process, we use a signature tree. We explain this optimization in more detail in later sections.

2.2 Inter-thread Compression

Intra-thread compression occurs on-line, and inter-thread compression begins after the instrumented application terminates. If the application is not multi-threaded, then the inter-thread compression step is skipped and inter-process compression begins. For a multi-threaded application, a separate compressor object maintains each thread's RSDs and PRSDs. After all threads of an application complete execution, RSDs and PRSDs of individual threads are compared and merged into an EPRSD when a match is found. PRSD lists are scanned for matching PRSDs with different thread identifiers but with the same signature. If regular memory access patterns are found then the base address for each EPRSD is represented as a function of the thread identifier.

As an example, *EPRSD1:<(0, K, 1),(1000, 400),(100, 4),ST_A* > denotes 100 occurrences of *store A* instruction with stride 4 and *base_address = (1000+400*thread_id)* such that $0 \leq thread\_id \leq K-1$. *(0, K, 1)* indicates that the pattern was found in K threads starting at 0 with stride 1. The OpenMP code in Figure 5 corresponds to this EPRSD.

```
MPI_Init(&argc, &argv);
#pragma omp parallel
{
    tid = omp_get_thread_num();
    for(j=400*tid; j<400*tid+100; j++) {
        a[j] =  j;
    }
}
MPI_Finalize(MPI_COMM_WORLD);
```

Figure 5: Sample Code for EPRSDs

A thread identifier's length is incremented and its stride is recomputed in the destination EPRSD at each stage of the merging process. For example, the final EPRSD $< T : 0, 4, 1 >$ indicates that the pattern occurred in four threads starting at 0 with a stride of 1. In our experiments, the number of threads was configured to be a power of two.

2.3 Inter-process Compression

An SPMD application runs as several processes each employing one or more threads. After the inter-thread compression, the EPRSDs of a process are merged with their matching counter-parts in other processes. Each process transmits the final list of EPRSDs to another process using MPI calls. The EPRSD list is scanned for matching EPRSDs with different MPI ranks but with the same signature. The binary radix tree approach is used to merge the inter-thread EPRSDs into inter-process EPRSDs. Hence, the inter-process compression completes in $(logN)$ steps, where N is the total number of processes in an SPMD application. If M EPRSDs are merged at each step, then the entire inter-process compression algorithm runs in $O(MlogN)$ time.

For example, $EPRSD1{:}< (0,\ N,\ 1),\ (0,T,1),\ (1000,\ 400),\ (100,\ 4),\ ST\_A >$ denotes 100 occurrences of *store A* instruction with stride 4 and $base\_address = (1000{+}400{*}thread\_id)$ such that $0 < thread\_id < T\text{-}1$ and $0 < node\_id < N\text{-}1$. $(0, T, 1)$ suggests the pattern was found in T threads starting at 0 with stride 1 and $(0, N, 1)$ suggests that the pattern is found in N processes starting at 0 with stride 1. The MPI-OpenMP code in Figure 5 corresponds to this EPRSD.

Figure 6 shows the radix tree used in inter-process compression for four processes. The repetitive memory reference patterns from different processes are combined into a single entity and other copies are discarded. A process identifier's length is incremented and its stride is recomputed in the destination EPRSD at each stage of the merging process. For example, the final EPRSD $< P : 0, 4, 1 >$ in the compressor object of process 0 indicates that the pattern occurred in four processes starting at 0 with stride 1. The same pattern applies to larger numbers of processes. In our experiments, the number of application processes and compressor processes were configured to be powers of two.

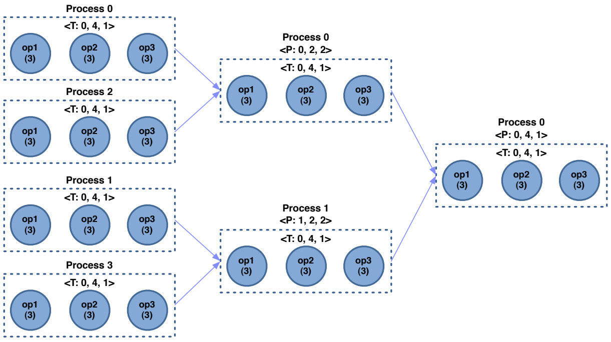

Figure 6: Design of Inter-process Compression

When the inter-process compression completes, the compressor object in process 0 contains the final list of EPRSDs merged from all compressor processes. This list of EPRSDs is saved into a file that of near-constant size, regardless of problem size, OpenMP thread count, or MPI process count.

3. MEMORY TRACE GENERATION

This section describes the components of ScalaMemTrace, our trace compression implementation. We describe the binary instrumentation, intermediate data structures, and other techniques we use to ensure scalability in the number of memory references within threads, across threads and across processes on different compute nodes.

3.1 Binary Instrumentation

Pin [10] is a dynamic instrumentation tool that can dynamically enable and disable instrumentation without incurring unnecessary overhead. Pin employs a just-in-time compiler (JIT) to instrument a binary at runtime. Pin consists of a virtual machine (VM), a code cache and an instrumentation API used by Pintools. Pin runs in user space and hence can instrument only user-level code. Three binary programs execute while an instrumented program is executing — Pintool, Pin and the application. Our memory tracing tool, ScalaMemTrace, runs as a Pintool that instruments MPI/OpenMP applications. Our tool instruments only load and store instructions in the application. For each load and store instruction, an entry is added to an EPRSD compressor object. As instrumentation continues, we compress intra-thread data on-the-fly until execution completes.

3.2 Stack walking

Stack walking is performed to obtain unique stack signatures for each memory reference and to help in matching references during the compression phase. Two different Stackwalkers are integrated in ScalaMemTrace. *1) ver0 Stackwalker*: obtains a series of return addresses by naively walking the stack frames using the current frame-pointer value returned by Pin; *2) $DynStackwalkerAPI$*: a freely available stackwalking library. Only the current stack frame information is provided during initialization. The remaining operations are managed by the library. This option is much slower than the ver0 stack-walk, but in some cases the ver0 stackwalker fails to identify stack frames due to compiler optimizations (e.g: Intel C Compiler), while DynStackwalkerAPI correctly generates accurate stack signatures. DynStackwalkerAPI also helps to obtain consistent signatures when shared library load addresses vary among processes.

A complete stack signature for every memory reference helps to preserve the program structure, whereas a naive approach of merging memory references based on only program counter values results in better compression but compromises the program structure. We use complete stack signatures, which help to preserve program structure.

3.3 Problems in Unique Signature Generation

Intel x86 is a CISC architecture, where multiple memory operations may be performed by a single instruction. For example, an increment instruction may perform a load, an add, and a store. The same signature is thus used for two different memory operations. To prevent false EPRSD matches and compression errors, we detect such instances during instrumentation and ensure uniqueness by XORing the instruction type with the PC value in the signature.

A stack-walk need not be performed for every memory reference. It is sufficient to walk the stack on function entry and exit points only, and corresponding PC values are appended to the signature for memory references within a function. Pin cannot always detect corresponding CALL and RET instructions accurately for instrumentation, as we found a mismatch between the number of CALL and RET instructions that it instrumented. In such cases, the signatures only reflect the partial call path, skipping undetected differences. For all such occurrences across all threads and processes, we can still identify the memory references uniquely. The anomalies during function boundary detection are rare and do not adversely affect the compression process. We detail the speedup of this approach in the results section.

4. EPRSD TEMPLATE LIBRARY

We have developed a C++ template library to facilitate the rapid development of EPRSD trace compression tools for high-performance applications. Users can derive classes and/or define their own data types to store trace data and compress them by providing just two objects. C++ classes are designed for intra-thread, inter-thread and inter-process compression. Most importantly, they are independent of any message-passing APIs. Users can use our library in combination with any message-passing implementation. We have implemented an MPI version of intra-thread, inter-thread and inter-process memory trace compression schemes.

4.1 Signature Trees

A signature tree is constructed as a separate C++ module and the classes are used by the EPRSD template library. The signature tree offers faster comparison of stack signatures than the XOR signature approach during compression.

For any two EPRSDs, if the XOR signatures do not match, then signatures are different and pair-wise comparison is not needed. If the XOR signatures match, then a pair-wise comparison of signatures is required. This pair-wise comparison is costly when matches are frequent within loops. The number of comparisons depends on the signature length. When a signature tree is used, two EPRSDs can be compared by simply testing if they point to the same leaf node in the signature tree. Two EPRSDs match if and only if they point to the same leaf node in a signature tree. Also, this comparison completes in constant time.

During inter-process compression, EPRSDs are exchanged between processes. The complete signature list must be transferred for every EPRSD if XOR signatures are used. This adds significant communication overhead. If a signature tree is used, the signature list of every EPRSD need not be exchanged. The signature tree must be transmitted once before the inter-process merging can take place followed by the transmission of EPRSDs with only a reference to the leaf node in the previously transmitted signature tree. This reduces the communication overhead significantly.

5. RESULTS

For our experiments, we used hybrid MPI/OpenMP versions of the Sequoia AMG [12], matrix multiplication, and vector addition micro-benchmarks. We used power-of-two thread counts and process counts. For AMG, we used four OpenMP threads and varied the number of processes from 1 to 16 with constant problem size (strong scaling). For matrix multiplication and vector addition benchmarks, we varied OpenMP thread count from four to 64 and kept problem size proportional (weak scaling). All experiments used a 16-node, dual-core, 2-way SMP 64-bit Opteron cluster at NCSU.

5.1 Compression

In this section, we report the results of our experiments with ScalaMemTrace. Files with EPRSD compressed traces also contain loop and thread dependency information along with the address and reference type. This includes memory address information (references within a loop, iteration count, starting address, address stride), thread information (number of threads, starting thread-ID, thread-ID stride) and node information (number of processes, starting node-ID, node-ID stride). Reported compressed trace file sizes include all of these details.

Figure 7 shows the size of the original traces and EPRSD-compressed traces of the vector addition microbenchmark with weak scaling. Each thread operates on partitions of two large EPRSD-compressed trace files for the vector addition integer arrays A and B and stores the result in another array C at the corresponding offset. The computation is $C[i] = A[i] + B[i]$, where i is a function of the thread id. Weak scaling is applied by increasing the array size proportionally with the number of threads. The figure illustrates the scalability of the EPRSD compression scheme for different concurrency levels and problem sizes. The compressed trace file size remained nearly constant even when problem size and number of threads were increased proportionally.

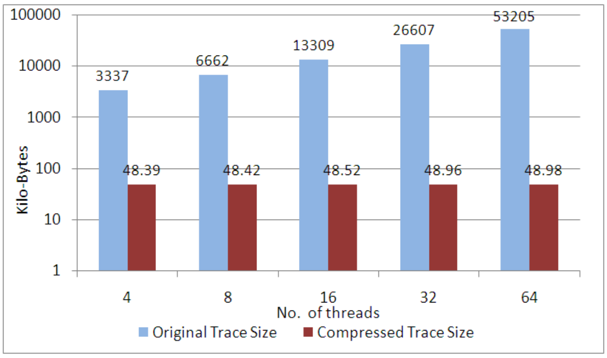

Figure 7: Weak Scaling Trace Size (Vector Addition)

Figure 8 shows the size of the original traces and EPRSD-compressed traces for the matrix multiplication microbenchmark with weak scaling. The figure illustrates the scalability of the EPRSD approach for different concurrency levels and problem sizes. The compressed trace file size was not constant, but it was an order of magnitude less than the original trace file size when the problem size and number of threads were increased proportionally. When the number of threads were increased from 4 to 64, the original trace file grew 55 times, but the compressed size increased only 3 times. For 16 threads, a larger number of RSDs were merged with fewer breaks in the sequence compared to 8 threads. Hence, the compressed trace size has reduced slightly.

Figure 9 shows the size of the original and EPRSD compressed trace files for the AMG benchmark. Each MPI process uses four OpenMP threads. The figure demonstrates the scalability of the EPRSD compression scheme for different concurrency levels. We keep the problem size steady and vary only the number of MPI processes (strong scaling). RSDs at the intra-thread level do not merge completely as sequences are separated due to branching and non-rectangular loops. The compressed trace file size grew linearly so that the size was reduced by half.

It should be noted that the scales are logarithmic. The raw trace file size increases exponentially with the number of

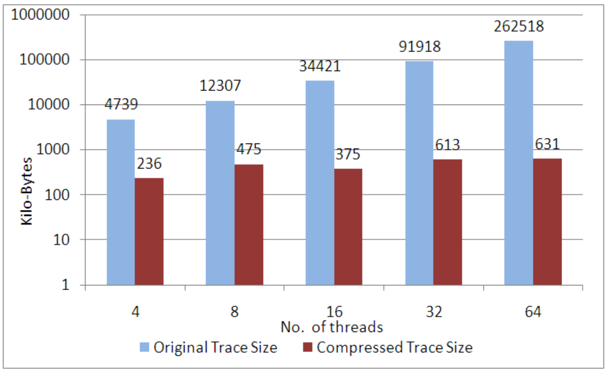

Figure 8: Weak Scaling Trace Size (Matrix Mult.)

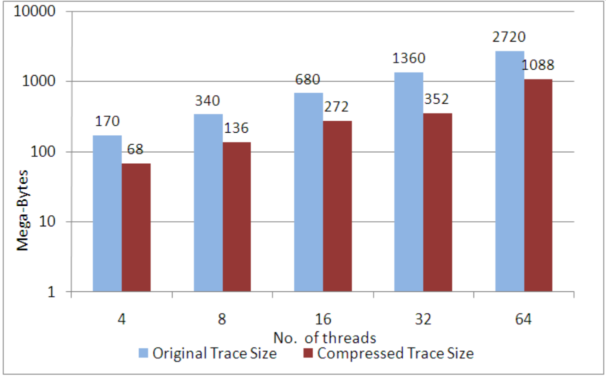

Figure 9: Trace Size for AMG Benchmark

threads. In contrast, the EPRSD trace file size remains almost constant for vector addition and grows sub-linearly for matrix multiplication. For the AMG benchmark, the compressed trace file size increases linearly but trace files were compressed by 50%. From the results, we can conclude that the space savings due to the EPRSD compression scheme is exponential and resulting traces are highly scalable for dense algebraic kernels, but linear for the AMG benchmark.

We verified the correctness of our compression scheme by replaying the traces using our replay tool. Vector addition and matrix multiplication compressed traces were replayed with 100% accuracy. AMG traces were replayed with 91% accuracy. This error is due to round-off errors caused by integer division in the compression algorithm.

5.2 Performance

In this section, we discuss the runtime performance of the instrumentation stage, the stack-walk and levels of compression supported by ScalaMemTrace. "Matmul 24x24" refers to the matrix multiplication benchmark with four OpenMP threads operating on 24x24 matrices. "Matmul 48x48" refers to the matrix multiplication benchmark with eight OpenMP threads operating on 48x48 matrices. "AMG n=1" refers to the AMG benchmark with one MPI process and four OpenMP threads. "AMG n=2" refers to AMG with two MPI processes and four OpenMP threads in each process.

Instrumentation adds overhead to the runtime of an executable. Even when instrumentation is disabled, application runtime increases when executed within Pin [10]. This overhead is due to the additional time required to execute Pin itself. The difference in application runtime within Pin with instrumentation turned on and off is shown in Figure 10. This difference is due to the additional overhead involved in executing dynamically rewritten application code snippets. The difference in runtime with regular and optimized stack-walk is also presented in the figure. For the "AMG n=2" case, the benchmark runs on two nodes in parallel and the reported measurement is the parallel execution time. The application runtime when executed on two nodes is larger due to additional MPI overhead. The instrumentation overhead is much larger than the original execution time irrespective of stackwalk optimization. From the results we can deduce that a larger number of instructions (due to parallelization overhead) is executed but fewer function calls (due to strong scaling) per node are made in case of "AMG n=2" compared to "AMG n=1". Hence, the instrumentation time with per-function stackwalk is lowered whereas the instrumentation time with per-instruction stackwalk has increased in "AMG n=2" compared to "AMG n=1" case. The optimized stack-walk involves tracing the stack once per function call whereas a regular stack-walk involves tracing the stack on every memory reference instruction. The performance speedup varied between 30% to 50%. From the results, we conclude that an optimized (per-function) stack-walk is significantly more efficient than a regular (per-instruction) stack-walk.

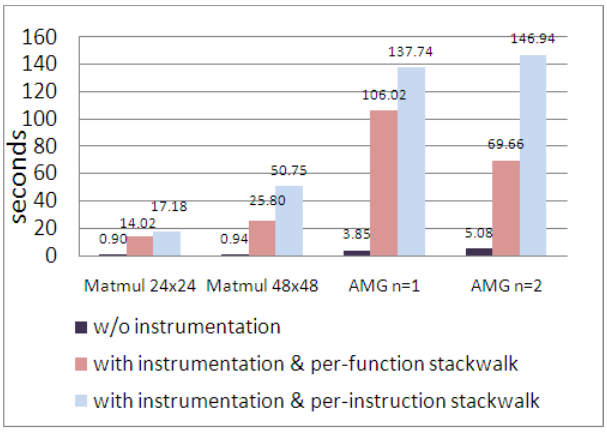

Figure 10: Instrumentation Overhead

The influence of optimized (per-function) stackwalk on the overall instrumentation time is depicted in Figure 11. Per-function stackwalking contributed only a minor portion of the overall instrumentation time while the major overhead is due to the instrumentation code and Pin overhead.

Table 1 shows the runtimes of various levels of compression for our benchmarks. The first three entries do not involve inter-process compression. Hence, only intra-thread and inter-thread compression runtimes are considered. In the corresponding Figure 12, we see that intra-thread compression time is almost equal to the instrumentation time. This is because intra-thread compression occurs on-the-fly and completes soon after the instrumentation terminates. Inter-thread compression runtime depends on the number of EPRSDs in each thread's compressor object after intra-thread compression. This number varies widely depending on the benchmark and runtime parameters. For matrix multiplication, inter-thread compression time is low because the resultant list of EPRSDs after intra-thread compression is much shorter than that of AMG. Inter-process compression incurs MPI communication overhead in addition to the merging overhead, which depends on the number of EPRSDs in each process's compressor object after inter-thread compression. For "AMG n=2", inter-process compression domi-

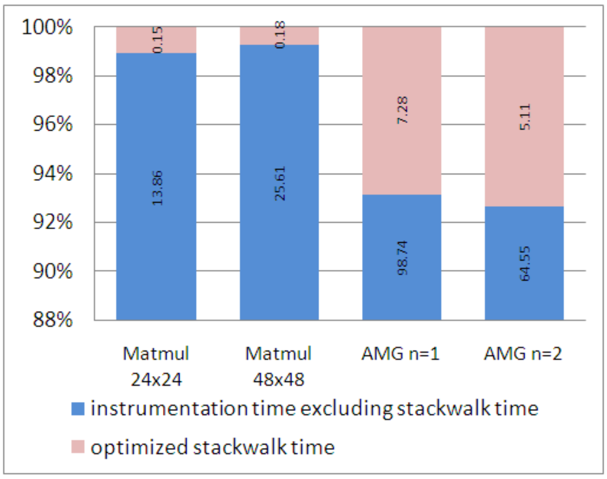

Figure 11: Stackwalk and Instrumentation Runtime

nates the compression time due to MPI overhead and merging of large numbers of EPRSDs across processes.

Table 1: Compression Runtime

Benchmark	**Intra-thread compression (msec)**	**Inter-thread compression (msec)**	**Inter-process compression (msec)**
Matmul 24x24	14,071.198	1.632	0.000
Matmul 48x48	25,854.444	8.064	0.000
AMG n=1	112,623.903	4,660.802	0.000
AMG n=2	70,046.435	109,836.724	1,163,648.747

6. RELATED WORK

RSDs were first proposed in [13] to analyze array accesses. PRSDs were used to compress memory traces in [9] to analyze cache coherence problems in OpenMP programs. This work did not represent addresses as function of thread IDs and did not address inter-process memory trace compression. SIGMA [14] is a data collection framework and a set of cache analysis tools that employs online trace compression by exploiting loops (similar to RSDs) but do not capture thread and process level dependencies.

Caches As Filters [15] is an analytical framework for analyzing and designing caches. This work introduces TSpec notation to represent memory references in a compact format. TSpec is more complex than RSDs and represents the state of a caching system, but the relation between memory references and threads is not gathered. Memory address trace compression through loop detection in a multi-programmed environment was described in [16], but it did not address the compression of traces in a cluster environment. Traces captured using such tools do not scale with the the number of threads or processes in a HPC environment.

Memory tracing has been explored in many facets, including performance counter analysis and concepts of reuse distance to analyze programs and cache behavior [6, 17, 18, 19, 20, 21, 22, 23, 24, 25]. Our work differs in that is further develops concepts of in-situ compression from ScalaTrace [1] and METRIC [9, 26, 27, 28, 29]. ScalaTrace addresses intra-task and inter-process compression of communication traces, but not memory traces. Also, ScalaTrace does not involve inter-thread compression. Trace compression discussed in [30] is based on statistical sampling and results in lossy compression and do not preserve order. ScalaMemTrace benefits from similar compression rates as METRIC due to the utilization of PRSDs, which has been shown to exceed even the best general-purpose compression schemes [11]. In this work, the PRDS compression similar to the one utilized by ScalaMemTrace has not only been shown to be applicable to dense matrix codes but also to the SPEC FP benchmark suite.

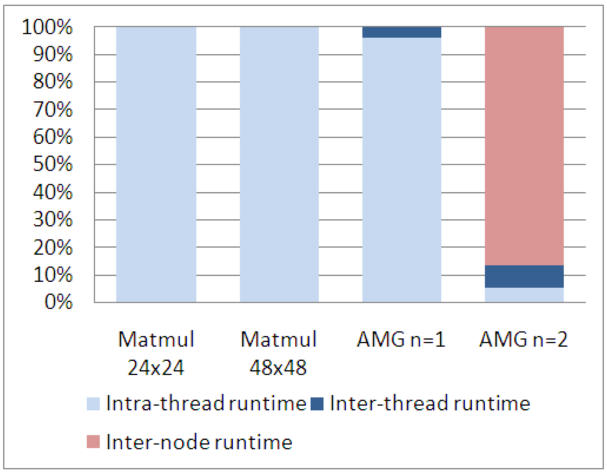

Figure 12: Compression Runtime

Our work addresses lossless, order preserving intra-thread, inter-thread and inter-process compression of memory traces. We also offer an EPRSD template library developed in C++, for the rapid development of EPRSD compression tools for HPC applications. Our work incorporates an optimization to speed up the memory trace compression process by using signature trees. Also, a separate reusable C++ module, SIGTREE, was developed to assist the development of tools needing signature tree functionality. In our memory compressor tool, ScalaMemTrace, both the XOR signatures and signature tree options are available, which can be configured at compile time. ScalaMemTrace also incorporates two different versions of Stackwalker libraries, a simple frame pointer traversal (Ver0) and the Wisconsin stackwalker [31], configurable at compile time.

ScalaMemTrace can be used along with the existing cache performance analysis tools [9, 15] to analyze cache performance in multi-threaded applications. ScalaMemTrace can also be integrated with communication tracing tools, such as ScalaTrace [1, 2], to combine online communication, I/O and memory trace compression of HPC applications.

7. CONCLUSION

Memory traces of multi-threaded SPMD applications are very large and do not easily facilitate analysis. Existing memory trace tools either produce unmanageable large lossless trace files or produce lossy statistical summaries.

We have developed ScalaMemTrace, a unique memory tracing framework that combines small, near-constant trace sizes with lossless compression. Our tool extracts full memory traces of dense algebraic kernels with near-constant size regardless of thread count or problem size. ScalaMemTrace preserves memory access details along with reference order.

Our scheme builds on previous work with Extended Power Regular Section Descriptors (EPRSDs), the basis of our efficient trace representation. We capture repetitive behavior at all levels of a hybrid application: within threads, among threads, and among processes, which enables us to eliminate both sequential and parallel redundancy in the trace. Finally, we have developed a replay mechanism that allows traces to be streamed on-the-fly without full decompression.

The level of compression achieved depends on the program structure. Some benchmarks used in our experiments have rectangular loops and the execution order across threads and processes is nearly identical. In such cases, the compressed trace size has remained nearly constant. Other benchmarks have non-rectangular loops and branches, resulting in variations in inter-process execution order and linear scaling of the trace size. Compression levels are thus an indicator of a program's structure and its dynamic behavior. A near-constant size of compressed trace files indicates a highly synchronous SPMD application, while a poor compression indicates an irregular code.

We achieved near-constant size compression for dense algebraic kernels (Matrix Multiplication and Vector Addition). For AMG, trace size grew linearly with process count.

Our work provides the basis for scalable root cause analysis into memory inefficiencies based on such aggressively compressed traces. Beyond analysis, the work also provides novel opportunities for system simulators, such as Dimemas, SST etc. [3, 4, 5, 6, 7]. It is also a key component of our on-going effort to automatically extract benchmarks from large-scale application at various levels of details, including the memory characteristics. Future work will address these areas based on the capabilities of ScalaMemTrace.

ScalaMemTrace is available under an open-source license from http://moss.csc.ncsu.edu/ mueller/ScalaTrace.

8. REFERENCES

[1] 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, August 2009.

[2] Karthik Vijayakumar, Frank Mueller, Xiasong Ma, and Philip C. Roth. Scalable multi-level I/O tracing and analysis. In *Petascale Data Storage Workshop*, November 2009.

[3] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A tool to visualise and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44 of *Transputer and Occam Engineering*, pages 17–31, April 1995.

[4] Jesús Labarta, Sergi Girona, and Toni Cortes. Analyzing scheduling policies using dimemas. *Parallel Computing*, 23(1-2):23–34, 1997.

[5] Sergi Girona, Jesús Labarta, and Rosa M. Badia. Validation of dimemas communication model for MPI collective operations. In *European PVM/MPI Users' Group Meeting*, pages 39–46, 2000.

[6] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Supercomputing*, November 2002.

[7] Arun F Rodrigues, Richard C Murphy, Peter Kogge, and Keith D Underwood. The structural simulation toolkit: exploring novel architectures. In *Poster at the 2006 ACM/IEEE Conference on Supercomputing*, page 157, 2006.

[8] X. Wu, F. Mueller, and S. Pakin. Automatic generation of communication specifications from parallel applications. In *International Conference on Supercomputing*, page (accepted), June 2011.

[9] 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, March 2003.

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[11] 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, April 2007.

[12] The ASC sequoia benchmark codes, 2008. https://asc.llnl.gov/sequoia/benchmarks.

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

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

[15] D. Weikle, S. McKee, Kevin Skadron, and W. Wulf. Caches as filters: A framework for the analysis of caching systems. In *Grace Murray Hopper Conference*, September 2000.

[16] 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.

[17] Jonathan Weinberg, Michael O. McCracken, Erich Strohmaier, and Allan Snavely. Quantifying locality in the memory access patterns of HPC applications. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 50, Washington, DC, USA, 2005. IEEE Computer Society.

[18] D.H. Bailey and A. Snavely. Performance modeling: Understanding the present and predicting the future. In *Euro-Par Conference*, August 2005.

[19] Engin İpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 195–206, 2006.

[20] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, 2002.

[21] E. Johnson and C. Schieber. Ratchet: Real-time address trace compression hardware for extended traces. *Performance Evaluation Review*, 21(3):22–32, 1994.

[22] A-T. Nguyen, P. Bose, K. Ekanadham, A. Nanda, and M. Michael. Accuracy and speed-up of parallel trace-driven architectural simulation. In *Proceedings of the 11th International Parallel Processing Symposium (IPPS'97)*, Geneva, Switzerland, April 1997. The Institute of Electrical and Electronics Engineers. IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

[23] T. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 191–202, June 2001.

[24] T. Chilimbi. Dynamic hot data stream prefetching for general-purpose programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*,

pages 191–209, June 2002.

[25] 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.

[26] J. Marathe, A. Nagarajan, and F. Mueller. Detailed cache coherence characterization for openmp benchmarks. In *International Conference on Supercomputing*, pages 287–297, June 2004.

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

[28] J. Marathe, F. Mueller, and B. R. de Supinski. Analysis of cache coherence bottlenecks with hybrid hardware/software techniques. *ACM Transactions on Architecture and Code Optimization*, 3(4):390–423, December 2006.

[29] J. Marathe, F. Mueller, T. Mohan, S. A. McKee, B. R. de Supinski, and A. Yoo. Source-code correlated cache coherence characterization of openmp benchmarks. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):818–834, June 2007.

[30] Todd Gamblin, Rob Fowler, and Daniel A. Reed. Scalable methods for monitoring and detecting behavioral equivalence classes in scientific codes. In *International Parallel and Distributed Processing Symposium*, 2008.

[31] Paradyn, parallel tools project - Wisconsin stackwalker library. http://www.paradyn.org/html/stackwalker1.1-features.html.