## ABSTRACT

BUDANUR RAMANNA, SANDEEP. Memory Trace Compression and Replay for SPMD Systems using Extended PRSDs. (Under the direction of Dr.Frank Mueller.)

Analyzing the memory traces of multi-threaded SPMD programs is a cumbersome and expensive process due to large trace size, program complexity and long running times. Though many binary instrumentation tools generate memory traces, they either gather statistical information with loss of details or generate large trace files that are difficult to handle. Our approach provides near-constant size memory traces for dense algebraic kernels irrespective of the problem size or number of threads involved while preserving the memory access details along with the order in which memory references are issued. Our scheme not only compresses loops but also groups similar memory access patterns across threads and processes into a single entity called Extended Power Regular Section Descriptor (EPRSD), which is an enhancement over the Power Regular Section Descriptor (PRSD) concept. We introduce a multi-level compression scheme exploiting memory access patterns in loops, thread dependences and process dependences that are capable of extracting an application's memory access structure. We further introduce a replay mechanism for the traces generated by our approach and discuss results of our prototype on the X86-64 architecture. Considering all the above features makes the EPRSD mechanism a promising approach for scalable memory trace compression and replay.

# Memory Trace Compression and Replay for SPMD Systems using Extended PRSDs

by
Sandeep Budanur Ramanna

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

Computer Science

Raleigh, North Carolina

2010

APPROVED BY:

| | |
|---|---|
| Dr.Xiaosong Ma | Dr.Xuxian Jiang |

Dr.Frank Mueller
Chair of Advisory Committee

## DEDICATION

Dedicated to my family.

# BIOGRAPHY

Sandeep was born in a small city called Mandya, located in the southern part of India. The place is well known for its lush green farmlands and sugar factories. He was brought up and educated in the city of his birth. After graduating in 2003 with a degree in Computer Science and Engineering, he moved to a nearby city called Bangalore. He worked in telecommunication software companies porting mobile platforms for clients around the globe. In the fall of 2008, he joined NC State University to pursue his Master's degree in Computer Science.

## ACKNOWLEDGEMENTS

First and foremost, I would like to thank Dr.Frank Mueller for his guidance, support and faith in me. I am grateful to Todd Gamblin, Bronis de Supinski and Martin Schulz for their valuable suggestions during the development of EPRSD template library. I would like to thank Dr.Xiaosong Ma, Dr.Xuxian Jiang and all other members of the System Research group. I would like to thank the members of pinheads online discussion group for their inputs on writing binary instrumentation tools. I would like to thank all my labmates for their help and inputs. I would also like to thank all my roommates for being cooperative.

# TABLE OF CONTENTS

# LIST OF TABLES

## LIST OF FIGURES

# Chapter 1

# Introduction

## 1.1 Background

Supercomputers are used in high performance computing due to their huge computational speed and have been popular for a few decades. Supercomputers were first introduced in 1960s and became popular in 1970s and 1980s. In the mid-1990s, the supercomputer market declined but since then, the interest in supercomputing has increased significantly in the last decade. It is a well known fact that today's supercomputers are tomorrow's ordinary computers. Today, ORNL's Jaguar is the fastest supercomputer with 1.75 peta flops, China's Nebulae occupies the second spot with 1.27 peta flops and Roadrunner at LANL, which was the fastest in 2008, holds the third spot with 1.04 peta flops performance as of June 2010.

Supercomputers are used for highly computation-intensive tasks such as problems related to quantum physics, molecular biology, weather forecasting, climate research, nuclear physics, aeronautics, astro physics and grand-challenge problems. Supercomputers are vital for problem solving in these domains because they not only speed up the computation but also solve problems that are otherwise unsolvable without huge computational power. Supercomputers are extensively used in the field of fluid dynamics to better understand turbulence, which was considered an unsolved problem of classical physics. Supercomputers running Computational Fluid Dynamics algorithms have simulated more realistic flight conditions than the highly expensive wind-tunnel method, thus reducing the cost.

Early supercomputers were very fast scalar processors, which were succeeded by vector processor machines. These machines were very expensive and,hence, focus shifted to massively parallel machines composed of off-the-shelf processors called commodity clusters combined with custom interconnects. At NC State university, the OPT cluster consists of AMD Opteron machines [25]. Another Sony PS3 cluster [26] consists of playstation gaming consoles. The

processors in a cluster, often called nodes, run the same program but operate on different data sets. This model is called Single Program Multiple Data (SPMD). The clusters are programmed using the message-passing programming model. They exchange messages for transferring data or synchronization. MPI is the most widely used message-passing model in the HPC industry.

Due to the limitations in uniprocessor design, symmetric multi-processors (SMPs) became popular several years ago. SMPs have multiple identical processors sharing memory and are often interconnected by a bus. Such SMP architectures are not scalable due to bus contention, which limits the number of processors connected to a bus. The most common programming model used in SMPs is OpenMP. Modern day clusters are composed of individual nodes that are in turn SMPs. This gives rise to a hybrid programming model where parallel applications use MPI for inter-node communication and OpenMP for shared memory parallel programming.

We limit the discussion to the above two topics only though other high-performance solutions exist in practice.

## 1.2 Motivation

SPMD applications employ multiple threads of execution, each on their own core operating on different data. Such programs tend to have memory access patterns that result in large memory traces. Effective execution on multi-cores requires efficient use of the memory hierarchy across threads. To analyze the memory access pattern of threads, tools are required. Most of the tools operate on excessively long memory traces. Since tools neither scale with the number of threads (or cores) nor the problem size, their analysis capabilities are severely limited. Some tools provide only statistical information of memory accesses in order to reduce trace size but are lossy and hardly useful for scalability analysis. Tools combining the best of both the worlds are desirable.

Recent research in the scalable compression of traces [1] has demonstrated that the traces can be stored in near constant size format irrespective of the problem size or concurrency. However, this was demonstrated with communication traces that do not reflect the memory access patterns across threads. Memory access patterns across threads are vital in identifying the memory bottlenecks while using memory hierarchies.

## 1.3 Hypothesis

We hypothesize that lossless, near-constant size memory traces that are highly scalable can be obtained by compressing memory traces for regular SPMD programs.

Resulting traces should be orders of magnitude smaller than the conventional memory traces

and near-constant in size irrespective of the problem size and concurrency.

## 1.4 Contributions

We have developed a memory trace generator, a generic trace compressor module as a C++ template library and a signature tree library in C++. These are combined to build a memory trace compressor tool that generates near constant size memory traces preserving the temporal order of accesses for dense algebraic kernels, irrespective of problem size and concurrency size. Our approach is based on the PRSD abstractions [1] but more fine-grained and, hence, called Extended PRSDs (EPRSDs). EPRSDs preserve the order of memory references and generalize memory access patterns across processes, threads and loops. The EPRSD template library can be reused to ease the development of other PRSD-based trace compression tools.

We have implemented a multi-level memory trace compressor involving intra-thread, inter-thread and inter-node compression schemes. We further implemented an optimization technique, signature trees, which speeds up the trace compression process. We have also optimized the trace generation process by implementing a per-function stack-walk approach instead of per-instruction stackwalks.

## 1.5 Evaluation

We evaluate the compressor tool using benchmarks involving both OpenMP and MPI code. The compressor tool is run on the OPT cluster at NC State university [25] for matrix multiplication, vector addition micro-benchmarks and AMG benchmark of Sequoia MPI and OpenMP benchmark suite. The proposed mechanism is platform independent, but due to time and resource constraints, we compressed memory traces from up to 16 processes, each process contributing one million memory references to obtain results at inter-node level. Also, the number of OpenMP threads within each process (for microbenchmarks only) was varied to collect the compressed traces at intra-thread and inter-thread level. Experimental results demonstrate that the proposed mechanism compresses the memory traces in near-constant size irrespective of the problem size and concurrency. Replay of memory traces is almost accurate except that some additional traces are issued due to round-off errors caused by integer division.

Overall, the results indicate that memory trace compression scales with the number of threads, processes and problem size. The compressed traces can be easily viewed for analyzing the loop and thread dependences. The smaller trace size improves the portability and replay can be performed without decompressing the entire trace. This can replace the complex tools requiring large amount of memory, disk space and processing power in managing memory traces

from many nodes.

## 1.6 Summary

To summarize, we propose a lossless and order preserving memory trace compressing scheme that produces near constant size traces capturing loop, thread and process dependences for dense algebraic kernels. We also propose optimizations to the trace compression process by making use of signature trees.

# Chapter 2

# Scalable Trace Compression

This work on memory trace compression is based on previous work on online communication trace compression called ScalaTrace [1]. ScalaTrace showed that compressed trace file sizes could be obtained with near constant size or, in some other cases, orders of magnitude smaller than the original trace size irrespective of the number of nodes or application run time. ScalaTrace recognizes loops dynamically and merges the repetitive entries into a single entity called RSD (Regular Section Descriptor) that represents traces in constant size. PRSDs (Power Regular Section Descriptor) represent nested loops by arranging the RSDs recursively. The challenge is that ScalaTrace dealt with the compression of communication traces, which is not directly applicable to compressing memory traces. A separate framework had to be developed to compress the memory traces dynamically. Our work focuses on the development of such a framework.

## 2.1 Instrumentation

Our memory trace compression tool builds on a freely available binary instrumentation tool to generate memory traces of load and store instructions. We have used Intel's Pin tool for binary instrumentation to generate memory trace dynamically [16]. This trace consists of instruction-type, accessed memory address, instruction-pointer, and stack signature. During trace generation a filter is used to separate application-specific instructions from system related instructions. This is achieved by extracting the range of addresses when the application image is loaded. Instructions only within this address range are included in trace generation. (Instructions executing prior to main() to initialize stack and registers are ignored as they do not contribute to the application execution or memory analysis). This trace is fed to the compressor module, which subsequently constructs EPRSDs to compress the traces.

## 2.2 Memory Trace Compression

Our memory trace compression scheme is based on the PRSD abstractions [1][3], 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 dependences. EPRSDs differ from PRSDs in that EPRSDs additionally represent inter-thread dependencies. Our tool 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.

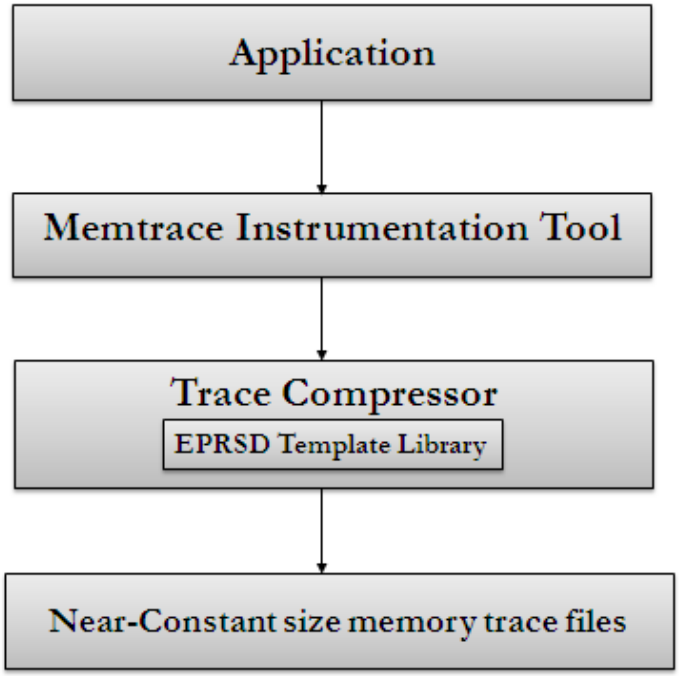


Figure 2.1: Data Flow Diagram of the Memory Trace Compressor

We rely on a binary instrumentation tool, Pin [16], to extract memory addresses from an application. This instrumentation tool filters application-specific memory references. The output from the binary instrumentation tool is a series of memory references, which is fed into the compressor module that combines the memory references into a single compressor object dynamically. The compressor module is built using the C++ EPRSD template library, which handles the dynamic merging of incoming memory references into a near-constant size compressed trace file. The data-flow diagram of the memory trace compressor is depicted in

Figure 2.1.

Pin is run as a set of MPI processes, either on a single node or across multiple nodes, where each process instruments a SPMD program separately and writes the generated memory trace to a pipe associated with the process. Similarly, the compressor module runs in parallel as multiple MPI processes on one or more nodes. Pin and compressor processes are identified uniquely by their MPI ranks in their respective communication domain. The same rank is used to uniquely name the pipes, which serve as a set of buffers between Pin and compressor processes. Within Pin, the memory tracing tool and the instrumented SPMD application act as a producer while the compressor process acts as a consumer. A compressor process reads from the pipe to which the corresponding Pin process writes the memory references. 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 2.2.

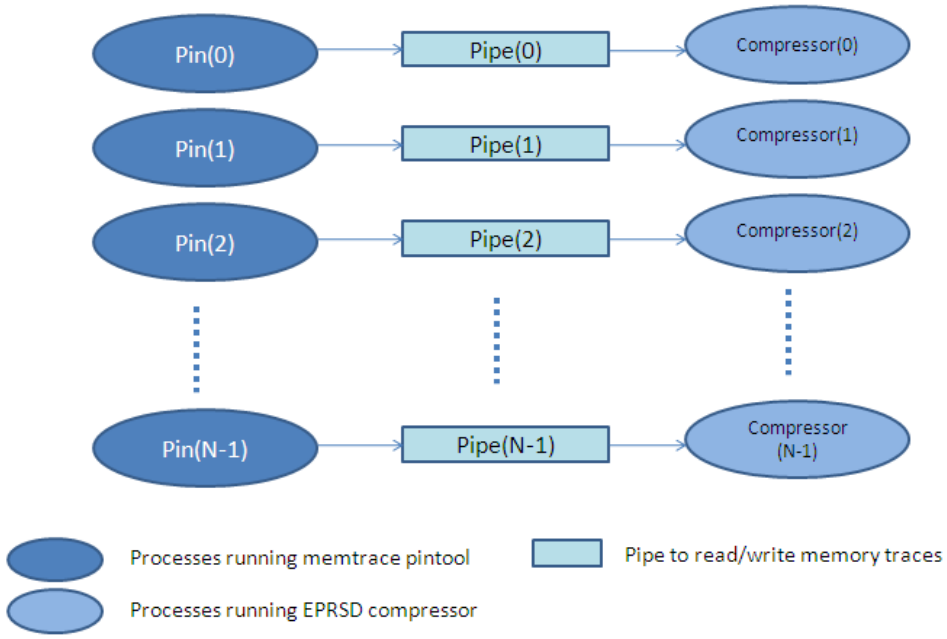


Figure 2.2: Design of the Memory Trace Compressor

The compressor module can be integrated with the Memtrace pintool to avoid copying to and from the pipe, but development and debugging is easier when these two modules are run as separate processes. Also, memory traces can be saved as files only once and the compressor can be run multiple times with different precision settings without the need to instrument every

time. This saves considerable amount of time during development. We utilize a networked file system in the OPT cluster for experiments and all pipes/files are visible across all machines.

Each compressor process generates the EPRSDs for all the threads in the respective application process and performs intra-thread and inter-thread merging before passing the EPRSDs to another compressor process for inter-node merging. The name inter-node merging or inter-node compression refers to the merging of repetitive memory access patterns (or EPRSDs) across multiple processes of a SPMD application. The MPI compressor processes involved in inter-node merging can be running on a single or different machines in a cluster. In some literature, these machines are also referred as nodes (not to be confused with the term inter-node used in this report). This results in order preserving, lossless and near-constant size memory traces, 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].

### 2.2.1 Intra-thread Compression

Intra-thread compression is achieved by exploiting the repetitive behavior of an application. Regular Section Descriptors (RSDs) [1] captures Load and Store instructions with in a loop in constant-size and Power Regular Section Descriptors (PRSDs) captures RSDs nested in multiple loops [1]. EPRSD is an extended version of a PRSD with thread dependences. For example, $RSD1 :< 1000, ST_B, LD_A >$ represents alternate 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 thread dependences ignored. The code snippet in Figure 2.3 corresponds to the PRSD mentioned above.

```
for(i = 0; i < 100; i++)
{
        for(j = 0; j < 1000; j++)
        {
                b = a; /* load from a's location and store to b's location */
        }
        d = c; /* load from c's location and store to d's location */
}
```

Figure 2.3: 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

**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 ≠ 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
```

traces generated by the instrumentation tool, new entries are appended to the list if no match is found, otherwise added to the matching window to find repetitive sequence. The compression algorithm involves finding repetitive patterns in the input memory trace and creating an RSD when a sequence of repetitions is found. To find the repetitive patterns, each memory reference is compared with a set of previous memory references. The extent to which this comparison is made depends on the size of the window used to buffer the memory references. The bigger the window size the higher is the compression achieved and vice versa. A large enough window size is needed to identify repetitions to achieve significant compression. To identify a loop of N memory references, a window size of at least 2N should be used to achieve compression. For M number of memory references, the algorithm runs in $O(M^2)$ time, if the window is not used and all previous references are compared. If the window size is S, then the algorithm runs in $O(MS)$ time.

An example of intra-thread compression is illustrated in Figure 2.4. The memory references op1, op2 and op3 are added to a list and matching patterns are found dynamically. As mentioned in Algorithm 1, on finding a matching sequence, the right portion is merged with the left portion and the count of RSDs is incremented. This process repeats for subsequent sequences of op1, op2 and op3 until the pattern disappears.

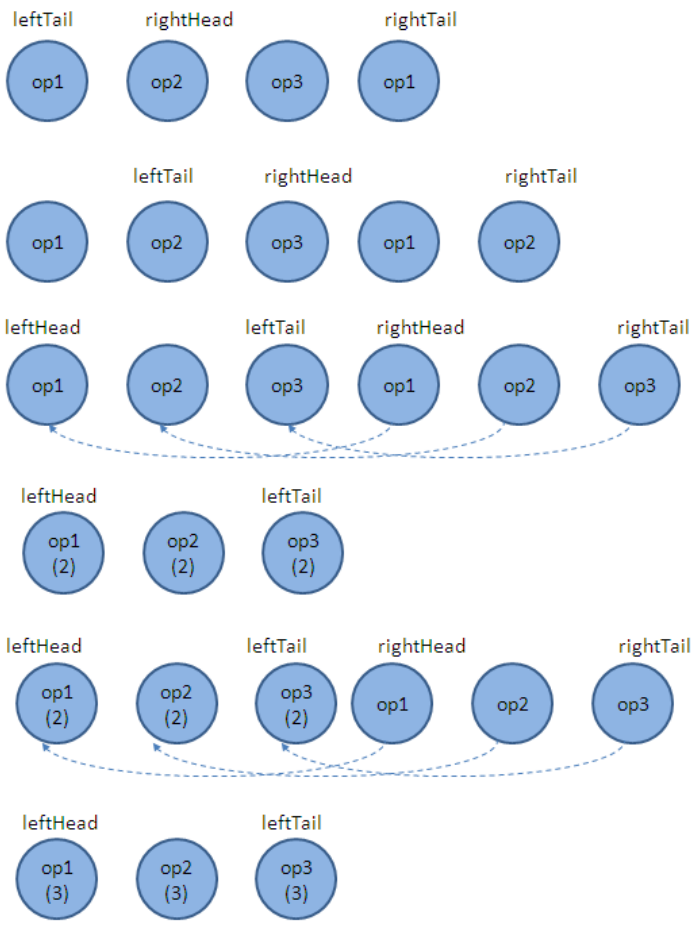


Figure 2.4: An Example of Intra-thread Compression

Intra-thread compression continues while the application executes. After the application completes execution, inter-thread memory trace compression commences. During trace generation, each instruction needs to be identified uniquely. Hence, a unique signature is computed for each instruction by performing a stack-walk. A series of return addresses and program counter values form the whole signature and their values are XORed to compute the XOR-signature. XOR-signature matching is a necessary (but not sufficient) condition for EPRSD merging. XOR-signatures are compared to speed up the matching process. Upon XOR-signature match, a whole 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. This signature is part of the memory trace fed to the compressor tool. An optimization using a signature tree is implemented to speed up the signature matching process, which is

explained in the later sections.

### 2.2.2 Inter-thread Compression

Intra-thread compression occurs on-the-fly 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-node compression is started. For a multi-threaded application, a separate compressor object maintains the RSDs and PRSDs of each thread. After all threads of an application complete execution, RSDs and PRSDs of individual threads are matched against each other and merged into an EPRSD when a match is found. The design of inter-thread compression is depicted in Figure 2.5. Pin instruments a process consisting of k threads and there are k compressor objects, one per thread, reading corresponding memory references from the pipe and performing intra-thread compression. After reading all memory references from the pipe, k compressor objects exchange EPRSDs, which marks the beginning of the inter-thread merging process.

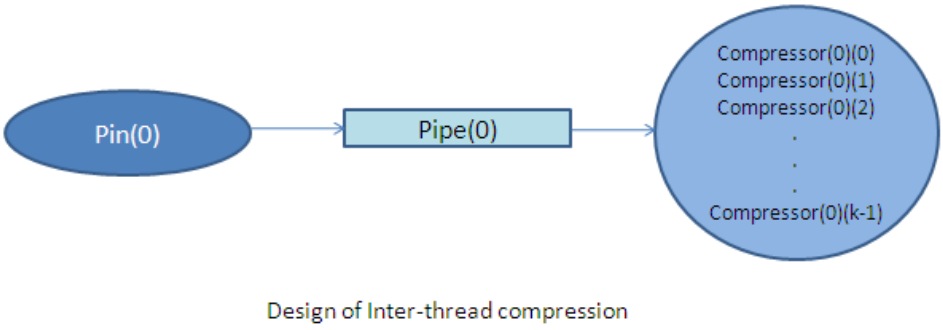


Figure 2.5: Design of Inter-thread Compression

PRSD lists are scanned for matching PRSDs with different thread-ids 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-id.

For 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 $0 \leq thread\_id \leq K - 1$. $(0, K, 1)$ suggests that the pattern is found in K threads starting at 0 with a stride of 1. The OpenMP code snippet in Figure 2.6 corresponds to the EPRSD mentioned above.

Inter-thread merging follows the binary radix tree approach to merge PRSDs from multiple

```
#pragma omp parallel
{
        tid = omp_get_thread_num();
        for(j = 400 * tid; j < 400 * tid + 100; j++)
        {
                /*  offset j depends on the thread identifier 'tid' */
                a[j] =  j;
        }
}
```

Figure 2.6: Inter-thread Compression: Sample Code for EPRSDs

threads into EPRSDs. This merging pattern is depicted in Figure 2.7 for four threads. As shown in the figure, the repetitive pattern of memory references are combined into a single entity and other copies are discarded. A thread ID's length is incremented and its stride is recomputed in the destination EPRSD at each stage of the merging process. The final EPRSD $< T : 0, 4, 1 >$ in the compressor object 0 indicates that the pattern occurred in four threads starting at 0 with a stride of 1. The same pattern applies to larger numbers of threads. In our experiments, the number of threads was configured to be a power of two.

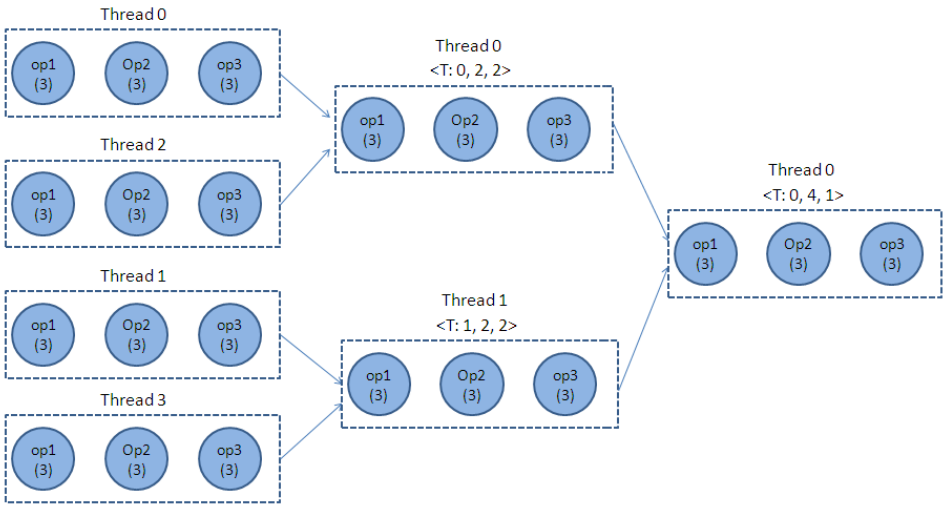


Figure 2.7: Design of Inter-thread Compression

When the inter-thread compression completes, compressor object 0 has the final list of EPRSDs merged from all the threads. From each compressor process, this single compressor object participates in the next level of inter-node compression.

### 2.2.3 Inter-node 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 counterparts in other processes. Each process has the final list of EPRSDs in compressor object 0, which are transmitted to another process using MPI calls. The EPRSD list is scanned for matching EPRSDs with different node-IDs but with the same signature. The binary radix tree approach is used to merge the inter-thread EPRSDs into inter-node EPRSDs. Hence, the inter-node 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 whole inter-node compression algorithm runs in $O(MlogN)$ time. The searching takes nearly constant time due to the SPMD nature of the applications and the worst case scenario of searching the whole EPRSD list seldom occurs.

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 $0 < thread\_id < T - 1$ and $0 < node\_id < N - 1$. $(0, T, 1)$ suggests that the pattern is found in T threads starting at 0 with a stride of 1 and $(0, N, 1)$ suggests that the pattern is found in N processes starting at 0 with a stride of 1. The MPI-OpenMP hybrid code snippet in Figure 2.8 corresponds to the EPRSD mentioned above.

```
MPI_Init(&argc, &argv);
#pragma omp parallel
{
        tid = omp_get_thread_num();
        for(j = 400 * tid; j < 400 * tid + 100; j++)
        {
                /*  offset j depends on the thread identifier 'tid' */
                a[j] =  j;
        }
}
MPI_Finalize(MPI_COMM_WORLD);
```

Figure 2.8: Inter-node Compression: Sample Code for EPRSDs

The radix tree approach of inter-node compression is depicted in Figure 2.9 for four processes. As shown in the figure, the repetitive pattern of memory references from different processes are combined into a single entity and other copies are discarded. A process ID's length is incremented and its stride is recomputed in the destination EPRSD at each stage of the merging process. 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 a stride of 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 a power of two.

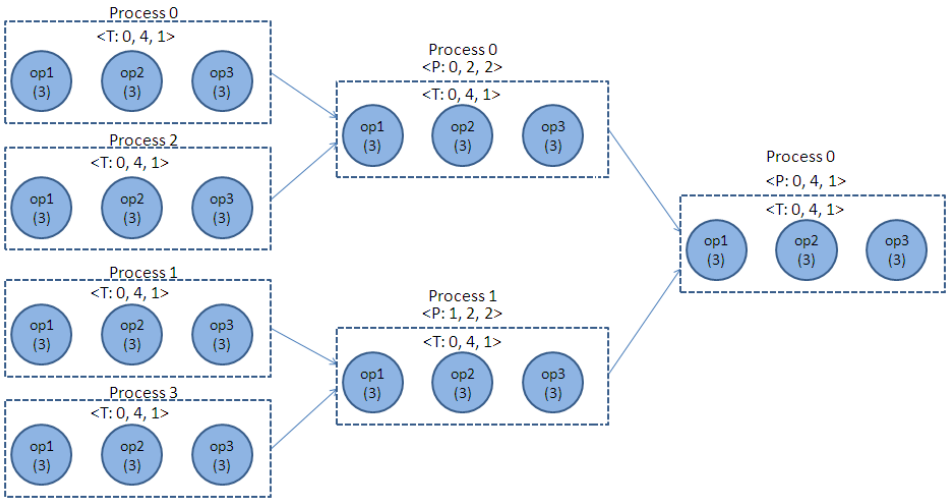


Figure 2.9: Design of Inter-node Compression

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

# Chapter 3

# Memory Trace Generation

## 3.1 Binary Instrumentation

Binary instrumentation involves inserting additional code at certain locations in a program. Static binary instrumentation inserts code at desired locations at compile time, whereas dynamic binary instrumentation inserts instrumentation code at runtime. Static binary instrumentation has additional overhead, even when instrumentation is turned off, as the instrumentation code exists with the program code at runtime. Dynamic instrumentation results in overhead only when instrumentation is active as the code is not inserted if instrumentation is turned off during execution. Pin [16] is a dynamic instrumentation tool that offers the flexibility to turn instrumentation on and off dynamically without incurring unnecessary overhead.

## 3.2 System Overview of Pin

Pin employs a just-in-time compiler (JIT) to instrument a binary at runtime. In Figure 3.1, the software architecture of Pin [16] is shown. Pin consists of a virtual machine (VM), a code cache and an instrumentation API used by pintools. Pin runs on top of the operating system and hence can instrument only user-level code. There are three binary programs while an instrumented program is executing - pintool, pin and the application. Our memory tracing tool, Memtrace, runs as a pintool. An MPI/OpenMP application is instrumented by Memtrace using Pin. Memtrace instruments only load and store instructions in the application. For each load and store instruction, an entry is added to EPRSD compressor object, which is discussed in detail in the following chapter. While the instrumentation continues, intra-thread compression occurs on-the-fly till the application terminates.

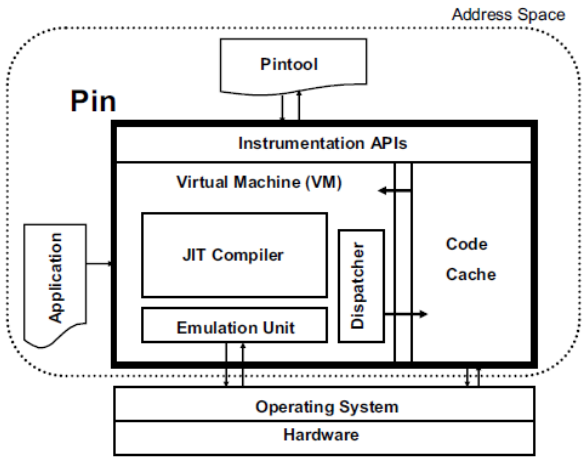


Figure 3.1: Software Architecture of Pin

## 3.3 Stack-walk

Stack-walk is performed to obtain unique stack signature for each memory reference and helps in matching the references during the compression phase. Stackwalker code is integrated with the Memtrace tool. There are two different stackwalkers available in the Memtrace tool: *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) Wisconsin Stackwalker* - freely available stackwalker library used to obtain stack signature. 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 the Wisconsin stackwalker correctly generates accurate stack signatures. The Wisconsin stackwalker library also helps to obtain consistent stack signature when shared library addresses vary across machines resulting in different signatures on different machines for the same application binary.

### 3.3.1 Stack Signature

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 incorporate the complete stack signature approach, which helps to preserve program structure. Consider the code snippet in Figure 3.2. Each memory reference is identified by a unique stack signature. Even though, the function bar() is executed twice, the stack signatures vary each time and

memory references are distinguished correctly. A naive comparison of program counter values would have incorrectly merged two different sequences of memory references A and B.

The stack signatures for memory references A and B in the trace are listed in Table 3.1. Stack signatures obtained during instrumentation are a series of return addresses ending with the instruction address referencing memory. EPRSDs are uniquely identified based on their stack signature.

```
bar()
{
        for(i = 1 to 3)
        {
                load A; //site 1
                store B; //site 2
        }
}

foo()
{
        call bar(); //site 4
}

main()
{
        call foo(); //site 5
        call bar(); //site 6
}
```

Figure 3.2: Sample Code to Demonstrate Stack Signatures

### 3.3.2 Problems in Unique Signature Generation

The Intel X86 ISA targets CISC architectures where multiple operations are performed by a single instruction. For example, an increment instruction performs both load and store in the same instruction along with the add operation. This results in identical signatures for two different memory operations. Such signatures result is false EPRSD matches and induce error in the compression process. Such instances were detected during instrumentation. To address this problem, their signatures were altered by XORing the instruction type with the Program-Counter value for uniqueness.

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, the corresponding PC values can be appended to the signature for all memory references within the function. Pin cannot always detect corresponding CALL and RET instructions accurately for instrumentation. We found a

Table 3.1: Unique Stack Signatures

| **Sequence No.** | **Operation** | **Address** | **Stack Signature** |
|---|---|---|---|
| 0 | LOAD | A | 5 4 1 |
| 1 | STORE | B | 5 4 2 |
| 2 | LOAD | A | 5 4 1 |
| 3 | STORE | B | 5 4 2 |
| 4 | LOAD | A | 5 4 1 |
| 5 | STORE | B | 5 4 2 |
| 6 | LOAD | A | 6 1 |
| 7 | STORE | B | 6 2 |
| 8 | LOAD | A | 6 1 |
| 9 | STORE | B | 6 2 |
| 10 | LOAD | A | 6 1 |
| 11 | STORE | B | 6 2 |

mismatch between the number of CALL and RET instructions instrumented by Pin. In such cases, the signatures only reflect the partial call path skipping over potential differences that remain undetected. 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 have incorporated this optimization in stack-walk (per-function). The performance speedup achieved is illustrated in the results section.

# Chapter 4

# EPRSD Template Library

We have developed a C++ template library to facilitate the rapid development of trace compression tools using EPRSDs 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-node compression. Most importantly, they are independent of any message-passing APIs. Users can incorporate this library in combination with any message-passing implementation. We have provided a sample MPI implementation of intra-thread, inter-thread and inter-node memory trace compression schemes. The source code is available for download at [27].

## 4.1 Design

The EPRSD library's main component is the template class EPRSD_COMPRESSOR, which is responsible for maintaining EPRSD lists and performing sequence matches of memory references. The EPRSD_COMPRESSOR template class takes three template arguments. The first is user-defined class to store trace information, the second is a static class implementing an EPRSD matching function and the third is a static class defining merge-notification callback. The EPRSD 'match' and 'merge' member functions declared as inline so that the C++ compiler can generate optimized code for better performance. Static classes are used so that the template library need not instantiate objects to invoke the 'match' and 'merge' member functions. A user-defined class must be derived from the base class INFO as some data members and member functions are internally used by the template library. The other supplementary classes are: (a) COMMON - used to fill generic EPRSD data like signature, XOR-signature, thread-id, node-id, etc.; (b)ITERATOR - used to iterate the EPRSD list of EPRSD_COMPRESSOR type objects. An example of using the EPRSD_COMPRESSOR class to compress traces is given in Figure 4.1.

```
/* add refID, address, threadid, nodeid etc. */
while(count)
{
          incr = count%5;
          /* user allocates memory for signature
          each time and library frees it in
          the end*/
          cmn = new COMMON(sign, signlen, 0x1234+incr,
          0x1001, 101, 10, "LD_ST");
          address = new MYINFO(0xABCD9874+incr);
          ecr1.add(cmn, address);
          count--;
}
/* display all EPRSDs in the compressor
object*/
ecr1.print();
```

Figure 4.1: EPRSD_COMPRESSOR Class Example

The EPRSD template library is designed such that, in a few lines of code, an intra-thread trace compressor can be built as depicted above. For an inter-thread and inter-node trace compressor, the user must iterate over the compressor object and exchange EPRSD data between processes to perform merging. The EPRSD template library provides classes EPRSD_COMPRESSOR::ITERATOR and EPRSD_DATA to iterate EPRSD lists and extract EPRSD data, respectively. The user is expected to incorporate these classes to implement inter-thread and inter-node trace compression, using the message-passing implementation of his choice. Isolating the template library implementation from the communication mechanism makes the library highly portable across various platforms and independent of platform-specific communication APIs.

## 4.2 Signature Trees

A signature tree is constructed as a separate C++ module and the classes are used by the EPRSD template library to incorporate signature tree functionality. The signature tree offers faster comparison of stack signatures than the XOR signature approach during trace 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 needed to ascertain the match. 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. If any two EPRSDs point to the same leaf node in a

signature tree, then they match; otherwise not. In either case, the comparison completes in constant time irrespective of the signature length.

During inter-node compression, EPRSDs are exchanged between processes. The complete signature list needs to be transferred for every EPRSD if XOR signatures are used. This adds significant communication overhead and increases the runtime of the compressor. If a signature tree is used, the signature list of every EPRSD need not be exchanged. The Signature tree needs to be transmitted once before the inter-node 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.

```
sigtree->initAdding();

while(!SignListisEmpty())
{
        val = getNextItemFromSignList();
        leaf = sigtree->add(val);
}
sigtree->finishAdding();
sigtree->print();
```

Figure 4.2: SIGTREE Class Example

The SIGTREE class is used to manage a signature tree. A SIGNODE class refers to a node within a signature tree. An example of using the SIGTREE class to build a signature tree is given in Figure 4.2. Each EPRSD_COMPRESSOR object contains its own SIGTREE object, which is built during the EPRSD_COMPRESSOR::add() operation. The SIGTREE::print() method prints the signature tree to stdout and its overloaded version prints the signature tree to a file. The SIGTREE::ITERATOR class has methods to iterate through a signature tree and extract individual nodes of type SIGNODE. These nodes are transmitted to other processes to reconstruct the signature tree before the inter-node merging begins.

When a signature tree is transmitted between processes and reconstructed again, the EPRSDs arriving thereafter have to be resolved to point to the newly formed leaf nodes. This is achieved by assigning unique keys to signature tree leaf nodes before transmission, which are stored in each EPRSD. When a new EPRSD arrives, the corresponding leaf node is found using this unique key, and a pointer to the leaf node is restored.

Consider the code snippet in Figure 4.3. The signature tree of two processes running this code is illustrated in Figure 4.4. The signature trees are constructed during intra-thread compression by the compressor object. The signature trees are exchanged during inter-thread

```
bar()
{
        for(i = 1 to 3)
        {
                load A; //site 1
                store B; //site 2
        }
}

foo()
{
        call bar(); //site 4
}

fun()
{
        for(j=1 to 5)
        {
                load C; //site 3
        }
}

main()
{
        call bar(); //site 5

        if(rank == 1)
                call foo(); //site 6
        else if(rank == 0)
                call fun(); //site 7
}
```

Figure 4.3: Sample Code to Demonstrate Stack Signatures

and inter-node compression before exchanging the EPRSDs. The signature tree in Figure 4.5 shows the signature tree after merging. In the example, process 1 is sending the signature tree to process 0 during inter-node compression.

## 4.3 Design Details

This section explains the design decisions taken during the development of the EPRSD template library and the signature tree library.

### 4.3.1 Why Templates?

Our objective is to devise a generic framework for the rapid development of EPRSD-based trace compressor tools. Each type of compressor has unique requirements but ad-hoc implementations are not reusable. For example, a memory trace compressor deals with memory addresses

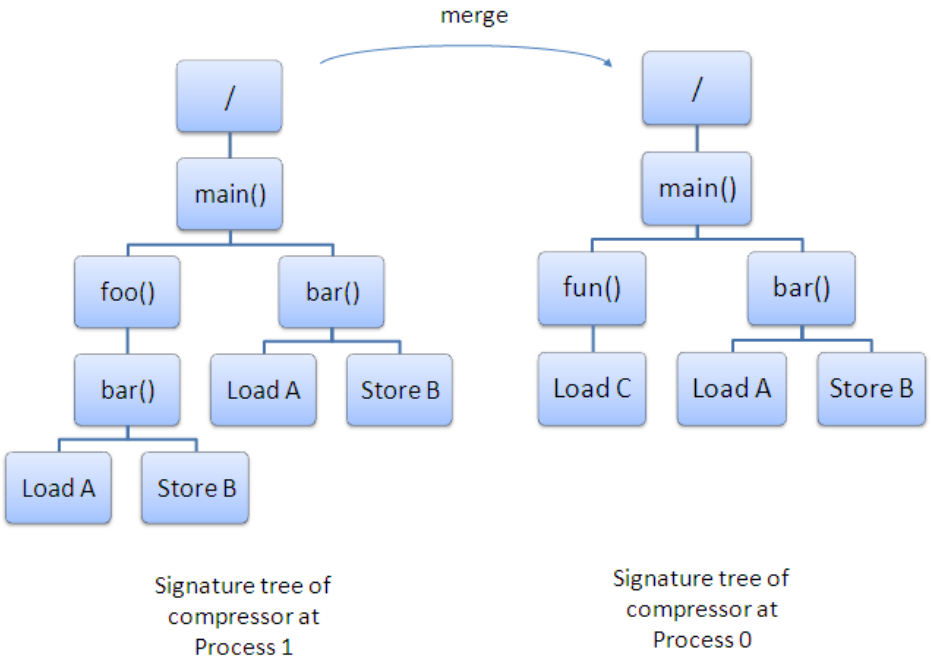


Figure 4.4: Signature Trees before Merging

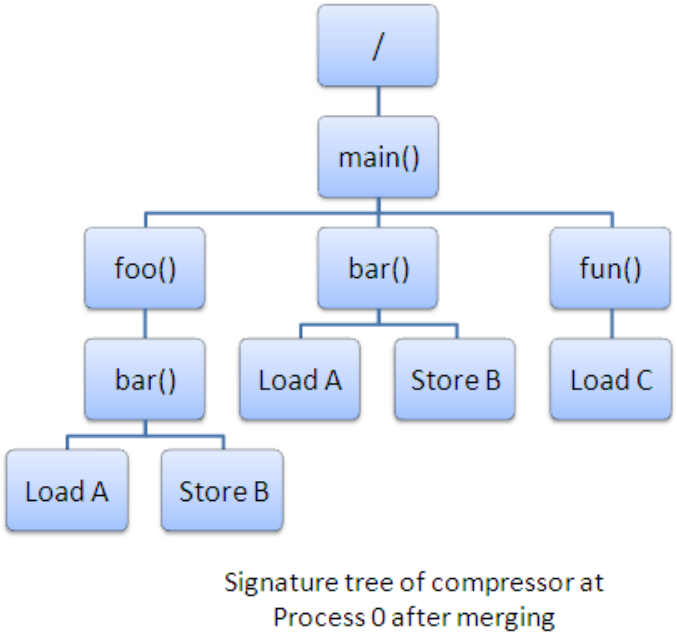


Figure 4.5: Signature Tree after Merging

```
template <class T, template <class M> class MATCH,
                        template <class G> class MERGE>
class EPRSD_COMPRESSOR
```

Figure 4.6: Template Class EPRSD_COMPRESSOR

whereas a communication trace compressor deals with message events. Each compressor tool needs the flexibility to define its own data-types but reuse the common EPRSD compression algorithm operating on these compressor-specific data. We decided to design a template library in C++ independent of any compressor-specific data-type. Such an implementation is reusable for various types of trace compressors. We have built our memory trace compressor using the EPRSD template library by instantiating template classes with user-defined data-types appropriate for memory trace compression.

### 4.3.2 Template Classes

#### (a) EPRSD_COMPRESSOR

The EPRSD_COMPRESSOR depicted in Figure 4.6 is the template class that implements methods to support trace compression. The parameters to this template class are: T - a user-defined class containing data members and methods needed by the user, MATCH - a template class in itself implementing a user-defined EPRSD comparing method that is inline and static, MERGE - a template class in itself implementing a user-defined callback function (inline and static) that is invoked each time EPRSDs are merged.

The compare and merge methods are not passed as function pointers while instantiating the EPRSD_COMPRESSOR class because the functions cannot be inlined by the compiler so that each time an actual function call is required, it would result in more overhead. Instead, the compare and merge methods are declared as inline functions so that the compiler can optimize the generated code. They are also declared as static so that they can be invoked without creating objects.

An example of instantiating the EPRSD_COMPRESSOR class is depicted in Figure 4.7. The user-defined class MYINFO is derived from the base class INFO, which contains common data members and methods required by the template library.

```
class MYINFO:public INFO
{
        public:
        MYINFO():INFO()
        {
        }
        ...
}

template <class T>
class MYMATCH
{
        public:
        inline static bool compare(T* src, T* dest)
        {
                ...
                return true;
        }
}

template <class T>
class MYMERGE
{
        public:
        inline static void mergecb(T* src, T* dest)
        {
                ...
                return;
        }
}

EPRSD_COMPRESSOR<MYINFO, MYMATCH, MYMERGE> ecr;
```

Figure 4.7: EPRSD_COMPRESSOR Class Instantiation

### (b) EPRSD

As the name suggests, the EPRSD class depicted in Figure 4.8 corresponds to an individual EPRSD. The EPRSD_COMPRESSOR class maintains a doubly linked list of EPRSD objects that is used to store and compare newly added events. This list is private to the EPRSD_COMPRESSOR class and not accessible to the user. Making the list private hides the implementation details from the user and prevents the user from accidentally corrupting the compressor state.

```
template <class T>
class EPRSD;
```

Figure 4.8: Template Class EPRSD

```
class EPRSD_COMPRESSOR
{
  template <class T, template <class M> class MATCH,
                        template <class G> class MERGE>
  friend class ITERATOR;
  ...
  ...
  class ITERATOR
  {
        ...
        ...
  };
};
```

Figure 4.9: Template Class ITERATOR

### (c) ITERATOR

The ITERATOR class depicted in Figure 4.9 is provided for the user to navigate the EPRSD list inside the compressor object and extract information in the form of EPRSD_DATA objects. Since access to the private linked list of the EPRSD_COMPRESSOR class is needed, ITERATOR is declared as a friend class inside the EPRSD_COMPRESSOR class.

### (d) EPRSD_DATA

The EPRSD_DATA class depicted in Figure 4.10 is used to extract the data portion of EPRSD objects, which are not directly accessible by the user. The EPRSD_DATA class is required to

```
template <class T>
class EPRSD_DATA;

class EPRSD
{
        ...
        template <class U> friend class EPRSD_DATA;
        ...
}
```

Figure 4.10: Template Class EPRSD_DATA

copy and transfer EPRSD data during inter-thread and inter-node compression. The EPRSD_COMPRESSOR class provides overloaded member functions to add events directly or as objects of EPRSD_DATA. The class EPRSD_DATA is declared as a friend class of the EPRSD class because access to EPRSD's private data is required while extracting the data present in an EPRSD type object. An example of using the EPRSD_DATA class is illustrated in Figure 4.11.

```
EPRSD_COMPRESSOR<MYINFO, MYMATCH, MYMERGE> ecr;
EPRSD_DATA<MYINFO> *data =  new EPRSD_DATA<MYINFO>(...);
ecr.interthread_add(data);
ecr.internode_add(data);
```

Figure 4.11: EPRSD_DATA Class Instantiation

### (e) COMMON and INFO

The trace data is divided into two parts. They are: (i) Common information - generic data, not application-specific and (ii) user information - application specific data.

This division of trace data prevents the user from having to manage generic data, the user only needs to manage application specific data. The COMMON class depicted in Figure 4.12 contains the generic data mandatory for the operation of EPRSDs, such as signature, thread details, process details, etc. The INFO class is an abstract class with one long integer data member required to store the initial value of an EPRSD, such as a memory address, MPI call type or File I/O call type. The INFO class is an abstract class and cannot be instantiated directly. The purpose of INFO class is to provide a base class to derive user-defined classes for storing application-specific information.

```
class COMMON;
class INFO;
```

Figure 4.12: Template Classes COMMON and INFO

```
class SIGTREE;
```

Figure 4.13: Class SIGTREE

```
class SIGTREE
{
        public:
        friend class SIGTREE_ITERATOR
        ...
        ...
};

class SIGTREE_ITERATOR
{
        ...
}
```

Figure 4.14: Class SIGTREE_ITERATOR

### (f) SIGTREE

A Signature tree module was developed to foster re-usability. Similar to EPRSD classes, signature tree classes can be used to build, traverse, merge and display signature trees. The SIGTREE class depicted in Figure 4.13 corresponds to a signature tree and provides methods to perform various operations on a signature tree.

### (g) SIGTREE_ITERATOR

The SIGTREE_ITERATOR class depicted in Figure 4.14 is provided to traverse the signature tree nodes used for the signature tree exchange during the inter-thread and the inter-node merging processes.

```
class SIGNODE;
```

Figure 4.15: Class SIGNODE

```
class SIGTREEITEM;
```

Figure 4.16: Class SIGTREEITEM

### (h) SIGNODE

The SIGNODE class depicted in Figure 4.15 corresponds to an individual node in signature tree, internal to the SIGTREE class and not directly accessible to the user.

### (i) SIGTREEITEM

The SIGTREEITEM class depicted in Figure 4.16 is provided to access the data portion of individual signature tree nodes. The methods in SIGTREE_ITERATOR class return objects of type SIGTREEITEM on traversing a signature tree.

## 4.3.3 Memory Management

Memory management is handled entirely by the EPRSD template library. The allocation and deallocation of memory is part of the template library and prevents the user from managing the memory required by the compressor. The memory management is also handled completely within the SIGTREE class and the users need to only instantiate objects of the SIGTREE class. This design provides transparency to the user and helps to prevent memory leaks caused due to mismanagement of user-allocated memory.

# Chapter 5

# Experimental Framework

We have used the hybrid MPI and OpenMP version of the Sequoia AMG benchmark, matrix multiplication and vector addition micro-benchmarks for the experiments. We have configured the number of threads and processes to be powers of two for micro-benchmarks. The AMG benchmark was executed with four OpenMP threads, the number of processes was varied from 1 to 16 keeping the problem size constant (strong scaling). Matrix multiplication and vector addition benchmarks were executed by varying the number of OpenMP threads from 4 to 32 and proportionally varying the problem size(weak scaling).

We ran all the experiments on the OPT cluster at NC State University, which has 16 nodes, each with two way SMP, dual core AMD Opteron 64 bit processors (x86_64).

# Chapter 6

# Results

## 6.1 Compression

We report the results of our experiments in this section. Files with EPRSD compressed traces also contain the loop and thread dependency information along with the address and reference type. This includes memory address information (number of references within a loop, number of iterations, 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). The compressed trace file sizes were measured by including all these details.

Table 6.1: Original vs. Compressed Trace Size of the Vector Addition Micro-benchmark for Problem Sizes Varying with the Number of Threads (Weak Scaling)

| **No. of OpenMP Threads** | **Original Trace Size (KB)** | **Compressed Trace Size (KB)** | **Problem Size** |
|---:|---:|---:|---:|
| 4 | 3,337 | 48.39 | 1024 |
| 8 | 6,662 | 48.42 | 2048 |
| 16 | 13,309 | 48.52 | 4096 |
| 32 | 26,607 | 48.96 | 8192 |
| 64 | 53,205 | 48.98 | 16384 |

Table 6.1 shows the size of the original trace files and EPRSD compressed trace files for the vector addition micro-benchmark. Each thread operates on partitions of two large 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

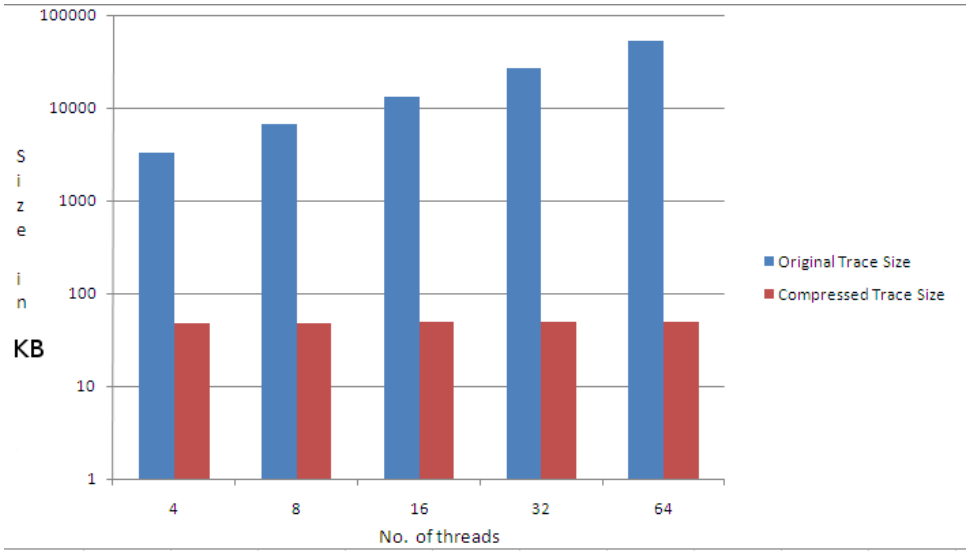


Figure 6.1: Weak Scaling-EPRSD Trace Size Comparison for Vector Addition

by increasing the array size proportionally with the number of threads. The figure shows the scalability of the EPRSD compression scheme for different concurrencies and problem sizes of the vector addition micro-benchmark. The compressed trace file size did not grow linearly, but remained nearly constant even when the problem size and number of threads were increased proportionally.

Table 6.2 shows the size of the original trace files and EPRSD compressed trace files for the matrix multiplication micro-benchmark under weak scaling. The figure illustrates the scalability of the EPRSD approach for different concurrencies and problem sizes of the matrix multiplication micro-benchmark. The compressed trace file size was not constant but 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 size increased 55 times, but the compressed trace file size increased only 3 times.

Table 6.3 shows the size of the original trace files and EPRSD compressed trace files for the AMG benchmark. Each MPI process involves four OpenMP threads. The figure demonstrates the scalability of the EPRSD compression scheme for different concurrencies of the AMG benchmark. Strong scaling is applied by keeping the problem size steady and varying only the number of MPI processes.

For the AMG benchmark, 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

Table 6.2: Original vs. Compressed Trace Size of the Matrix Multiplication Micro-benchmark with Problem Sizes Varying with the Number of Threads (Weak Scaling)

| No. of Threads | Original Trace Size(KB) | Compressed Trace Size(KB) | Problem Size |
|---|---|---|---|
| 4 | 4,739 | 236 | 10x10 |
| 8 | 12,307 | 475 | 14x14 |
| 16 | 34,421 | 375 | 20x20 |
| 32 | 91,918 | 613 | 28x28 |
| 64 | 262,518 | 631 | 40x40 |

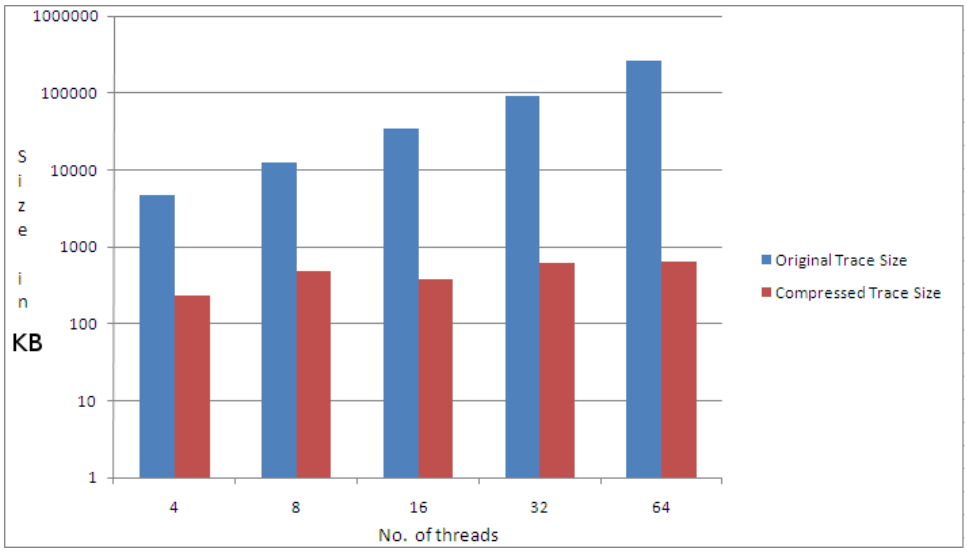


Figure 6.2: Weak Scaling - EPRSD Trace Size Comparison for Matrix Multiplication

size grew linearly with the number of MPI processes so that the size was reduced by half. Better compression can be achieved by detecting non-rectangular loops and merging them. Also user defined matching and merging capabilities can be included to override the default procedures to achieve better compression.

Table 6.4 shows the size of the original trace files and EPRSD compressed trace files for the Aztec benchmark. The figure demonstrates the scalability of the EPRSD compression scheme for different concurrencies of the Aztec benchmark. Weak scaling is applied by varying the problem size proportionally with the number of MPI processes.

The compressed trace file size grows linearly with the number of MPI processes so that the size is reduced by three times on average. The EPRSDs across multiple processes do not merge

Table 6.3: Original vs. Compressed Trace Size of the AMG Benchmark with a Fixed Problem Size and Varying Number of Processes (Strong Scaling)

| No. of MPI Processes | Original Trace Size (MB) | Compressed Trace Size (MB) |
|---|---|---|
| 1 | 170 | 68 |
| 2 | 340 | 136 |
| 4 | 680 | 272 |
| 8 | 1360 | 352 |
| 16 | 2720 | 1088 |

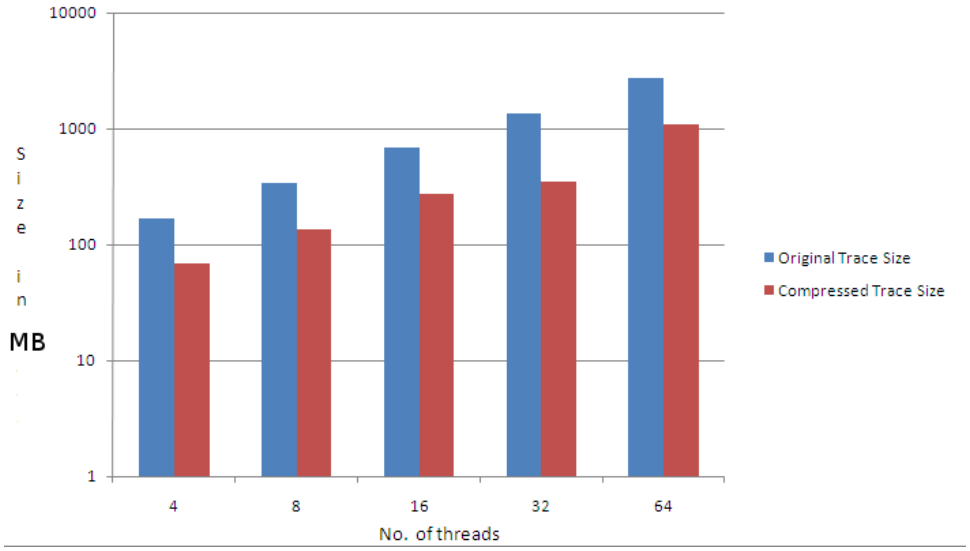


Figure 6.3: EPRSD Trace Size Comparison for AMG Benchmark

as they occur in different orders, which prevents further compression.

It should be noted that the scales are logarithmic. The raw trace file size increases exponentially with the number of threads. In contrast, the EPRSD trace file size remains almost constant in case of the vector addition microbenchmark and grows sub-linearly for the matrix multiplication microbenchmark. In case of the AMG benchmark, the compressed trace file size increases linearly with the number of processes but trace files were compressed by 50%. In case of the Aztec benchmark, the compressed trace file grows linearly with the number of MPI processes though the size is reduced by 65%. From the results, we can conclude that the space savings due to the EPRSD compression scheme is exponential and resulting traces are highly scalable in case of dense algebraic kernels (Matrix Multiplication and Vector Addition) and

Table 6.4: Original vs. Compressed Trace Size of the Aztec Benchmark with Problem Sizes Varying with the Number of Processes (Weak Scaling)

| No. of MPI Processes | Original Trace Size (KB) | Compressed Trace Size (KB) |
|---|---|---|
| 1 | 625 | 282 |
| 2 | 1327 | 438 |
| 4 | 2845 | 596 |
| 8 | 14944 | 6652 |
| 64 | 213824 | 72221 |

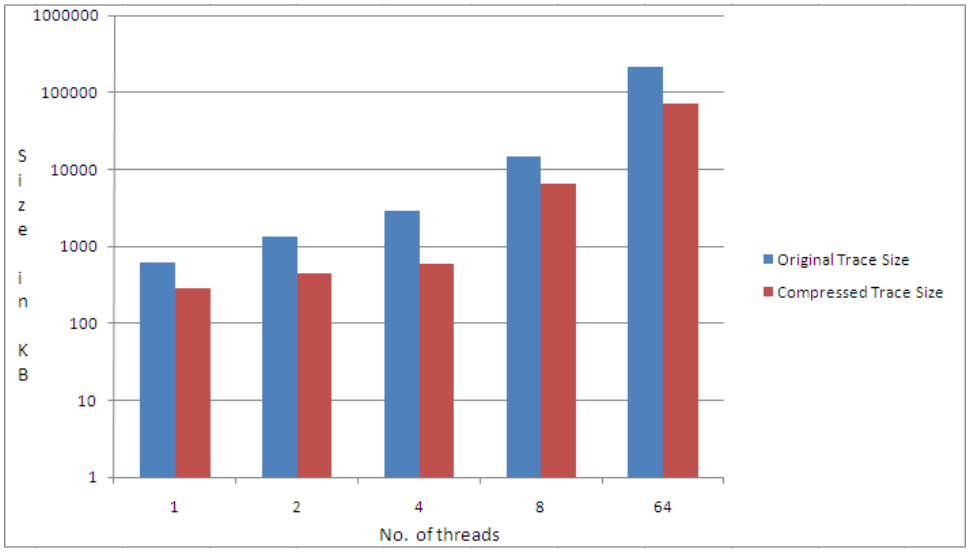


Figure 6.4: EPRSD Trace Size Comparison for Aztec Benchmark

linear in case of other benchmarks (AMG and Aztec).

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 and Aztec traces were replayed with 90% accuracy. In case of the AMG and Aztec benchmarks, the error is due to round-off errors caused by integer division in the compression algorithm.

## 6.2 Performance

In this section, we discuss the runtime performance of instrumentation, stack-walk and different levels of compression. "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 the AMG benchmark with two MPI processes and four OpenMP threads in each process.

Instrumenting an application incurs additional overhead than running a stand-alone executable. Even when instrumentation is disabled, application runtime increases when executed within Pin [16]. This overhead is due to the additional time required to execute Pin itself. The difference in application runtimes within Pin with instrumentation turned on and off is depicted in Table 6.5. This difference is due to the additional overhead involved in executing dynamically rewritten application code snippets. The difference in runtimes with regular and optimized stack-walk is presented in Figure 6.5. 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.

Table 6.5: Runtime of Benchmarks with and without Instrumentation

| Benchmark | runtime inside Pin w/o instrumentation (sec) | runtime w/ instrumentation and w/ stack-walk optimization (sec) | runtime w/ instrumentation and w/o stack-walk optimization (sec) |
|---|---|---|---|
| Matmul 24x24 | 0.897195 | 14.016049 | 17.181311 |
| Matmul 48x48 | 0.936515 | 25.797583 | 50.751666 |
| AMG n=1 | 3.849418 | 106.020933 | 137.740902 |
| AMG n=2 | 5.082595 | 69.658261 | 146.941715 |

Stack-walk is part of our Memtrace tool and contributes to the overall instrumentation time. The influence of optimized (per-function) stack-walk on the overall instrumentation time is depicted in Table 6.6. The figure shows that stack-walk (per-function) contributed only a minor portion of the overall instrumentation time while the major overhead is due to the

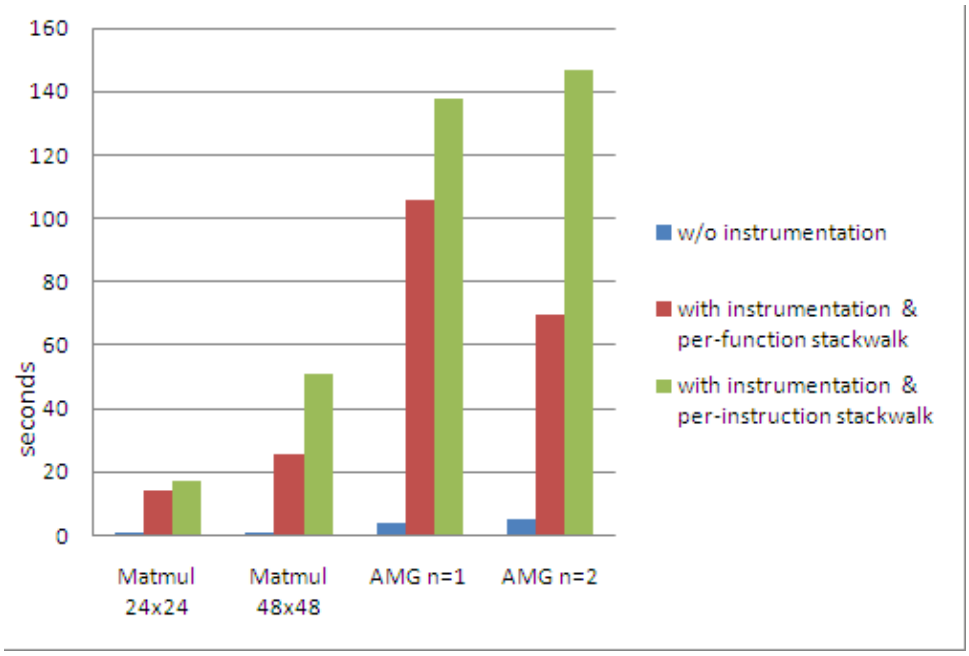


Figure 6.5: Instrumentation Overhead Comparison

instrumentation code and Pin overhead.

Table 6.6: Stack-walk and Instrumentation Runtime Comparison

| Benchmark | optimized stack-walk time (sec) | instrumentation time excluding stack-walk time (sec) |
|---|---|---|
| Matmul 24x24 | 0.153050 | 13.862999 |
| Matmul 48x48 | 0.182989 | 25.614594 |
| AMG n=1 | 7.280247 | 98.740686 |
| AMG n=2 | 5.112795 | 64.545466 |

Table 6.7 lists the runtime of various levels of compression for the given benchmarks. First three entries do not involve inter-node compression. Hence, only intra-thread and inter-thread compression runtimes are considered. From the corresponding Figure 6.7, we can derive that intra-thread compression time is almost equal to the instrumentation time listed in Table 6.5. This is because intra-thread compression occurs on-the-fly and completes soon after the instru-

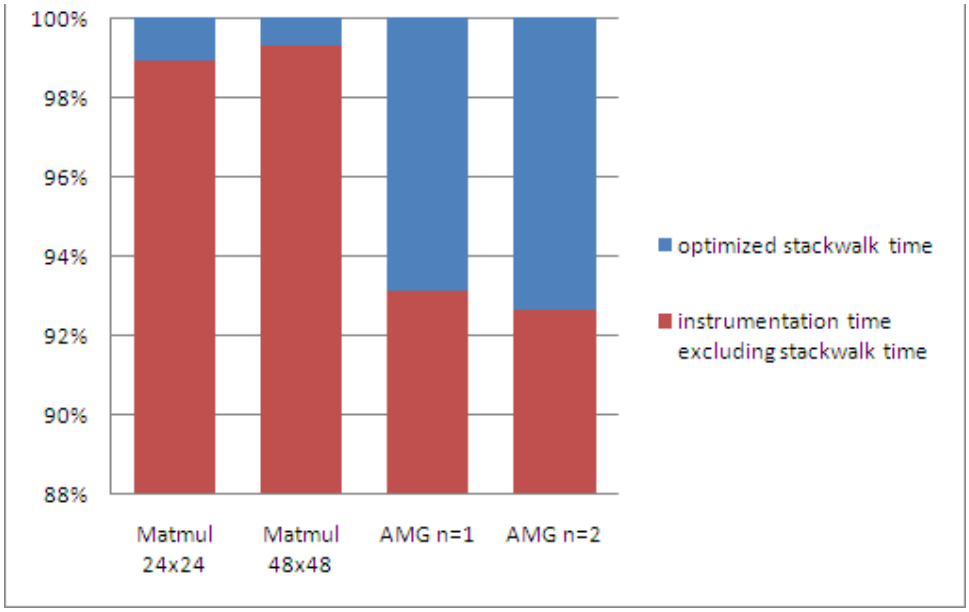


Figure 6.6: Stack-walk and Instrumentation Runtime Comparison

mentation terminates. Inter-thread compression runtime depends on the number of EPRSDs in each thread's compressor object after intra-thread compression. Its value varies widely depending on the benchmark and its runtime parameters. Inter-node compression involves 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. In the "AMG n=2" case, inter-node compression dominates the compression time due to MPI overhead and merging of large numbers of EPRSDs across processes.

Table 6.7: Compression Runtime Comparison

| **Benchmark** | **Intra-thread compression runtime (sec)** | **Inter-thread compression runtime (sec)** | **Intern-ode compression runtime (sec)** |
|---|---|---|---|
| Matmul 24x24 | 14.071198 | 0.001632 | 0.000000 |
| Matmul 48x48 | 25.854,444 | 0.008064 | 0.000000 |
| AMG n=1 | 112.623903 | 4.660802 | 0.000000 |
| AMG n=2 | 70.046435 | 109.836724 | 1,163.648747 |

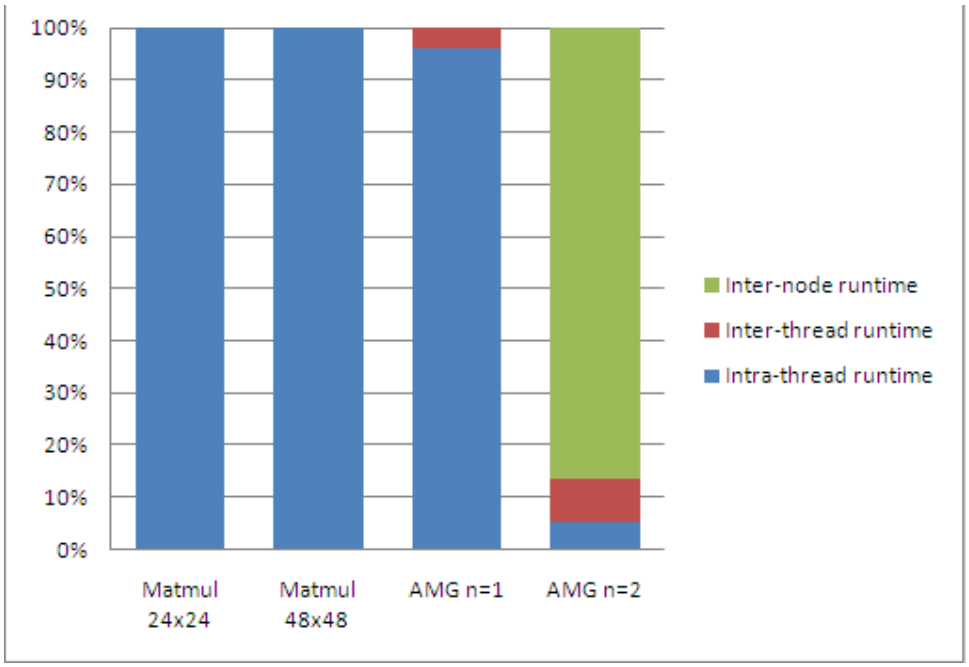


Figure 6.7: Compression Runtime Comparison

# Chapter 7

# Related Work

RSDs were first proposed in [8] to analyze array accesses. PRSDs were used to compress memory traces in [3] to analyze cache coherence problems in OpenMP programs. This work did not represent addresses as function of thread IDs and did not address inter-node memory trace compression. SIGMA [12] 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 dependences.

Caches As Filters [11] is an analytical framework for analyzing and designing caches. This work introduces TSpec notation to represent memory references in a compact format. The TSpec notation 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 [10], 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.

ScalaTrace [4] addresses intra-task and inter-node compression of communication traces, but not memory traces. Also, ScalaTrace [4] does not involve inter-thread compression. Trace compression discussed in [9] is based on statistical sampling and results in lossy compression and do not preserve order.

Our work addresses lossless, order preserving intra-thread, inter-thread and inter-node compression of memory traces. Imprecision is adjustable to achieve more compression by compromising the accuracy and order. With imprecision disabled, our tool compresses traces without loss of accuracy and order. We also offer an EPRSD template library developed in C++, for the rapid development of compression tools using EPRSDs 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, both the XOR signatures and signature tree options are available, which can be configured at compile time. Our tool also incorporates two different versions of Stackwalker libraries - a simple frame pointer traversal (Ver0) and the Wisconsin stackwalker [21], configurable at compile time.

Our tool can be used along with the existing cache performance analysis tools [3][11] to analyze the cache performance for multi-threaded applications. Our memory tracing tool can also be integrated with communication tracing tools [4] to combine online communication and memory trace compression of HPC applications.

# Chapter 8

# Conclusion

Memory traces of multi-threaded applications on SPMD machines are very large in size and do not easily aid in analyzing application behavior. The existing memory trace tools either produce large lossless trace files beyond disk capacities or produce lossy concise traces with only statistical details.

We developed a unique memory tracing framework that combines the advantages of both the above mentioned tracing tool types. Our tool extracts full memory traces and represents them in near-constant size regardless of the number of threads or problem sizes for dense algebraic kernels while preserving the memory access details along with the order in which memory accesses were issued. The developed scheme not only compresses loops but also groups similar memory access patterns across threads and processes into a single entity called Extended Power Regular Section Descriptor (EPRSD), which is an enhancement over the PRSD concept. We employ EPRSDs to compress memory traces of multi-threaded high-performance applications. Compression is performed at three levels: (a) Intra-thread, by using memory access patterns in loops within a thread; (b) Inter-thread, by using thread-ids to represent repetitive memory access patterns across multiple threads; (c) Inter-node, by using node-ids (or ranks) to represent repetitive memory access patterns across multiple processes. We also developed a replay mechanism to generate the memory traces from the compressed trace on-the-fly without ever decompressing the trace.

We observed that the compression achieved depends on the program structure of applications. Some benchmarks used in the experiment have rectangular loops and the order in which code is executed across multiple threads and processes is almost identical. In such cases, the compressed trace size has remained nearly constant. In some other benchmarks, there are non-rectangular loops and branches, hence, the order of code execution across different threads and processes is unique. In such cases, the compressed trace size has grown linearly with the

problem or concurrency size. The compression achieved is an indicator of a program's structure and its dynamic behavior. A near-constant size of compressed trace files indicates that a SPMD application's execution is highly synchronous. A poorly compressed trace file indicates the irregularity in a SPMD application's structure and execution.

As claimed in our hypothesis, we achieved near-constant size compression for dense algebraic kernels (Matrix Multiplication and Vector Addition). For other benchmarks (AMG and Aztec), compressed trace size grew linearly with the original trace size.

# Chapter 9

# Future Work

## Recognizing Triangular Loops

The current work does not handle detection and compression of triangular loops. It could be extended to recognize and compress triangular loops. This should help to obtain better intra-thread compression in the AMG benchmark, which utilizes non-rectangular loops.

## Lossy Compression by Noise Filtering

The current memory trace compression approach is lossless and order preserving. As observed in the experiments, the lossless approach does not work for all applications. If a subset of memory references only occurs within selected threads or processes, they might be treated as "noise" and could be ignored such that only matching references can be retained. This would result in better compression more suitable for trace analysis.

## User Pluggable Compression Schemes

The EPRSD template library is customizable and users could add their own compression scheme to override the default compression mechanism. This could be accomplished by defining the EPRSD "match" and "merge" functions, which are passed as template parameters to the EPRSD_COMPRESSOR class to suit the user-defined compression scheme. The current implementation of the compression algorithm is statically bound. The template library could be extended so that user-plug-ins in the form of shared libraries could be linked with the template library to customize the compression scheme.

## Integration with ScalaTrace

The ScalaTrace [4] library handles the compression of communication traces using an ad-hoc implementation. Our EPRSD template library could be incorporated in ScalaTrace to re-design the compression mechanism for better modularity and readability of code.

## REFERENCES


[1] P. Ratn, F. Mueller, Bronis R. de Supinski, Michael Noeth and M. Schulz, *ScalaTrace: Scalable Compression and Replay of Communication Traces for High Performance Computing* , Journal of Parallel and Distributed Computing, accepted Sep 2008, pages 1-14.

[2] Prasun Ratn, M.S. Thesis, *Preserving Time in Large-Scale Communication Traces* , North Carolina State University, Aug 2008.

[3] J. Marathe F. Mueller, T. Mohan, S. McKee, B. de Supinski, A. Yoo, *METRIC: Memory Tracing via Dynamic Binary Rewriting to Identify Cache Inefficiencies*, ACM Transactions on Programming Languages, Vol. 29, No. 2, Apr 2007, pages 1-36.

[4] M. Noeth and F. Mueller and M. Schulz and B. de Supinski *Scalable Compression and Replay of Communication Traces in Massively Parallel Environments*, P=ac2 Conference, IBM T.J. Watson, Oct 2006.

[5] 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*, International Symposium on Code Generation and Optimization, Mar 2003, pages 289-300.

[6] Heidi Pan and Krste Asanovic - Massachusetts Institute of Technology, Robert Cohn and Chi-Keung Luk - Intel Corporation *Controlling Program Execution through Binary Instrumentation*,

[7] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi and Kim Hazelwood *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation*, International Symposium on Code Generation and Optimization, Mar 2003, pages 289-300.

[8] Paul Havlak and Ken Kennedy. *An implementation of interprocedural bounded regular section analysis*, IEEE Transactions on Parallel and Distributed Systems, July 1991, 2(3):350360.

[9] Todd Gamblin, Rob Fowler, Daniel A. Reed. *Scalable Methods for Monitoring and Detecting Behavioral Equivalence Classes in Scientific Codes*, IPDPS 2008

[10] E.N. Elnozahy *Address trace compression through loop detection and reduction*, IBM Austin Research lab, 1999

[11] Dee A. B. Weikle, Kevin Skadron, Sally A. Mckee, William A. Wulf, University Of Virginia *Caches as Filters: A Unifying Model for Memory Hierarchy Analysis*, 2000

[12] DeRose, Luiz and Ekanadham, K. and Hollingsworth, Jeffrey K. and Sbaraglia, Simone *SIGMA: a simulator infrastructure to guide memory analysis*, Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, 2002

[13] John L. Hennessy, David A. Patterson *Computer Architecture: A Quantitative Approach, 4th Edition*

[14] David Culler, J.P. Singh, Anoop Gupta *Parallel Computer Architecture: A Hardware/-Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*

[15] A User's Guide to MPI. *Peter S. Pacheco, Department of Mathematics, University of San Francisco, 1995*

[16] Pin binary instrumentation tool.
*http://www.pintool.org/*

[17] MPICH2 - ANL/MSU MPI implementation.
*http://www.mcs.anl.gov/research/projects/mpich2/*

[18] MPI - The Message Passing Interface (MPI) standard.
*http://www.mcs.anl.gov/research/projects/mpi/*

[19] OpenMPI - A High Performance Message Passing Library.
*http://www.open-mpi.org/*

[20] OpenMP - The OpenMP API specification for parallel programming.
*http://openmp.org/wp/*

[21] Paradyn, Parallel Tools Project - Wisconsin Stackwalker library.
*http://www.paradyn.org/html/stackwalker1.1-features.html*

[22] Sequoia Benchmark Suite.
*https://asc.llnl.gov/sequoia/benchmarks/*

[23] The AZTEC Benchmark Code.
*https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/aztec/*

[24] Ranking of supercomputers according to the LINPACK benchmark as on June 2010.
*http://www.top500.org/*

[25] OPT cluster at NC State University.
*http://moss.csc.ncsu.edu/~mueller/cluster/opt/*

[26] Sony PS3 cluster at NC State University.
*http://moss.csc.ncsu.edu/~mueller/cluster/opt/*

[27] Memtrace tool, EPRSD Template library and EPRSD compressor source code.
*http://moss.csc.ncsu.edu/~mueller/ScalaTrace/ScalaMem-0.1.tgz*

# APPENDIX

# Appendix A

# Code Samples

## A.1 EPRSD Merging

### A.1.1 Intra-thread Merging Algorithm

Each compressor object is a list of RSDs and EPRSDs to which trace information is added as simple nodes to the tail end and checked for matching patterns. The number of nodes searched depends on the window size. The intra-thread compression algorithm is shown in Figure A.1.

### A.1.2 Inter-thread Merging Algorithm

In our implementation, Memtrace pintool is compiled with mpicc and spawned using mpirun as multiple processes. Intra-thread and Inter-thread trace compression occurs within a process and no inter-process messages are exchanged. Each thread has a separate EPRSD_COMPRESSOR object and these objects exchange the EPRSDs in a binary radix tree like fashion. The algorithm for inter-thread merging algorithm is given is Figure A.2.

### A.1.3 Inter-node Merging Algorithm

After the inter-thread compression, each process sends the EPRSDs to another process for merging. This communication pattern is designed such that the process with the lowest rank completes the final merging. This communication pattern is depicted in Figure A.3, where eight processes ($N = 8$) are involved in the inter-node memory trace compression. The direction of the arrows shows the direction of EPRSD transmission. A similar pattern is applicable for higher values of N. The inter-node merging steps are similar to inter-thread merging shown in Figure A.2, but uses MPI calls to transmit signature tree and EPRSD data between processes. EPRSD_COMPRESSOR class implements internode_XXXX methods to perform inter-node

```
EPRSD_COMPRESSOR::add(EPRSD neweprsd)
{
     addToTail(neweprsd);
     repeat = true;

     while(1)
     do
         repeat = false;
         match = NULL;
         rmatchbegin = rmatchend = lmatchbegin = lmatchend = NULL;
         match = findLastMatch();

        if(match)
        then
             rmatchbegin = rmatchend = listtail;
             lmatchbegin = lmatchend = match;

             while(lmatchend != rmatchbegin->prev)
             do /* search for matching sequence */
                   if(lmatchbegin->prev && rmatchbegin->prev)
                   then
                        if(isSignMatch(lmatchbegin->prev, rmatchbegin->prev))
                        then
                             lmatchbegin = lmatchbegin->prev;
                             rmatchbegin = rmatchbegin->prev;
                        else /* matching stops */
                             break;
                        endif
                   else /* beginning of list reached */
                             break;
                   endif
             end while

             if(lmatchbegin && lmatchend && rmatchbegin && rmatchend)
             then /* matching sequence found, merge */
                  if(lmatchend == rmatchbegin->prev)
                  then
                       if(mergeLeftRightPortions() == SUCCESS)
                            repeat = true;
                       else
                            repeat = false;
                       endif
                  end if
             endif

         else /* match not found */
               repeat = true;
         end if

         if(repeat == false || match == NULL)
              break;
         endif
     end while
     rmatchbegin = rmatchend = lmatchbegin = lmatchend = NULL;
     return;
}
```

Figure A.1: Sample Code for Intra-thread Compression

```
/* reduce the stride such that the first object in the array accumulates
   all the EPRSDs in the end */
for(stride = N_THREADS/2; stride > 0; stride = stride/2)
{
     /* merge the EPRSDs of threads with a given stride of thread id */
     for(dest = 1; dest <= stride; dest = dest + 1)
     {
           if(cmprsr[dest] == NULL || cmprsr[dest+stride] == NULL)
           {
                break;
           }
           /* number tree nodes uniquely before merging signature trees */
           /* merge (dest+stride) compressor object's sigtree and EPRSDs to
              (dest) compressor object */
           /* dest+stride is also the thread id */
           cmprsr[dest+stride]->ecr->numberSigTree(dest+stride, myrank);
           cmprsr[dest]->ecr->mergeSigTree(cmprsr[dest+stride]->ecr);
           /* copy unique keys of leaf nodes to the dependent EPRSDs */
           cmprsr[dest+stride]->ecr->assignUniqueKeys();
           /* iterate the EPRSD list of 'dest+stride' object and add them to
             'dest' object */
           iter = new EPRSD_COMPRESSOR<MYINFO, MYMATCH, MYMERGE>::ITERATOR
                                                 (cmprsr[dest+stride]->ecr);
           cmprsr[dest]->ecr->newEPRSDSetBegins();
           iter->begin();
           while(1)
           {
                cur_prsd = iter->getNext();
                if(!cur_prsd)
                     break;
                tmpdata.copy( (CODE)(0xFFFF & cur_prsd->code), cur_prsd->attr,
                cur_prsd->det, cur_prsd->det_len, cur_prsd->cmninfo,
                cur_prsd->userinfo);
                cmprsr[dest]->ecr->intertask_add(&tmpdata);
           }
           delete iter;
           iter = NULL;
     }
}
```

Figure A.2: Sample Code for Inter-thread Compression

compression.

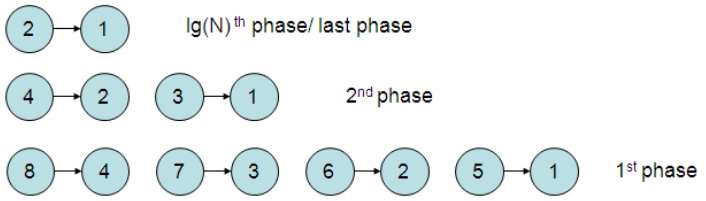


Figure A.3: EPRSD Exchange Pattern between Processes

## A.2 Signature Tree

Code sample to use the SIGTREE class is given below.

```
SIGTREE *sigtree = new SIGTREE();
SIGNODE *leaf = NULL;
void *val = NULL;
sigtree->initAdding();
/* assume signature list is 0x36364545 0x8090 0x7060 0x5040 0x1234 -
where 0x1234 is the PC and the rest are return addresses */
while(!SignListisEmpty())
{
        val = getNextItemFromSignList();
        /* leaf contains the pointer to a tree node
        at which val is added */
        leaf = sigtree->add(val);
}
sigtree->finishAdding();
/* after the loop terminates,   leaf   contains the pointer to leaf node
of the signature tree at which the above signature was added. Leaf node
contains the PC 0x1234 and the upper levels contain the return addresses */
sigtree->print(); /* prints the signature tree to stdout */
```

Figure A.4: Sample Usage of SIGTREE Class

Figure A.6 depicts the signature tree for the assembly code shown in Figure A.5. The signature tree is built using the signatures of load/store instructions only. The root of the signature tree is a dummy node used only for reference. The program initialization code is not shown for brevity.

```
0000000000400474 <bar>:
   400474:       55                      push   %rbp
   400475:       48 89 e5                mov    %rsp,%rbp
   400478:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
   40047f:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
   400486:       eb 0d                   jmp    400495 <bar+0x21>
   400488:       8b 45 fc                mov    -0x4(%rbp),%eax
   40048b:       89 05 5f 04 20 00       mov    %eax,0x20045f(%rip)
   400491:       83 45 fc 01             addl   $0x1,-0x4(%rbp)
   400495:       81 7d fc 9f 86 01 00    cmpl   $0x1869f,-0x4(%rbp)
   40049c:       7e ea                   jle    400488 <bar+0x14>
   40049e:       c9                      leaveq
   40049f:       c3                      retq

 00000000004004a0 <jam>:
   4004a0:       55                      push   %rbp
   4004a1:       48 89 e5                mov    %rsp,%rbp
   4004a4:       b8 00 00 00 00          mov    $0x0,%eax
   4004a9:       e8 c6 ff ff ff          callq  400474 <bar>
   4004ae:       c9                      leaveq
   4004af:       c3                      retq

 00000000004004b0 <foo>:
   4004b0:       55                      push   %rbp
   4004b1:       48 89 e5                mov    %rsp,%rbp
   4004b4:       b8 00 00 00 00          mov    $0x0,%eax
   4004b9:       e8 e2 ff ff ff          callq  4004a0 <jam>
   4004be:       c9                      leaveq
   4004bf:       c3                      retq

 00000000004004c0 <main>:
   4004c0:       55                      push   %rbp
   4004c1:       48 89 e5                mov    %rsp,%rbp
   4004c4:       b8 00 00 00 00          mov    $0x0,%eax
   4004c9:       e8 e2 ff ff ff          callq  4004b0 <foo>
   4004ce:       b8 00 00 00 00          mov    $0x0,%eax
   4004d3:       c9                      leaveq
   4004d4:       c3                      retq
   4004d5:       90                      nop
   4004d6:       90                      nop
   4004d7:       90                      nop
   4004d8:       90                      nop
   4004d9:       90                      nop
   4004da:       90                      nop
   4004db:       90                      nop
   4004dc:       90                      nop
   4004dd:       90                      nop
   4004de:       90                      nop
   4004df:       90                      nop
```

Figure A.5: Assembly Code Snippet

Figure A.6: Sample Signature Tree Generated by the SIGTREE Class

```
(nil)
     0x400395
     0x40039d
     0x40039e
     0x4003b4
     0x400380
     0x400386
     0x400370
     0x400303
     0x400376
     0x3e11e1eab0
             0x4004f0
             0x4004f5
             0x400508
             0x40050d
             0x400512
             0x400517
             0x400530
             0x400578
     0x40035c
     0x4003c0
     0x400535
             0x400361
                     0x4003d2
                     0x400361
             0x400366
                     0x400450
                     0x400458
                     0x400470
                     0x400471
                     0x400366
             0x40036b
                     0x400580
                     0x400584
                     0x400589
                     0x4005b3
                     0x4005b4
                     0x4005b5
             0x40036f
             0x400556
             0x40055b
             0x400560
             0x400565
             0x40056a
             0x40056f
     0x3e11e1eb1d
             0x4004c0
             0x4004c9
             0x4004ce
                     0x4004b0
                     0x4004b9
                     0x4004be
                             0x4004a0
                             0x4004a9
                             0x4004ae
                                     0x400474
                                     0x400478
```

```
                                0x40047f
                                0x400495
                                0x400488
                                0x40048b
                                0x400491
                                0x4004e2
                                0x40049e
                                0x40049f
                                0x4004ae
                        0x4004af
                        0x4004be
                0x4004bf
                0x4004d3
        0x4004d4
0x4005bc
0x4003e0
0x4003e4
0x4003e9
0x4003f7
0x400436
0x400441
0x400442
0x400443
0x3e11e1eb24
        0x3e11e35b82
                0x3e11a0e69c
                        0x4005c5
```