Memory Trace Compression, Replay and Extrapolation for SPMD Systems using Extended PRSDs

Sandeep Budanur Ramanna, North Carolina State University, sbudanu@ncsu.edu
Dr.Frank Mueller, North Carolina State University, mueller@cs.ncsu.edu

***Abstract*—Analyzing the memory traces of multithreaded 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. We propose an approach that provides near constant size memory traces irrespective of the number of threads involved while preserving the memory access details along with order in which memory accesses are done. The proposed scheme also groups the memory accesses with in a loop to a single entity called Extended Power Regular Section Descriptor (EPRSD) which is an enhancement over PRSD concept. We propose bi-level compression scheme based on memory access pattern and thread identifiers that are capable of extracting application's memory access structure. We further propose a replay mechanism for the traces generated by our approach and discuss results of our implementation on X86-64bit architecture. We propose an extrapolation mechanism as the next step which pinpoints the scalability issues. Considering all the above features makes EPRSD mechanism a novel approach for memory trace compression and replay.**

I. Introduction

SPMD (Single Process Multiple Data) systems have multiple processors executing multiple threads of a single process have complex memory access pattern and result in large amount of memory traces. The effective use of multiprocessors requires efficient memory access across threads. To analyze the memory access pattern of threads, tools are required. Most of the tools produce a large dump of memory trace which is difficult to handle. Such traces are not useful for analysis and are too large to scale well along with the problem size. Some tools provide only statistical information of memory accesses in order to reduce trace size but are lossy and hardly useful for scalability analysis.

We propose an innovative approach that produces lossless near-constant size memory traces that is highly scalable. This approach is based on the PRSD abstractions but more fine-grained and hence called Extended PRSDs(EPRSDs). EPRSDs preserve the order of memory references and generalize the memory access pattern across threads and loop dependencies. Our tool extracts complete memory traces and that are orders of magnitude smaller than the conventional memory traces, near-constant in size irrespective of the problem size. Figure-1 shows the overview of our EPRSD based memory trace compression tool.

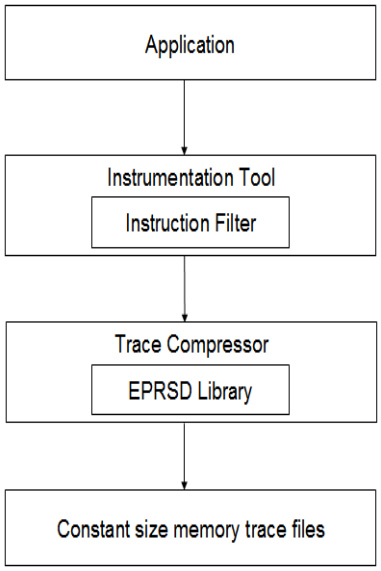

Figure 1. Block diagram of Memory Trace compressor tool

We rely on the binary instrumentation tool to generate memory traces of the application. This instrumentation tool has the logic to filter application specific memory traces. The output from the binary instrumentation tool is fed into the compressor module which runs parallely across several nodes. Each node generates the EPRSDs for a single thread and pass the EPRSDs to another node for merging based on thread identifiers and memory access pattern of the individual EPRSDs. 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.

The following content explains the individual components involved in the memory trace compression tool in detail.

II. Instrumentation

Our memory trace compression tool makes use of 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 traces dynamically. This trace consists of type of instruction, accessed memory address, instruction pointer,

stack frame pointer (base pointer) and 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 related to initialization of stack and registers are ignored as they are not application specific and not useful for scalability analysis. We further plan to incorporate the ability to filter instructions when dynamically linked libraries are involved. This trace is input to the compressor module which constructs EPRSDs to compress the traces.

III. Trace Compression

Memory traces are compressed based on two criteria. Fist one is intra-task compression based on starting address and stride within a thread. Second one is inter-task compression based on thread identifiers having similar memory access pattern. The latter is sub-divided into identifying multiple threads accessing same memory location and multiple threads accessing different memory locations but with a constant stride between them. PRSDs address the former scenario and EPRSDs address the latter.

We achieve intra-task compression by extending the concept of representing single loops using 'Regular Section Descriptors' (RSDs) to express load and store instructions nested in loops in constant-size. We represent an RSD as <start address, stride, length, type >, where length indicates the loop count, stride indicates the distance between the memory accesses of each iteration. We use Power RSDs (PRSDs) to represent recursive RSDs nested in multiple loops. PRSDs represent RSDs or PRSDs themselves as memory accesses to express nested loops as nested lists and sequence of loops as sequence of lists. For example, the tuple RSD1:<0x1234ABCD, 4, 1000, LD>represent 1000 load instructions starting with address 0x1234ABCD with a stride of 4 bytes and PRSD1:<0x11112222, 8, 100, ST, RSD1>represents 100 occurrences of store instructions starting with address 0x11112222 with a stride of 8 bytes followed by 100 occurrences of the loop denoted by RSD1. Inter-task compression is achieved if same patterns are found in multiple threads and thus thread-ids form a part of the PRSDs which assist in replaying of memory traces.

EPRSDs are built on the PRSD mechanism in which order of instructions is preserved and *similar* memory access patterns across multiple threads are captured as a function of thread-id. Thus an EPRSD consist of thread-id based offset to the base address along with the other entries of PRSD. EPRSD is represented as a 5-tuple against the 4-tuple representation of PRSD.

Below is a comparison of three sequential instructions whose range of addresses depend on the ids of the threads operating on them. In such a scenario the PRSDs and EPRSDs are represented as follows.

$C[i] = A[i] + B[i]$

$PRSD1 :< (base\_addr), (stride, length), LD_A >$

$PRSD2 :< (base\_addr), (stride, length), LD_B >$

$PRSD3 :< (base\_addr), (stride, length), ST_C >$

$EPRSD1 :< (base\_addr, thread\_id\_based\_offset), (stride, length), LD_A, LD_B, ST_C >$

Clearly the EPRSDs preserve the order of instructions and the base addresses are represented as a function of thread-id which is absent in PRSD. Also fewer EPRSDs are required than PRSDs to represent a given set of instructions in a loop. Still, EPRSD approach results in more computational time than PRSD approach.

A. *Intra-task Compression*

The compression algorithm involves finding the repetitive patterns in the input memory trace and creating an EPRSD when repetitions are 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. Bigger the window size and compression achieved is higher and vice versa. If window size is not large enough to identify repetitions, then compression achieved by EPRSDs is similar to that of PRSDs without preserving order. To identify a loop of N memory references, a window size of atleast 2N should be used to achieve maximum compression. After creating an EPRSD, it is compared with the existing set of EPRSDs. If a match is found existing EPRSD's length is updated otherwise new EPRSD entry is created. For compression to occur EPRSDs must belong to the same thread and have identical pattern. Intra-task compression continues till the application keeps running. After the application completes execution, inter-task memory trace compression begins.

Each instruction needs to be identified uniquely. So a unique signature is computed for each instruction by performing a stackwalk. A series of return addresses and program counter value is XORed to compute the unique signature. This signature should match while comparing EPRSDs for merging to take place. The logic for computing the signature is included in the instrumentation tool discussed earlier.

Intra-task compression is distributed across multiple processes and each process performs the intra-task compression of a single thread. If window size is 2N, then $\theta(N^2)$ comparisons are made in case of EPRSDs apart from table lookup and file I/O time. In case of PRSDs, only table lookup and file I/O time is involved as pattern matching is not involved. PRSD scenario can be considered as an EPRSD scenario with window size, $N = 1$. The algorithm used for intra-task compression is given as follows.

```
AddToWindow(WINDOW* window, TRACE* new_trace_node)
{
 FRESH_MATCH:
 if(l_match_begin == NULL)//no match found yet
 {
  match = window->getMatch(new_trace_node);
  if(match)//match found
  {
   if(l_match_begin == NULL)
```

```
  {
   l_match_begin = match;
  }
  l_match_end = match;

  //add to window and store the postition
  //as right window's starting point
  window->addTail(new_trace_node);
  if(r_match_begin == NULL)
  {
   r_match_begin = window->tail;
  }
   r_match_end = window->tail;
 }
 else
 {
   //add to window
   window->addTail(new_trace_node);
 }

}
else//some match was found earlier
{
 if(l_match_end->next != r_match_begin)
 {
  //matching continues
  if((l_match_end->next)->refId ==
      new_trace_node->refId)
  {
   l_match_end = l_match_end->next;
   window->addTail(new_trace_node);
   r_match_end = window->tail;
  }
  else //matching stops
  {
   //reset the begin and end points of
   //l and r windows
   l_match_begin = l_match_end = NULL;
   r_match_begin = r_match_end = NULL;
   //use the new trace node to match
   goto FRESH_MATCH;
  }

 }
 else//l_match_end is just before r_match_begin
 {
  //Pattern Matched
  //add to EPRSD table
  addWindowToEPRSDTable(window);
  //reset the begin and end points of
  //l and r windows
  l_match_begin = l_match_end = NULL;
  r_match_begin = r_match_end = NULL;
  window->head = window->tail = NULL;
  window->count = 0;
  goto FRESH_MATCH;
 }
}
 return;
}
```

B. *Inter-task compression*

After the intra-task compression is over, each process sends the EPRSDs to another process for merging. This communication pattern is designed such that the process with the highest rank completes the final merging. This communication pattern is depicted in Figure-2 when eight processes($N = 8$) are involved in the intra-task compression. The direction of the arrow shows the direction of EPRSD transmission. Similar pattern is applicable for higher values N. Each EPRSD list is scanned for matching EPRSDs of different threads with the same signature and dimension. If regular memory access patterns are found then base address for each EPRSD is represented as a function of thread-id. A binary radix tree approach is used to merge the individual EPRSDs and hence the merging of each EPRSD list takes $O(lgN)$ time where N is the number of threads in application (or N processes each handling intra-task compression). This process repeats for all the EPRSDs in the compressor list. If there are M EPRSDs (M unique signatures)the whole inter-task compression algorithm runs in $O(MlgN)$ time.

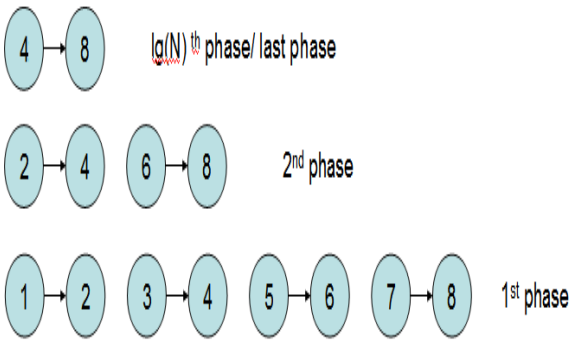

Figure 2. EPRSD exchange pattern between processes

This stage also involves determining the memory access pattern across different threads and representing the base address of each EPRSD as a function of thread-id. For example, $EPRSD1 : <(1000, 400), (100, 4), LD_A>$ denotes 100 occurrences of load A instruction with stride 4 and base address = (1000+400*thread_id) such that $0 \leq thread\_id \leq N - 1$. Additionally, this stage involves computing and storing the offset values for the local variables, storing the stack pointers of each thread in a table to assist during replay.

C. *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 using just a couple of objects. C++ Classes are designed for for

both intra and inter-task compression. Most importantly they are independent of any message passing APIs. Users can use this library in combination of any message passing API library. We have provided a sample MPI implementation of intra and inter task compression schemes. Code is available for download from *http : //www4.ncsu.edu/ sbudanu/CSC548project/*

D. Results

Table-I shows the size of the original trace files and EPRSD compressed trace files for a matrix multiplication program operating on 8x8 matrices with various number of threads. Table-II shows the size of the original trace files and EPRSD compressed trace files for a matrix multiplication program operating on 16x16 matrices with various number of threads. Figure 3 and Figure 4 show the scalability of EPRSD approach for different concurrencies and problem sizes. It should be noted that the sclaes are logarithmic. The raw trace file size increass exponentially with the no. of threads and the EPRSD trace file size remains constant for all thread and problem sizes. By looking at the results we can conclude that the space savings of the EPRSD compression scheme is exponential and resulting traces are highly scalable.

TABLE I
ORIGINAL VS COMPRESSED TRACE SIZE OF A 8X8 MATRIX MULTIPLICATION PROGRAM FOR VARIOUS NO. OF THREADS

No. of Threads	Original Trace Size (Bytes)	Compressed Trace Size (Bytes)
1	1170160	39691
4	4664566	39691
8	9323788	39692
32	37279480	39744

TABLE II
ORIGINAL VS COMPRESSED TRACE SIZE OF A 16X16 MATRIX MULTIPLICATION PROGRAM FOR VARIOUS NO. OF THREADS

No. of Threads	Original Trace Size (Bytes)	Compressed Trace Size (Bytes)
1	8972536	39766
4	35874088	39766
8	71742797	39767
32	286955558	39819

EPRSD compressed files also contain the loop and thread dependency information. The number of loop nesting levels, loop size, loop count, memory address length/stride, thread id length/stride and node id length/stride

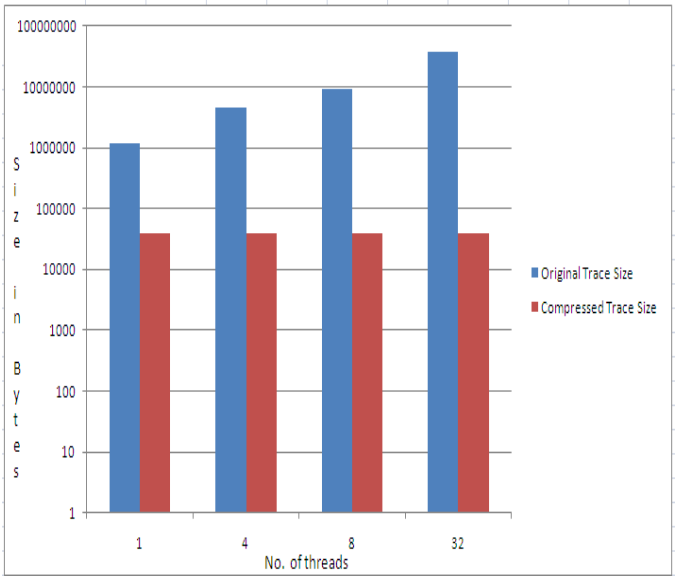

Figure 3. EPRSD trace size comparison for 8x8 matrix multiplication

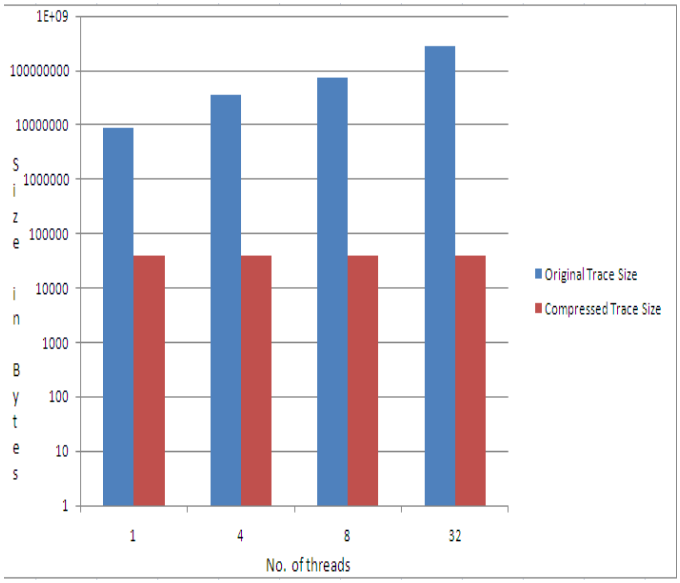

Figure 4. EPRSD trace size comparison for 16x16 matrix multiplication

IV. Replay of memory traces

We propose to replay the memory traces to execute load and store instructions in the same order as their structure and order is preserved in the compressed trace. Replay happens at run time without the need to decompress the traces. Thus the application's memory access behaviour can be assessed without actually running the application. The correctness of our compression scheme can be verified by using the replay engine.

V. Extrapolation for scaling

The compressed memory traces can be analyzed for task scaling (strong scaling), problem scaling and weak scaling (task and problem size scaled proportionally). The scal-

ability problems can be detected easily as they form the inefficient part of the compressed trace.

We propose to extrapolate larger memory traces using the smaller ones which can be used to (a) replay the larger memory traces to detect performance bottlenecks empirically. (b) to determine the constraints on scalability analytically.

VI. Conclusion

Memory traces of multithreaded applications on SPMD machines are very large in size and hard to analyze application behaviour. The existing memory trace tools either produce lossless and large trace files which are too big to store on disk or produce lossy concise traces with only statistical details.

We present 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 tasks or problem size while preserving the memory access details along with order in which memory accesses are done. The proposed scheme also groups the memory accesses with in a loop to a single entity called Extended Power Regular Section Descriptor (EPRSD) which is an enhancement over PRSD concept. We employ extended power regular section descriptors (EPRSDs) to achieve compression. Compression is performed at two levels,(a) Intra-task: using memory address access pattern within a task. (b) Inter-task: using thread ids to represent regular memory access patterns across multiple threads.

We also propose a replay mechanism to generate the memory traces from the compressed trace on the fly without decompressing the trace completely. We present the extrapolation scheme to analyze scalability problems by extrapolating larger memory traces from smaller ones and replaying them.

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