Compactly Representing Parallel Program Executions

Ankit Goel   Abhik Roychoudhury   Tulika Mitra

School of Computing
National University of Singapore
Republic of Singapore 117543
[ankitgoe,abhik,tulika]@comp.nus.edu.sg

ABSTRACT

Collecting a program's execution profile is important for many reasons: code optimization, memory layout, program debugging and program comprehension. Path based execution profiles are more detailed than count based execution profiles, since they present the *order* of execution of the various blocks in a program: modules, procedures, basic blocks etc. Recently, online string compression techniques have been employed for collecting compact representations of sequential program executions. In this paper, we show how a similar approach can be taken for shared memory parallel programs. Our compaction scheme yields one to two orders of magnitude compression compared to the uncompressed parallel program trace on some of the SPLASH benchmarks. Our compressed execution traces contain detailed information about synchronization and control/data flow which can be exploited for post-mortem analysis. In particular, information in our compact execution traces are useful for accurate data race detection (detecting unsynchronized shared variable accesses that occurred in the execution).

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Algorithms, Measurement

Keywords

Path profiling, Program path compression, Dynamic program analysis.

1. INTRODUCTION

Profiling a program and analyzing program profiles have traditionally been an important area of research. Profiles are useful for guiding code transformations, code layout, data layout and many other purposes. Indeed the performance of selected benchmark programs in architectural simulations is often used in developing novel processor and memory architectures. Traditionally the program profiles have been *aggregate* in nature, denoting the total count of basic block executions, cache misses etc. Path profiles of programs are considered more useful simply because they capture more information: the entire execution trace of a program for a particular input. However, path profiles are also more expensive to collect and store. Recently, Larus developed the notion of a *Whole Program Path* (WPP) [11] which captures the entire control flow trace of a program's execution. The storage overhead for this trace is reduced drastically by employing on-line string compression techniques [14]. Subsequently, [4] has studied the compression of dynamic data access pattern of a program. [24] studies the break-up of a program's control flow trace into per-function sub-traces, and stores these sub-traces compactly.

A natural question that arises from this line of work is whether such path profiling techniques extend to shared memory parallel programs. Developing execution profiles for parallel programs (containing the synchronization and control/data flow in an execution trace) is also crucial for various reasons. It allows the programmer to study the shared variable data access patterns across threads. This may be important for deciding on the memory layout in each thread, detection of useless cache misses due to inter-thread artifactual communication etc. It may also help in choosing a multiprocessor architecture for running a program. In particular, the choice of the granularity of coherence and the cache coherence protocol are important for multiprocessor memory system performance. Intelligent choices can be made for both (i.e., the coherence granularity and the coherence protocol) by studying programs' execution traces.

Furthermore, the complete execution trace of a parallel program can reveal bugs which are unique to shared memory programs: the presence of data races. A data race is an unsynchronized shared variable access in a program. A program with data races can produce different results depending on the relative speed of the processors and is often considered as unintended by the programmer. By inspecting the synchronizations and data/control flow information in an execution trace, it is possible to accurately identify data races manifested in the trace.

In this paper, we develop techniques to extract compressed execution profiles for shared memory parallel programs. For multithreaded programs running on a single processor, one can easily adapt the notion of Whole Program Paths (WPP) by changing the alphabet of the string denoting the execution path. The string will then store information about entry and exit from a thread, which appear at the points where a new thread is scheduled. However, for parallel programs, we need to separately maintain the execution trace on each processor. Across processors we need to maintain information about synchronization (which is achieved through primitives such as locks, unlocks and barriers). In particular, we can represent the control flow trace in each thread as a WPP, and the shared memory locations accessed by each load/store instruction can also be compressed on-the-fly. The resultant representation achieves a high degree of compression since the memory addresses accessed by a particular instruction often form a progression (e.g., if the instruction sweeps the elements of a unidimensional or multidimensional array). The unified representation of control and data flow in a trace is useful for guiding software prefetching or finding conflicting instruction pairs in different threads which cause data races.

Contributions of this Paper. The contributions of this paper can be summarized as follows.

- We study the feasibility of collecting and storing compressed execution profiles for shared memory parallel programs. To the best of our knowledge, little work has been done on collecting compact representations of parallel program executions.
- We study a unified representation for storing the control and data flow in an execution trace (see CDR representation in Section 2). This can be useful for storing sequential program traces as well. We show that our unified representation achieves one to two orders of magnitude compression on some of the SPLASH benchmarks. The compression technique outperforms general purpose compression schemes such as *gzip* [25].
- The execution trace of a parallel program can be analyzed post-mortem to find data races. Unfortunately, some of these data races may be artifacts. To detect data races which are artifacts, we need detailed data/control flow information in the execution traces, so that we can compute the data and control dependencies. We can afford to capture much of this required information in our profiles because of its compressed nature.

Section Organization. The rest of this paper is organized as follows. Section 2 presents our compressed representation of parallel program execution profiles. Section 3 presents experimental results on compression efficiency and compression time using the SPLASH benchmarks. Section 4 describes the application of our profiles, in particular for post-mortem data race detection. The next section describes related work. Conclusions appear in Section 6.

2. COMPRESSED REPRESENTATIONS

In this section, we present compressed representations of parallel program executions. Our starting point is the *Whole Program Path* (WPP) representation developed by Larus [11]. As mentioned before, WPP is a compressed representation for sequential program executions. It is based on the SEQUITUR algorithm [14] which represents a finite string σ (the execution itself) as a context free grammar whose language is the singleton set $\{\sigma\}$. The execution path of a sequential program can be viewed as a string (over an alphabet of basic block executions, say) from which the grammar is synthesized *on-the-fly*. The time complexity of the synthesis algorithm is linear in the length of the input string. The grammar generated is represented as a Directed Acyclic Graph, called a *Whole Program Path* (WPP).

SEQUITUR Overview. The SEQUITUR algorithm reads symbols one-by-one from the input string and restructures the rules of the grammar to maintain the following invariants:

- no pair of adjacent symbols appear more than once in the grammar
- every rule (except the rule defining the start symbol) is used more than once.

To intuitively understand the algorithm, we briefly describe how it works on the string *abcabc*. After reading the first four symbols, the grammar consists of the single production rule $S \rightarrow abca$ (where S is the start symbol). On reading the fifth symbol, it becomes $S \rightarrow abcab$. Since the adjacent symbols ab appear twice in this rule (violating the first invariant), SEQUITUR introduces a non-terminal to get

$$S \rightarrow AcA \qquad A \rightarrow ab$$

Note that here the rule defining non-terminal A is used twice. Finally, on reading the last symbol of the string *abcabc* the above grammar becomes

$$S \rightarrow AcAc \qquad A \rightarrow ab$$

This grammar needs to be restructured since the symbols Ac appear twice. SEQUITUR introduces another non-terminal to solve the problem

$$\begin{array}{l} S \rightarrow BB \\ B \rightarrow Ac \\ A \rightarrow ab \end{array}$$

However, now the rule defining non-terminal A is used only once. So, this rule is eliminated to produce the final result. Note that the grammar accepts only the string *abcabc*.

$$S \rightarrow BB \qquad B \rightarrow abc$$

Synchronization Primitives. We seek to develop similar compressed representations for parallel program executions. Consider a parallel program $P_1 \parallel \ldots \parallel P_n$ where each thread P_i is running on a different processor. We assume a shared memory model where the different threads communicate via shared variables, and synchronize using lock, unlock and barrier operations. The lock and unlock operations have a well-understood semantics. They are of the form $\langle\texttt{lock l}_\texttt{j}\rangle$ and $\langle\texttt{unlock l}_\texttt{j}\rangle$ where l_j is a lock variable. In general, there can be several lock variables used in a single parallel program. A barrier operation is of the form $\langle\texttt{barrier b}_\texttt{j}\rangle$ where b_j is the barrier variable. Any thread P_i blocks at $\langle\texttt{barrier b}_\texttt{j}\rangle$ until all other threads reach their respective $\langle\texttt{barrier b}_\texttt{j}\rangle$ statements. Thus a barrier operation serves as a handshake among *all* the n threads. Lock, unlock and barrier form a popular set of synchronization primitives used commonly by programmers.

Any representation of a parallel program execution needs to consider: (a) control flow in each thread (b) shared data access pattern in threads (c) synchronization pattern across threads. Before we describe our compact representations, we first establish the relations among control flow, shared data access and synchronization pattern across threads.

2.1 Synchronization Pattern

The synchronization pattern across threads for a parallel program execution can be visualized as a Message Sequence Chart [3, 23]. A Message Sequence Chart captures a scenario in system execution by depicting inter-process interactions. Each process is represented as a vertical line. Each vertical line is annotated with a sequence of events which are assumed to be in total order (*i.e.*, the event occurring at the top of process p "happens-before" the event occurring at the bottom of process p's vertical line). The possible events for any process p are (a) $p!q, m$: p sends a message m to process q, (b) $p?q, m$: p receives a message m from process q, and (c) p, a: p executes internal computation a. The diagram depicts a *partial order* among all the events which is obtained from the transitive closure of the following ordering relations: (1) the send of any message "happens-before" its corresponding receive, and (2) the events in any process p are in *total order*.

To represent the execution of a parallel program $P_1 \parallel \ldots \parallel P_n$ we depict each P_i as a vertical line. Furthermore, we represent an additional process M denoting a single shared memory containing *all* the lock variables (in general there can be many locks in a realistic program, each guarding different structures/objects). Each unlock operation in thread P_i over lock variable l is shown as a send event $P_i!M, l$. Similarly, a lock operation in thread P_i over lock variable l is a receive event $P_i?M, l$.[1] In other words, send/receive events are used to depict hand-over of locks. Barriers are handled using send-receive edges in a similar fashion. All other computations, including read/write of shared variables are considered as internal events.

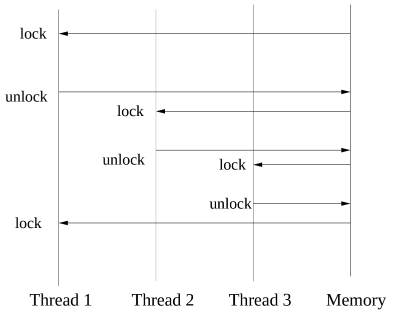


Figure 1: Synchronization Pattern

For example consider the program execution shown in Figure 1 where all the lock and unlock operations are on a single lock variable. In this example, the total order of synchronization operations is:

$(lock, 1), (unlock, 1), (lock, 2), (unlock, 2),$
$(lock, 3), (unlock, 3), (lock, 1), \ldots$

In our representation, the only communication across different threads that is captured is via the synchronization primitives. Communication via shared variable accesses is not explicitly represented, unlike the "interaction diagrams" of [2]. In other words, shared variable data dependences (a `load` operations reads the value of which `store` operation) are not explicitly represented. These dependences can be inferred from our representation if the parallel program concerned is "properly synchronized" (all shared variable accesses are protected by locks).

Given the above depiction of a parallel program execution, we can develop some simple observations on how the execution can be represented in a compact manner.

A. All the synchronization events form a total order and can be stored as a sequence S_{synch}.

B. The control flow/shared data access in each thread forms a total order and can be individually compressed.

C. Any synchronization event occurring in a thread P_i appears in the compressed representation of P_i as well as in S_{synch}. Thus, these two occurrences (of the same event) must be associated.

Observation (A) follows from the fact that we assume a single shared memory containing all synchronization variables. We assume that accesses to the synchronization variables are serialized. Observation (B) suggests the use of compression algorithms like SEQUITUR (which have proven effective for compressing sequential program executions) on control flow/shared data access of each thread. Observation (C) indicates that there should be some association between S_{synch} and the compressed control flow/shared data access per thread. We now explain how this is achieved using synchronization counts.

2.2 Synchronization Counts

To represent the execution trace of a parallel program, it is not sufficient to only represent the control/data flow in each thread and the global synchronization pattern. This does not allow us to retrieve the synchronization segments (the execution between a lock and its corresponding unlock) in a thread P_i without decompressing or naively traversing the trace of P_i. To solve this problem, we need to annotate the compacted traces of the individual threads.

Note that a WPP is a hierarchical representation of a string, which can be reconstructed by a depth-first traversal of the WPP (where the children of a node are chosen left-to-right). Given the WPP of some string σ and an edge e in this WPP, consider the directed acyclic graph rooted at the source node of e. Now consider the depth-first traversal of this directed acyclic graph until e is encountered. Let the string of terminal symbols constructed via this depth-first traversal be called σ_e. We now annotate the compacted trace of thread P_i as follows. Each edge e in P_i's Whole Program Path is annotated with a synchronization count $synch\_cnt(e)$. It denotes the number of synchronization operations executed by P_i in the execution sequence σ_e.

[1]To distinguish various acquires/releases of the same lock variable as distinct messages, we can annotate the message with an id, *i.e.* $P_i!M, l^{(k)}$ and $P_i?M, l^{(k)}$ denote the k-th release/acquire of lock variable l.

```
Data   : WPP of Thread $P_i$; Order of Synchronization
         Operation $m$.
Result : Root to leaf path $Path_m$ of the $m$th synchro-
         nization operation by $P_i$ in its WPP.

X:= root of WPP; $Path_m$ := $null$; $val$ := $m$;
while $X \neq$ a leaf node do
    let $e_1, \ldots, e_k$ be the outgoing edges of $X$;
    find smallest $i$ s.t. $synch\_cnt(e_{i+1}) \geq val$;
    if no such $i$ exists then
        set $i := k$
    end
    append $e_i$ to $Path_m$;
    set $val := val - synch\_cnt(e_i)$;
    set $X$ := destination node of $e_i$;
end
return $Path_m$;
```

Algorithm 1: Identifying the root-to-leaf path of a particular synchronization operation

Figure 2 illustrates this representation with an example. Only the control flow in each thread is shown. Thus, the alphabet of the WPP of P_i is Σ_i = set of basic blocks in thread P_i. The numbers $1 \ldots 6$ in the execution trace represent the basic blocks in the two threads (all of the code is not shown). Basic blocks $3, 4$ in Thread 1 and basic blocks $1, 2$ in Thread 2 contain synchronization operations. The count annotations on the edges of the WPP represent the *synch_cnt*. To illustrate these annotations consider the bold edge $S \to A$ in Figure 2; let this edge be e. Then, by the preceding definition σ_e is the string $\langle 1, 2, 3, 4, 5 \rangle$. Since there are two synchronization operations in this sequence, therefore $synch\_cnt(e) = 2$ (as shown in the figure). On the other hand, consider the bold edge $A \to 4$ in Figure 2; let this edge be e'. Then, $\sigma_{e'}$ is the sequence $\langle 3 \rangle$, and hence $synch\_cnt(e') = 1$.

Given this annotation, it is easy to associate the synchronization events with the control/data flow trace. Suppose, we are interested to find out the execution trace between a lock operation and the corresponding unlock operation of a thread P_i. Let the lock and unlock be the m^{th} and n^{th} $(n > m)$ synchronization operations executed by thread P_i in S_{synch} respectively. We can now use the *synch_cnt* to identify the root to leaf paths $Path_m$ and $Path_n$ for the two synchronization operations in the WPP of thread P_i (refer Algorithm 1). Let X be the lowest common non-terminal node of the two paths $Path_m$ and $Path_n$. Then the subtree generated by X gives the execution trace of P_i between the lock and the corresponding unlock operation. Thus, the synchronization counts capture the association between synchronization events and control flow/shared data access, which allows random navigation of the execution trace.

2.3 Control and/or Data Flow

To complete the description of our compact representation, we now need to fix an alphabet Σ_i. The control/data flow in thread P_i is defined as a string over Σ_i. This string can be stored in a compacted format using a SEQUITUR grammar representation. We consider three different compact representations.

- *Control only representation (COR)* Representing only control flow in each thread.
- *Data only representation (DOR)* Representing shared data accesses in each thread.
- *Control + Data representation (CDR)* Representing combined control flow and shared data accesses in each thread.

We now elaborate on each of these representations.

COR Representation. The COR representation of an execution of parallel program $P_1 \parallel \ldots \parallel P_n$ consists of:

WPP The control flow in each thread P_i is represented as a Whole Program Path (WPP) [11] over the alphabet Σ_i = the set of basic blocks in thread P_i.

$\mathbf{S_{synch}}$ Synchronization operations are stored as a sequence of entries of the form: $\langle threadId, \texttt{lock}, lockVar \rangle$ or $\langle threadId, \texttt{unlock}, lockVar \rangle$ or $\langle \texttt{barrier}, barrierVar \rangle$.

synch_cnt Each edge $s \to t$ in P_i's WPP is annotated with a synchronization count $synch\_cnt(s \to t)$.

Note that for barriers in S_{synch}, we do not maintain the thread identifier. A barrier operation is placed in the total order S_{synch} when the last thread reaches that barrier. The above scheme can be extended to threads with *fork / join* operations by augmenting thread P_i's WPP with a unique $start_i$ and end_i node. The effect of fork and join can then be represented by storing incoming edges into $start_i$ and outgoing edges from end_i respectively.

DOR Representation. We can represent the data accesses of a parallel program execution as follows. As before, let the parallel program be $P_1 \parallel \ldots \parallel P_n$.

- The sequence of data accesses in each thread P_i is compressed using SEQUITUR. The alphabet of the string denoting the execution is Σ_i, defined as follows. Σ_i consists of synchronization operations and shared memory operations in thread P_i. The synchronization operations consist of lock, unlock and barrier entries as shown in COR Representation. The shared memory operations are of the form $\langle \texttt{load}, loc \rangle$ or $\langle \texttt{store}, loc \rangle$ where *loc* is a shared memory location. Note that a single load/store instruction in a thread may access different locations at different points of time in an execution (*e.g.* if the instruction involves array access). This will lead to different letters in our alphabet.
- Each edge $s \to t$ in the WPP of P_i is annotated with a synchronization count $synch\_cnt(s \to t)$ as in COR representation.
- The synchronization pattern *across* threads is stored separately as a total order of synchronization operations as in COR representation.

CDR Representation. Finally, we consider another representation for parallel program executions, where the shared data access pattern for each load/store instruction is considered. This can be important for software pre-fetching. If we know the data access pattern for a particular instruction, the compiler can add prefetch instructions before them (such as adding the instruction `prefetch A[i+1]` before `load A[i]`

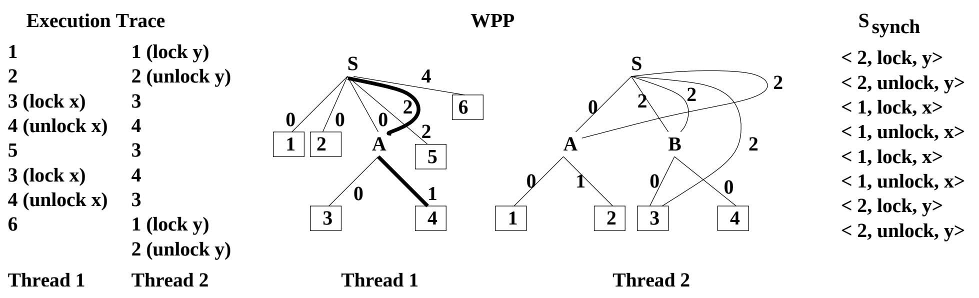


Figure 2: Synchronization Count Annotations

if A is an array which is accessed sequentially). Knowing the data access pattern for an instruction I is also useful for finding instructions in other threads which have a potential data race with I.

The shared data access and control flow in a parallel program execution is represented as follows.

WPP The control flow in each thread P_i is stored as a Whole Program Path as in COR Representation. The set of terminals of P_i's WPP is the set of basic blocks of P_i and is called Σ_i.

Synchronization The synchronization pattern across the various threads is stored as in COR Representation.

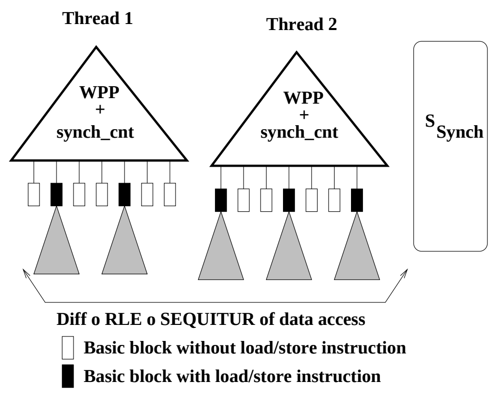
Data access per instruction For each terminal $a \in \Sigma_i$ (representing a straight-line code fragment in thread P_i), let $\mathcal{I}_a$ be the shared data load/store instructions in a. Then, for each $I \in \mathcal{I}_a$, we store the sequence of memory locations accessed by I as a WPP.

2.4 Achieving Further Compaction

The CDR representation serves as a unified description of control flow and shared variable accesses. The memory locations accessed by a load/store instruction I can be simply compacted into a WPP using the SEQUITUR grammar. However, the regularity of the memory locations accessed by an instruction in a program can be exploited to achieve further compaction. We first describe how this is achieved and then discuss potential uses.

Our compression of the sequence of memory locations accessed by a shared data load/store instruction is shown in a phase-by-phase manner. However, the effect of all the three phases can be achieved by an *online* algorithm.

1. *Diff:* The sequence of memory locations is converted into a difference representation. Thus, the sequence $X_1, X_2, \ldots, X_k$ is converted to the sequence
$$X_1, X_2 - X_1, X_3 - X_2, \ldots, X_k - X_{k-1}$$
2. *RLE:* The difference representation of Step 1 is compacted using run-length encoding (*RLE*), that is, by recording contiguous repeated occurrences of the same difference. If the sequence of memory locations accessed is $10, 14, 18, 22, 42$, then at the end of step 1 we have $10, 4, 4, 4, 20$ and at the end of step 2 we have $10^1, 4^3, 20^1$. This means that 10 occurs once, followed by 4 occurring thrice, and so on.



▯	Basic block without load/store instruction
▮	Basic block with load/store instruction

Figure 3: CDR representation

3. *SEQ:* The SEQUITUR compression algorithm [14] is applied over the run-length encoded string obtained in step 2.

To achieve the effects of the above steps in an online algorithm we need to feed symbols into SEQUITUR only when we reach the end of a run of identical differences. For the string $10, 14, 18, 22, 42$ we first send 10^1 to SEQUITUR. Subsequently we encounter a difference of 4; however we keep on consuming symbols until we reach 42 (when the difference is no longer 4).

Figure 3 depicts the CDR representation. Let us now explain the rationale for our representation. Step 1 (*Diff*) is motivated by the fact that an instruction often involves accesses to successive elements of an array (whose memory addresses are in Arithmetic Progression). Step 2 (*RLE*) is needed for the following compression inefficiency of the SEQUITUR algorithm. Let us consider a string $x, x, \ldots, x$ (the same symbol x repeated 2^k times, say). SEQUITUR introduces k production rules of the form

$$S \to A_1 A_1 \quad A_1 \to A_2 A_2 \quad \ldots \quad A_{k-1} \to xx$$

In step 2 (*RLE*), we will simply represent it as a single symbol x^{2^k}. Finally, employing the SEQUITUR algorithm over such composite symbols in step 3 (*SEQ*) allows us to compactly represent (a) repeated accesses to the same portion of an array, or (b) repeated accesses to different portions (which are of the same length) of a multidimensional array. This is often the case, *e.g.* if the instruction is of the form `store A[j][k]` occurring within a nested loop as shown in the following code fragment. We assume that the dimen-

Benchmark	Source	Description	Input size
FFT	SPLASH-2	1-D fast Fourier transform	2^{16} complex data points
LU	SPLASH-2	Parallel dense blocked LU factorization	256×256 matrix
Mp3d	SPLASH	Particle flow in simulated wind tunnel	50 K particles
Water	SPLASH	Molecular dynamics simulator	64 molecules
SOR	Treadmarks	Red-black successive over-relaxation on a grid	100×100 grid

Table 1: Description of benchmark applications.

sions of matrix A are 1..X, 1..Y where M1 > 1, N1 > 1, M2 < X and N2 < Y. Thus, the code sweeps through a sub-grid of the array.

```
for (j = M1; j <= M2; j++) {
     for (k = N1; k <= N2; k++) {
            A[j][k] = ...
            ...
     }
}
```

In the above code fragment, the shared memory accesses

$$\begin{aligned}&\langle A[M1][N1]\ldots A[M1][N2]\rangle,\\&\langle A[M1+1][N1]\ldots A[M1+1][N2]\rangle,\\&\ldots\\&\langle A[M2][N1],\ldots A[M2][N2]\rangle\end{aligned}$$

are mapped to $(M2-M1+1)$ occurrences of d^{N2-N1} where d is the size of a cell in the matrix A. Note that here d^{N2-N1} will be treated as a terminal symbol of the grammar. The occurrences of the composite terminal d^{N2-N1} are then further compressed by the SEQUITUR algorithm. This allows us to compactly represent multidimensional array accesses which is very common in many scientific computing benchmarks. In particular, if a matrix is operated by several threads in parallel and each thread is allocated a rectangular region of the matrix, then the above situation is likely to happen. This is because all the memory addresses accessed by an instruction `load A[j][k]` or `store A[j][k]` in a thread are not contiguously placed in the memory layout of the matrix. Instead they correspond to runs of contiguous positions in the matrix and these runs are typically of the same length.

3. COMPRESSION EFFICIENCY

The main goals of this work are (1) compact representation of the execution trace of a parallel program and (2) using this representation in profile-driven debugging and optimization. In this section, we quantitatively evaluate the compression efficiency of our representations. The next section describes how the representations can be used to facilitate profile-driven debugging.

3.1 Benchmarks & Methodology

We used five different shared-memory parallel applications for our study. The description of the applications is given in Table 1. Out of these, `FFT` and `LU` are from SPLASH-2 benchmark applications suite [22], `Mp3d` and `Water` are from SPLASH benchmark suite [21], and SOR is from TreadMarks benchmark applications [9]. The inputs used to generate the execution traces are also shown in Table 1.

To collect execution traces, we ran each of these applications on RSIM-1.0 simulator [8]. RSIM is an execution-driven simulator that models shared-memory multiprocessors with aggressive state-of-the-art ILP processors, aggressive memory system, and multiprocessor cache coherence protocol. As we are interested only in the execution sequence, we configured RSIM processors with simple in-order execution units. We used versions of the applications that have been ported for RSIM. All the applications were compiled and linked for SUN SPARC V9.

Identification of shared memory accesses requires some clarification. In RSIM, shared memory is allocated dynamically starting from a "base address". All the load/stores accessing addresses above the base address will be marked as shared memory references. For the control flow, our tool takes in the binary executable and identifies the basic blocks. During execution, the instruction addresses are matched against basic block boundaries and the basic block number is sent to the compression algorithm.

To evaluate the scalability of our representations, we ran each benchmark application on 2, 4 and 8 processors. The characteristics of the benchmark applications for the different number of processors are given in Table 2. The column *Shared Mem. Size* represents the maximum amount of shared memory allocated for the applications. As can be seen from Table 2, the maximum amount of shared memory allocated hardly increases with increasing number of processors if the input size is kept constant. The columns *Barriers* and *Locks* represent the number of dynamic barriers and lock/unlock pairs executed by all the processors. `LU`, `FFT` and `SOR` use only barriers for synchronization whereas `Mp3d` and `Water` use both barriers and locks for synchronization. Again, note that the number of lock/unlock pairs executed remains almost the same as we increase the number of processors while keeping the input constant. The number of

Pgm.	Proc.	Shared Mem. Size (bytes)	Barriers	Locks
FFT	2	3,220,200	14	0
	4	3,228,904	14	0
	8	3,295,528	14	0
LU	2	526,220	4	0
	4	526,660	4	0
	8	526,660	4	0
Mp3d	2	2,378,040	13	1,103
	4	2,379,064	13	1,104
	8	2,381,240	13	1,113
Water	2	50,032	6	2,090
	4	50,032	6	2,100
	8	50,032	6	2,120
SOR	2	53,256	5	0
	4	53,384	5	0
	8	53,640	5	0

Table 2: Characteristics of benchmark applications.

barriers executed of course remains the same irrespective of the number of processors.

3.2 COR Representation Efficiency

We first present the compression efficiency of COR representation. Recall that the COR representation only captures the control flow and the synchronization operations. Table 3 shows the compression efficiency for the different benchmarks with 2, 4 and 8 processors. The column *Orig. Trace* represents the cost of storing the uncompressed basic block level execution trace, whereas the column *WPP* represents the storage requirement after compressing the *Orig. Trace* using SEQUITUR. For parallel programs, in addition to the WPPs for each processor, we also need to store the order of the synchronization operations. The column S_{Synch} shows the cost of storing the total order of synchronization operations across threads. As `Mp3d` and `Water` have lot of synchronizations, they require more space for S_{Synch}. The first % column represents the space requirement of $WPP + S_{Synch}$ as a percentage of *Orig. Trace*. For larger benchmarks, it is less than 1% and even for smaller benchmarks, it is always under 9%. Notice that with increasing number of processors, the size of *Orig. Trace* remains almost the same, but the *WPP* size increases by a small amount. It shows that temporal and spatial locality decrease per processor as we employ more processors. However, the compression ratio does not change significantly.

Finally, we also include the cost of annotating the WPP edges with *synch_cnt* to facilitate random access into the execution trace given a synchronization point. The last column presents the space requirement of $WPP + S_{Synch} + synch\_cnt$ as a percentage of *Orig. Trace*. As can be seen from the table, including the annotations on the edges has very little space overhead, but makes the representation more flexible for navigation.

3.3 DOR Representation Efficiency

Table 4 shows the compression efficiency of the DOR representation for the different benchmarks with 2, 4 and 8 processors. The column *Orig. Trace* represents the uncompressed shared memory access trace of all the processors. As with control flow, the trace size increases very little with more processors. The column *SQ* represents the space requirement after *Orig. Trace* is compressed using SEQUITUR. Again S_{Synch} represents the cost for storing total order of the synchronization operations and the next percentage column indicates $SQ + S_{Synch}$ as a percentage of *Orig. Trace*. Notice that compared to COR representation where the compression ratio was less than 9% for all the benchmarks, it is at least 35% for DOR representation and can be as bad as 79%. These results indicate that there is very little locality of access to be exploited by SEQUITUR in global shared memory access patterns. As with COR representation, we annotated the edges with *synch_cnt*. The final column of $SQ + S_{Synch} + synch\_cnt$ as percentage of *Orig. Trace* shows that the efficiency deteriorates to the point of making the compressed version take more space than uncompressed version for `Water`. The CDR representation alleviates this problem.

3.4 CDR Representation Efficiency

In this subsection, we study the efficiency of CDR representation which captures combined data and control flow.

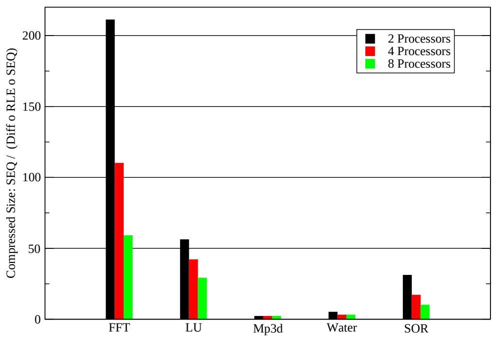


Figure 4: Advantage of CDR Representation (Both SEQ and Diff ∘ RLE ∘ SEQ were performed on the data access sequences of individual load/store instructions)

The column *Orig. Trace* represents the total storage cost for the uncompressed control flow trace in terms of basic blocks and the uncompressed shared memory access trace. The *Compressed Trace* column shows the compressed size of these two components and their total. Recall that the cost to store control flow and synchronization is identical to COR representation. The only additional cost is to store the shared memory access pattern per load/store instruction after compression with our three phase *Diff ∘ RLE ∘ SEQ* scheme. Notice that compared to Table 4, which compresses the complete shared memory access trace per processor using SEQUITUR, this separate compression per load/store instruction achieves significantly better compression ratio (compare column *SQ* in Table 4 with the column *Compressed Trace → Data* in Table 5). The only exception is the `Mp3d` benchmark where no significant space savings is obtained by compressing the memory access pattern of each load/store instruction separately. Except for `Mp3d`, other benchmarks achieve about 10–200 times more compression with *Diff ∘ RLE ∘ SEQ*. `Mp3d` simulates particle flow which performs completely random data access and hence cannot be captured easily using our compression technique. The compression efficiency for control and data together in Table 5 varies from 0.25%–9.81% — a significant improvement over DOR representation.

To show that the main advantage of our scheme comes from the *Diff ∘ RLE* pre-processing step, Figure 4 compares the compression efficiency of pure SEQUITUR with *Diff ∘ RLE ∘ SEQ* (refer Section 2.4 to see the transformations in *Diff*, *RLE* and *SEQ* steps). In both cases, we apply compression separately on the shared memory access trace of each individual load/store instruction. *Diff ∘ RLE ∘ SEQ* achieves as high as 211 times size difference compared to SEQUITUR for `FFT`. The improvement is impressive for `SOR` and `LU` as well. For `Water`, the size difference is 3–5 times, whereas for `Mp3d` the difference is 2 times. Notice that the size difference decreases with increasing number of processors indicating the decrease in regularity of access pattern with increasing processor count.

Benchmark	Proc.	Orig. Trace	WPP	$S_{Synch.}$	%	synch_cnt	%
FFT	2	66.06 M	406.32 K	28	0.62	84.89 K	0.74
	4	66.16 M	424.58 K	28	0.64	88.15 K	0.78
	8	66.37 M	453.70 K	28	0.68	93.44 K	0.82
LU	2	54.73 M	229.35 K	8	0.42	47.75 K	0.51
	4	55.32 M	244.81 K	8	0.44	50.54 K	0.53
	8	56.50 M	281.65 K	8	0.50	57.24 K	0.60
Mp3d	2	67.42 M	231.77 K	15.49 K	0.37	92.86 K	0.50
	4	67.42 M	259.79 K	15.51 K	0.41	103.16 K	0.56
	8	67.43 M	295.36 K	15.64 K	0.46	116.25 K	0.63
Water	2	5 M	81.01 K	29.77 K	2.22	30.15 K	2.82
	4	5.01 M	97.03 K	29.91 K	2.53	35.87 K	3.25
	8	5.02 M	118.98 K	30.19 K	2.97	43.77 K	3.85
SOR	2	90.09 K	2.69 K	10	3.00	474	3.50
	4	91.01 K	4.79 K	10	5.30	837	6.20
	8	92.80 K	7.86 K	10	8.56	1385	10.04

Table 3: Compression Efficiency for COR Representation. All the sizes are in bytes.

Benchmark	Proc.	Orig. Trace	SQ	$S_{Synch.}$	%	synch_cnt	%
FFT	2	77.68 M	27.34 M	28	35.20	5.19 M	41.88
	4	77.69 M	27.74 M	28	35.70	5.19 M	42.39
	8	77.72 M	27.97 M	28	35.99	5.20 M	42.68
LU	2	89.42 M	56.10 M	8	62.74	12.24 M	76.43
	4	89.42 M	56.20 M	8	62.85	12.24 M	76.54
	8	89.42 M	56.24 M	8	62.89	12.24 M	76.58
Mp3d	2	19.84 M	8.71 M	15.49 K	43.98	2.88 M	58.50
	4	19.83 M	9.45 M	15.51 K	47.74	2.87 M	62.20
	8	19.84 M	10.05 M	15.64 K	50.73	2.87 M	65.20
Water	2	1.15 M	833.52 K	29.77 K	75.07	324.81 K	103.31
	4	1.15 M	849.75 K	29.91 K	76.50	324.84 K	104.74
	8	1.16 M	883.82 K	30.19 K	78.80	324.90 K	106.81
SOR	2	907.34 K	392.71 K	10	43.28	76.97 K	51.77
	4	908.58 K	393.83 K	10	43.35	77.02 K	51.83
	8	908.06 K	394.94 K	10	43.50	77.12 K	52.00

Table 4: Compression Efficiency for DOR Representation. All sizes are in bytes.

Pgm	Proc.	Orig.	Compressed Trace				gzipped trace			
		Trace	COR	Data	Total	%	Control	Data	Total	%
FFT	2	143.74M	491.24K	144.28K	635.52K	0.44	827.47K	13.78M	14.61M	10.16
	4	143.85M	512.76K	276.21K	788.97K	0.55	3.32M	12.58M	15.90M	11.05
	8	144.09M	547.17K	521.46K	1.07M	0.74	3.46M	12.34M	15.80M	10.97
LU	2	144.15M	277.10K	83.42K	360.52K	0.25	1.88M	2.93M	4.81M	3.34
	4	144.74M	295.36K	123.36K	418.72K	0.29	3.76M	3.05M	6.81M	4.71
	8	145.92M	338.89K	189.78K	528.67K	0.36	5.17M	2.78M	7.95M	5.45
Mp3d	2	87.26M	340.12K	8.09M	8.43M	9.66	1.03M	6.20M	7.23M	8.29
	4	87.25M	378.46K	8.10M	8.48M	9.72	1.16M	6.11M	7.27M	8.33
	8	87.27M	427.25K	8.13M	8.56M	9.81	1.31M	4.47M	5.78M	6.62
Water	2	6.15M	140.93K	85.55K	226.48K	3.68	292.87K	108.75K	401.62K	6.50
	4	6.16M	162.81K	132.58K	295.39K	4.79	563.49K	130.98K	694.47K	11.20
	8	6.18M	192.94K	190.18K	383.12K	6.2	676.05K	171.05K	847.10K	13.75
SOR	2	997.46K	3.17K	10.91K	14.09K	1.41	1.10K	71.73K	72.83K	7.30
	4	999.59K	5.64K	20.15K	25.79K	2.58	1.57K	77.61K	79.18K	7.92
	8	1M	9.25K	32.52K	41.77K	4.18	2.69K	87.24K	89.93K	9.00

Table 5: Compression Efficiency for CDR Representation. All the sizes are in bytes.

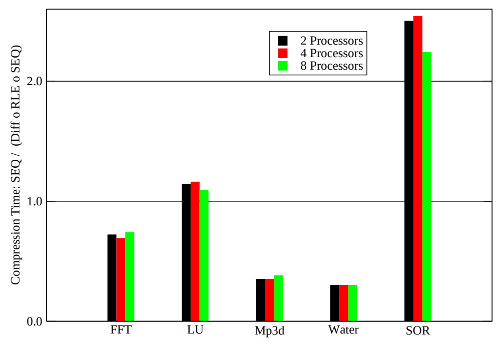


Figure 5: Compression Timing Overhead for CDR representation. (Both SEQ and Diff ∘ RLE ∘ SEQ were performed on the data access sequences of individual load/store instructions)

3.5 Timing Overheads

The pre-processing stages *Diff* ∘ *RLE* add extra overhead for `Water`, `FFT`, and `Mp3d` as shown in Figure 5. For `SOR` and `LU` our scheme is actually faster than SEQUITUR as the pre-processing brings out more locality in the string. For all the benchmarks, the absolute time taken for compression using our scheme varies from 1 second to at most 16 minutes. The measurements were taken on a Pentium III 1 GHz machine with 906 MB of memory.

Note that our experiments were conducted using the RSIM simulator, not a real multiprocessor. One possible concern is that in a real multiprocessor the tracing overheads can distort the program behavior, thereby producing a different execution. A solution to this problem is to trace minimal number of operations in a parallel program execution so as to ensure deterministic replay ([12] shows that less than 2% of the shared memory operations usually need to be traced). Then, our compressed path profile can be collected during the deterministic replay.

3.6 Comparison with Gzip

Table 4 shows the comparison of our CDR compression scheme against LZ77 [25], a classic data compression algorithm proposed by Ziv and Lempel. We use standard *gzip* utility based on LZ77 coding to compress the control flow and the shared memory access pattern of individual load/store instructions. The compression efficiency of control flow with SEQUITUR is significantly better than gzip except for `SOR`. For the shared memory access pattern per load/store instruction, our three phase *Diff* ∘ *RLE* ∘ *SEQ* scheme performs better than gzip in all benchmarks except `Mp3d`. As mentioned before, `Mp3d`'s random access pattern was not amenable to our scheme. Overall, the compression efficiency of gzip for control and data together varies from 3.34%–13.75% — much worse than CDR representation. Moreover, gzip will not allow random access of the compressed control and data flow representation to identify data races as discussed in the next section.

4. APPLICATIONS

So far we have studied compressed representations of executions of shared memory parallel programs. One point needs to be noted in this context. For describing the execution in each thread, we have considered three alternate representations in Section 2 which store different information. COR representation describes control flow by storing the basic blocks executed, while DOR representation stores the sequence of shared memory data accesses executed in a thread. The CDR representation explicitly stores the sequence of shared memory data accesses per instruction (rather the overall sequence of shared memory data accesses in a thread). If it is not possible to store all of the three representations, we believe that it is crucial to store the control flow as well as shared memory data accesses per thread. Hence, we recommend that a parallel program execution be stored using either (a) COR and DOR representations, or (b) CDR representation.

4.1 Apparent Data Race Detection

We now discuss how our representation of a parallel program execution can be used to perform accurate data race detection in a post-mortem fashion. The CDR representation is sufficient for these purposes.

For the convenience of the reader, we recall that an apparent data race exists between two shared variable reads/writes a and b of a parallel program if (1) a and b occur in different threads of the program (2) a and b operate on the same shared variable and at least one of them is a write, and (3) they are not prevented from happening in parallel by synchronization operations. There is a rich literature on data race detection for parallel and multithreaded programs, see [5, 7, 13, 17, 18, 20] for example.

For finding the set of all apparent data races post-mortem, the usual technique divides the execution in each thread into "synchronization segments", that is, segments of code between synchronization operations [18]. The task then is to find out the various synchronization segments in different threads which may happen in parallel. We can traverse either the CDR or the DOR representation of an execution profile to compute: (a) synchronization segments in threads, and (b) the conflicts among these segments.

Note that apparent data race detection involves detecting the sequence of acquires and releases (or locks and unlocks) executed on a particular lock variable. For example, consider the execution shown in Figure 6. Suppose we want to find which segment in thread 1 is in conflict (may happen in parallel) with the segment marked in bold in thread 2. For this purpose, we need to compute the following in thread 1:

- last unlock in thread 1 before $(lock, 2)$, say u_1.
- the lock occurring in thread 1 immediately after u_1

Now, the next lock of thread 1 as mentioned above turns out to be the second occurrence of lock in thread 1. Note that in the Whole Program Path of thread 1, *lock* and *unlock* are simply two terminal symbols. The individual occurrences of lock and unlock in thread 1 are *not* distinguished. However, we want to locate the segment of code between the first unlock and second lock in thread 1 to compute the segment marked in bold in thread 1 (refer Figure 6). This segment of thread 1 is in conflict with the segment marked in bold in thread 2. We want to compute this segment in thread 1 without decompressing the WPP of thread 1. This can be achieved by identifying the root-to-leaf path of the respective lock and unlock occurrences using Algorithm 1.

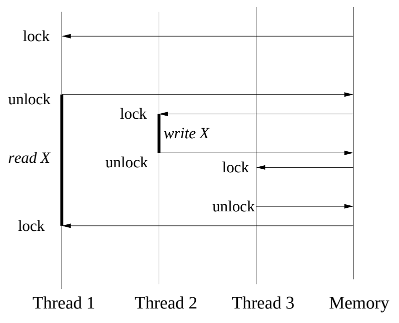


Figure 6: An apparent data race

Using our compressed representation, we have performed post-mortem detection of apparent data races. Some of the benchmarks (including LU) reported large number apparent data races. In particular, the LU benchmark (with 2 threads) reported 63,832 apparent data races in 6 seconds on a Pentium III 1 GHz machine with 906 MB of memory. Note that a single pair of operations on a shared variable may be executed many times in a parallel program execution, each time being reported as a data race. In particular, the 63,832 data races in the LU benchmark correspond to only five distinct pairs of conflicting instructions in the two threads of the benchmark. To avoid this problem, we can use the CDR representation instead of DOR representation for data race detection. In the CDR representation, we mark the instruction corresponding to a shared data access. Consequently replicated data race reports from the same pair of instructions can be avoided.

4.2 Detecting Data Race Artifacts

Consider the parallel program execution in Figure 7 where `A` is a shared array and `Subscript` is a shared variable. Initially we assume `Subscript` $= 0$. Note that this code does not contain any synchronization operations. The two data race pairs exhibited in this code fragment are: (1) the reading and writing of `Subscript`, (2) the reading and writing of `A[1]`. Clearly, the second one happens due to the occurrence of the first data race. In particular, the value of `Subscript` is propagated to `x` which controls the shared memory location accessed by `A[x]`. In this case, it is more meaningful to report only the first data race pair, since the second data race may not even occur (`A[1]` may not be even read in the second thread) if the first data race does not occur. Hence, the second data race is called an *artifact* of the first data race. Detecting and not reporting artifact data races reduces the burden on the programmer.

```
Thread 1           Thread 2
Subscript := 1;    x := Subscript
...                ...
A[1] := 0 ;        y := A[x]
```

Figure 7: Example of an artifact data race

The above notion of data race artifact has been formally studied by Netzer and Miller in [13]. Roughly speaking, they consider an *event-control* dependence relation E s.t. $(a, b) \in E$ if

1. a writes a shared variable which is used by b (directly or through other variables) to determine which locations to access, or
2. a precedes b in the same thread, or
3. there exists c such that $(a, c) \in E$ and $(c, b) \in E$.

The event-control dependence is defined on events which in practice is a fragment of sequential code (simply a single instruction at the finest level of granularity). Note that by the above definition the write of `Subscript` event-controls both the read of `A[x]`, as well as the write of `A[1]`. Under this condition, the second data race (involving the read/write of `A[1]`) is deemed to be an artifact of the first. Formally a data race pair (c, d) is an artifact of another data race pair (a, b) if $(a, c) \in E$ and $(a, d) \in E$, or $(b, c) \in E$ and $(b, d) \in E$. In other words, either a or b event-controls both c and d.

The above definitions provide a clean way of detecting artifact data races. This reduces the number of data races reported to the programmer and hence is more useful in detecting potential bugs in the program. Computing the event-control dependences however depends on data flow information. This is captured in our representation of parallel program execution. In particular, in the execution of Figure 7 we can infer that `Subscript := 1` event-controls the read of `A[x]` since: (1) the read of `A[x]` uses local variable `x` which is set by the statement `x := Subscript`, and (2) the statement `x := Subscript` can occur after `Subscript := 1` in Thread 1 (it is not constrained otherwise by synchronization operations). Clearly, this inference can be mechanized, allowing us to automatically compute the E relation. The compact nature of our representation allows us to store the sequence of shared variable accesses. This can be exploited in the computation of the event control dependence, and hence for accurate data race detection.

5. RELATED WORK

Execution profile-driven code optimization has been studied in the programming languages, compiler and architecture community for many years. We review some of the relevant literature in this section.

Work on Path Profiles. Due to the huge size of program traces, traditionally only short sub-paths of the execution path (describing the behavior of the program in a specific procedure, say) used to be captured. Recently, Larus showed that the control flow in a program's entire execution path can be stored by employing online data compression techniques [11]. The SEQUITUR algorithm was adapted by Larus to represent the complete control flow information in a sequential program execution. Similar trace compaction techniques have been employed in [16] for the purposes of program understanding and software visualization. Chilimbi studied how variants of SEQUITUR can be used for representing data access patterns in a sequential program execution [4]. This is useful for the purposes of memory layout (*e.g.* which blocks should map to the same cache line) and enables software pre-fetching. Zhang and Gupta [24] suggested compact representations where the execution trace is broken up into per-function path traces (thereby enabling post-mortem retrieval of execution traces of specific functions). In particular, a dynamic call graph is maintained to record the sequence of function calls/returns and for each function, the various path traces encountered in its different calls are compacted.

In our work, we have studied compressed representations for parallel program traces. Parallel programs have two different kinds of structuring: the per-function structuring in each thread (as in sequential programs) and the per-thread structuring. We have concentrated on supporting the per-thread structuring by representing each thread as a Whole Program Path (WPP). Each WPP may contain information pertaining to synchronization as well as control and/or shared memory data flow. Furthermore, we have studied two kinds of shared data access patterns in an execution: the global shared data accesses in each thread, as well as the shared data accessed by each instruction. For the latter (which we call the CDR representation), it is often advantageous to represent the memory accesses in a difference representation and perform run-length encoding of the differences before employing the SEQUITUR algorithm. In other words, even though we found the SEQUITUR algorithm to be useful for program path compression, it is often necessary to manipulate the alphabet of the string being compressed (*i.e.* the set of terminal symbols using which the program path is described).

Recently, we have been pointed to a work on compressing instruction and data address traces in a sequential program [15]. Unlike our method, this is a two pass method which creates a descriptor for each basic block in the first pass. It uses the attributes in this descriptor in the second pass to create a compressed trace. This compression technique also identifies the unique offsets between successive data addresses referenced by an instruction. However, this technique cannot afford to collect the entire trace of data references by an instruction (as is done in our CDR method). This is because the method does not employ any on-line compression scheme and the entire trace is too big to be stored in memory. Instead, the compressed trace simply intersperses the instruction and data references, where data references indicate which unique offset is used.

Work on Deterministic Replay. Most of the work in parallel program tracing is geared towards deterministic replay of executions (reconstructing and replaying an execution from recorded information) [10, 12, 19]. This is primarily useful for program debugging and comprehension; it does not require recording all shared memory accesses. However, our aim is not only to store enough information so that the current execution can be "replayed". Indeed, one of the intended uses of our compact representation is memory performance optimization. Our parallel program execution can serve as a profile which is used to optimize some of the "artifactual communication" [6] (additional communication between levels of the shared memory hierarchy which takes place as an artifact of the program's interaction with the memory organization). Note that our compressed representations can also be used for deterministic replay of properly synchronized parallel programs. In programs which are not properly synchronized, we need the data dependences between unsynchronized shared variable reads and writes for exact replay. These dependences are not captured in our representation.

Work on Data Race Detection. In Section 4, we discussed how the precise data flow information maintained by our compact traces can be exploited for accurate dynamic data race detection. Dynamic data race detection techniques are typically categorized into two classes: on-the-fly [5, 7, 20] and post-mortem [1, 13, 18]. One of the major drawbacks of the post-mortem method is the huge size of the trace. This problem can be alleviated using our compact trace representations. In fact, the compressed nature of our traces allows us to instrument/record more information during execution: the exact order of the shared variable accesses in each thread. This additional data flow information enables more accurate post-mortem analysis of data races.

6. DISCUSSION

Execution profiles are important for analysis, optimization as well as debugging since they provide valuable information about a program's dynamic behavior. In this paper we have presented compression schemes for representing execution profiles of shared memory parallel programs. Our representation captures control/data flow and synchronizations in the execution of a shared memory multithreaded program running on a multiprocessor architecture. Our experimental results show substantial compaction in SPLASH benchmark traces using a representation which combines control and data flow. We also present evidence of scalability of the compression efficiency with increasing number of processors. Finally, we have studied how this compact path profile can be used for post-mortem program debugging such as detecting data races.

The execution trace of a shared memory parallel program should maintain the control/data flow of each processor as well as the their interactions during execution. In this paper, the control/data flow of each processor is maintained individually as Whole Program Paths (WPP). The total order of the synchronization operations executed by all processors and the annotation of each processor's WPP with synchronization counts help us to capture those inter-processor communications which are protected via synchronization primitives such as lock, unlock, and barriers.

However, our compact trace currently does not capture the actual order of unsynchronized shared variable accesses by different processors. Maintaining the order of all inter-processor communication (via shared variable accesses) can lead to prohibitively high storage requirement. Fortunately, it is not necessary to maintain every inter-processor communication for capturing a parallel program execution. Netzer's work [12] shows that it is sufficient to maintain the *transitive reduction* of the program order on each processor and the shared variable conflict orders. Moreover, he proposed an on-the-fly technique to identify the orderings in the transitive reduction relation. It is possible to adapt our tracing mechanism to record these orderings in the transitive reduction as the only information about inter-processor communication (instead of recording the synchronization operations as is done currently).

We plan to investigate whether the order of operations appearing in the transitive reduction can be captured via annotations of the WPP edges (in a manner similar to our current handling of synchronization counts). This can lead to new uses of the compressed execution profiles. For example, given a compact program trace that captures the order of shared variable accesses across processors, we can use the trace to characterize whether an invalidation-based or update-based cache coherence protocol is suitable for the parallel program. An update-based protocol typically performs better if between two consecutive writes (updates) by a processor to the same variable, other processors read that variable. For any shared variable `v`, we can identify consecutive writes to `v` by processor P_i as follows. We look up the terminal node for `store v` in the WPP of P_i and its two consecutive incoming edges. Given these edges and their annotations, we find out the shared memory accesses performed by other processors during that interval. If the interval contains many read accesses of `v` by other processors, then an update-based protocol will perform better. Formalizing these observations remains a topic of future work.

7. ACKNOWLEDGMENTS

The reviewers' comments were helpful in revising the paper. This work was partially supported by National University of Singapore Research Project R-252-000-088-112. Ankit Goel was partially supported by a scholarship from IDA, Singapore.

8. REFERENCES

[1] S.V. Adve, M.D. Hill, B.P. Miller, and R.H.B. Netzer. Detecting data races in weak memory systems. In *ACM International Symposium on Computer Architecture (ISCA)*, 1991.

[2] R. Alur and R. Grosu. Shared variable interaction diagrams. In *International Conference on Automated Software Engineering (ASE)*, 2001.

[3] R. Alur, G.J. Holzmann, and D.A. Peled. An analyzer for message sequence charts. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), LNCS 1055*, 1996.

[4] T.M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *ACM International Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[5] J-D. Choi and S.L. Min. Race frontier: Reproducing data races in parallel program debugging. In *ACM International Conference on Principles and Practice of Parallel Programming (PPoPP)*, 1991.

[6] D.E. Culler, J.P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, 1999.

[7] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *ACM International Conference on Principles and Practice of Parallel Programming (PPoPP)*, 1990.

[8] C.J. Hughes, V.S. Pai, P. Ranganathan, and S.V. Adve. RSIM: Simulating shared-memory multiprocessors with ILP processors. *IEEE Computer*, 35(2), 2002.

[9] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Winter USENIX Conference*, 1994.

[10] R. Konuru, H. Srinivasan, and J-D. Choi. Deterministic replay of distributed java applications. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2000.

[11] J.R. Larus. Whole program paths. In *ACM International Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[12] R.H.B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993.

[13] R.H.B. Netzer and B.P. Miller. Improving the accuracy of data race detection. In *ACM International Conference on Principles and Practice of Parallel Programming (PPoPP)*, 1991.

[14] C.G. Nevill-Manning and I.H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7, 1997.

[15] A.R. Pleszkun. Techniques for compressing program address traces. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1994.

[16] S.P. Reiss and M. Renieris. Encoding program executions. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2001.

[17] M.C. Rinard. Analysis of multithreaded programs. In *Static Analysis Symposium (SAS)*, 2001.

[18] M. Ronsse and K. de Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2), 1999.

[19] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared memory applications. In *ACM International Conference on Programming Language Design and Implementation (PLDI)*, 1996.

[20] S. Savage et al. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[21] J.P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *ACM SIGARCH Computer Architecture News*, 20(1), 1992.

[22] S. C. Woo et al. The SPLASH-2 programs: Characterization and methodological considerations. In *International Symposium on Computer Architecture (ISCA)*, 1995.

[23] Z.120. Message Sequence Charts (MSC'96), 1996.

[24] Y. Zhang and R. Gupta. Timestamped whole program path representation and its applications. In *ACM International Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[25] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3), 1977.